McBasic

Build Desktop Apps with BASIC on the Mac

A step-by-step user manual

McBasic 0.7.881 · Build 881

English edition · Image instructions included

Copyright and edition

McBasic: Build Desktop Apps with BASIC on the Mac

First English edition

Author: ANIELLO CASTELLANO

Version covered: This edition describes McBasic™ 0.7.881 (Build 881).

Copyright

Copyright (c) **2026 – ANIELLO CASTELLANO –** All rights reserved.

No part of this book may be reproduced, stored, or distributed in any form without written permission from the copyright holder, except for short quotes used in reviews.

McBasic™ and related product names belong to their respective owners. This book is an independent user guide unless the copyright holder adds an approved publisher or product relationship here.

McBasic™ is a trademark of Aniello Castellano. Visual Basic is a trademark of Microsoft Corporation. McBasic™ is an independent product and is not affiliated with, sponsored by or endorsed by Microsoft.

Disclaimer

Software changes. The screens, menu names, examples, and supported features in this book may differ from a later McBasic™ release. Always check the release notes for the version you use.

The examples are for learning. Test an app with your own data before relying on it for important work. The author and publisher are not responsible for loss of data, interrupted service, or problems caused by changes to the software or the operating system.

About this book

This book teaches McBasic™ through small tasks. Each task explains what to do, what should happen, and what to check when the result is different. The figure notes explain what each screenshot or diagram is intended to show. They can be skipped when the same result is already visible in your own project.

Contents

Copyright 2
Disclaimer 2
About this book 2
Chapter 1 — Welcome to McBasic™ 25
Who this book is for 25
A pattern you will use often 25
A first look at a real project 26
What McBasic™ is not 27
Your first checkpoint 27
Expected result 27
Check your work 28
Small challenge 28
Tips and tricks 28
Chapter 2 — Install, open, and protect a project 29
Check the release you are using 29
The Welcome window 29
Recent Projects 31
Close a project 32
Open a project 32
Create a project 32
Know the important project files 33
Set the app's name, version and credits 33
Choose the appearance 34
Tabs in Classic 36
DTPicker and MonthView in Classic 37
Disabled controls in Classic 37

System colors....38
Choose the app icon....40
Make a working copy....41
Check your work....42
Expected result....42
Small challenge....42
Tips and tricks....42
Chapter 3 — Build your first working app....45
Before you start....45
Step 1: Open the sample....45
Step 2: Run the sample....46
Step 3: Read the button code....47
Step 4: Make a small change....47
Step 5: Open the second window....47
Check your work....48
Small challenge....48
The project that grows with the lessons....48
Tips and tricks....49
Chapter 4 — Find your way around the IDE....51
The Welcome window and the IDE window....51
The main areas....51
The toolbar....52
Toolbar settings....54
The Classic toolbar....54
The Ribbon....55
Adding documents to a project....56
Knowing that your work is saved....56
Design and Code....57
Documents, objects, and events....58
The document tab bar....58
The Properties area....60
Check before Run....61

If you are not sure where to start 61
Helpful editor tools 61
The AI panel 62
Expected result 63
Check your work 63
Small challenge 63
Tips and tricks 63
Chapter 5 — Design a form 65
Start with the user's task 65
Add a control 65
Choose useful names 66
Set the first important properties 67
Keep the layout easy to change 67
The Anchor editor 67
The common visual contract 67
Containers are part of the design 68
Right-click menus 69
Control arrays: many controls, one name 70
Make a form usable with the keyboard 73
Plan for accessibility while you design 73
Practice form: customer entry 75
Check your work 75
Tips and tricks 76
Chapter 6 — Write McBasic™ event code 77
Start with `Option Explicit` 77
Declare and change values 77
Choose a scope for each value 77
Separate assignment from comparison 78
Use `If` for a decision and `Select Case` for choices 79
Check a value before using it 79
Use control properties and methods 79
Events start procedures 80

One event procedure for a control array 80
Adding and removing members while the app runs 81
Subs and Functions 83
Learn the event families 84
Form life cycle 84
Pointer and keyboard input 85
Value, selection, and popup events 85
Drag and drop 85
Drag a control inside your app 86
DataGrid and TreeView events 87
Give a procedure one job 88
Give one procedure to the AI assistant 88
Open another form 89
Handle failures with `Try` and `Catch` 89
Find a bug with five questions 90
Use the language reference carefully 90
Expected result 90
Check your work 90
Small challenge 90
Tips and tricks 91
Chapter 7 — Choose the right component 93
Component groups 93
Text and input 93
Buttons and choices 93
Display and feedback 93
Containers 93
Data and menus 93
The complete standard catalog 93
A component's four parts 94
A few controls deserve special handling 95
CommonDialog 95
Line and Shape 95

TreeView and DataGrid....95
ValidationProvider and Timer....95
Ask four questions....96
Example: a simple search box....96
Shared properties....96
Check your work....98
Tips and tricks....98
Chapter 8 — Work with more than one form....99
Create a second form....99
`Show`, `Hide`, and `Unload`....99
Pass information to a form....100
Close a form from its own button....100
Plan your windows....101
Open a dialog and wait for the answer....101
Pass information to the dialog....102
Get the answer back....102
OK and Cancel....103
Lay out a dialog the Mac way....104
Confirm before deleting....104
Settings and About windows....105
What happens when windows close....105
Keep the work out of the forms....105
A small two-form exercise....106
Where a window opens....106
The window frame and its buttons....107
Window state, floating and transparency....108
Enabled, Visible, Tag, Cursor and AccessibleName....109
A background picture....110
Menus....111
The Menu Editor....111
The window....112
Rules the editor checks....113

The menu strip in the designer....114
Menus in the project tree and the code editor....114
A menu array....114
Convert an old Menu component....114
Current limits....115
Check your work....115
Tips and tricks....116
Chapter 9 — Organize your code....119
Form code....119
Shared modules....119
Use a form's controls from other code....119
Start the app from Sub Main....121
A splash window....121
A login window....122
When the app ends....122
Check....123
Classes....123
When a class starts and ends....123
A simple rule for moving code....124
Check your work....124
Tips and tricks....125
Chapter 10 — Handle input and validation....127
Validate before saving....127
React when text changes....127
Focus and selection....128
Read-only input....128
Keyboard actions....128
Practice exercise....128
Check your work....129
Tips and tricks....129
Chapter 11 — A small McBasic™ language toolkit....131
The BASIC habits that prevent most bugs....131

Choose a type deliberately....131
Keep scope small....132
Be explicit about object references....133
Use arrays and collections for groups of values....133
Optional parameters, named arguments, and `ParamArray`....133
Conditions....134
A decision with more than one rule....134
Choose between several values....135
Loops....135
`For` for a known range....135
`Do` when the end depends on the work....135
`For Each` for a collection....136
Arrays and collections....136
Dates and times....137
Built-in constants....137
Arrays and collections....138
Error handling with `Try` and `Catch`....138
Worked example: choose a message....138
Expected result....139
Check your work....139
Small challenge....139
A safe learning method....139
Tips and tricks....139
Chapter 12 — Work with SQLite data....141
Plan the data first....141
Open a per-user database....141
Use migrations....142
Use parameters....143
Read query results....144
Change several rows together....145
Keep the window responsive....146
Show data in a grid....146

Handle database errors 147
Export rows to CSV 148
Test database code 148
What is not supported 148
The complete WeatherDesk CRUD cycle 149
A safe WeatherDesk workflow 149
Check your work 150
Tips and tricks 150
Chapter 13 — Files, network calls, and reports 153
Work with files safely 153
Read and write files 153
Store per-user data and settings 154
Save a document safely 155
Download a file 156
Call a real public API 156
Make the database part visible 158
Treat the response as untrusted input 158
Export a PDF report 158
Check your work 159
Tips and tricks 160
Chapter 14 — Reuse components and offline packages 161
Two kinds of reusable package 161
What an offline package does 161
Install once, reference from each project 162
Trust and development packages 162
Package preview is not package execution 162
Build the WeatherBadge exercise 163
A safe package exercise 163
Check your work 164
Tips and tricks 164
Chapter 15 — Check, test, and debug 165
Use three different checks 165

Check the project 165
Test the running app 165
Test a release build 166
Use the debugger to answer one question 166
Read the first error first 167
A small debugging loop 167
Test the unhappy path 167
Write automated tests 167
Write a test 168
Unit and UI tests 168
Run the tests 169
Save and recovery 169
Code written with the AI assistant 169
Check your work 170
Tips and tricks 170
Chapter 16 — Work with the AI assistant 171
What the assistant can and cannot do 171
Choose and set up a provider 172
External Codex with your ChatGPT account 173
A Responses API endpoint (for example OpenRouter) 173
LM Studio (local) 174
A provider this version no longer offers 175
A tour of the chat panel 176
Conversations 179
Context chips: what the chat may send 180
Chips during a plan 181
Show exact content 182
Write and send a message 184
Retry and Edit 185
Chat modes 185
When the AI asks you a question 187
Ask questions or ask for changes 188

Cards: how the chat shows AI work....189
Choose how AI changes are applied....190
Plans and stages....192
The plan card....192
What happens after Approve and run....195
Stage cards....196
Read the changes....201
Ask me first....201
Plugins and migrations: dependency cards....204
When Check fails three times....204
When the assistant says a stage is incomplete....207
The stage continues by itself, one missing piece per slice....207
When slicing stops....207
When you change the project during a plan....209
Stop a plan....209
When a plan pauses....210
When the plan is complete....213
Plan an app with /plan....213
Finish an imported project with AI....214
Single changes outside a plan....214
What the chat sends in each mode....216
Add to Chat and the scope....220
The permission card....222
Live activity....223
Undo, Redo, and saving....224
Resume a plan later....225
Errors in the chat....227
Run the result....228
Where chats are stored....228
Privacy....229
Limits....230
The request budget....230

Request size 231
Over budget 231
Other limits 233
Lookups, time, and cost 234
Troubleshooting 235
Codex is not found, or does not start 235
Codex is not signed in, or the model list is empty 235
The plan stopped and shows a status 235
"This chat has no context chips…" 235
"AI provider changed to …" 235
"Provider settings updated. Press Resume again." 235
"The checked change was set aside…" 235
A stage keeps restarting 236
"Project documents changed…" 236
MCB4732 after a stage 236
MCB4733: lookups without finishing 236
"The AI sent nothing for 10 minutes…" 236
"This AI request ran for 60 minutes…" 236
The card's Undo is disabled 236
A request is over budget 237
"Send a message to reconnect this chat, then try again." 237
"Fix works on the diagnostics of the latest Check, Run or Test. Run Check first." 237
"Scope changed: … left this chat's scope. Review the chips, then Send." 237
"Stop asking ends a Plan interview. Switch the chat to Plan, then press it again." 237
The mode cannot be changed 237
Walkthrough: build a small calculator with the assistant 237
Check your work 238
Tips for good requests 239
Check your work 239
Tips and tricks 240
Chapter 17 — Build and publish an app 243

Before you build....243
Test the output, not only the source....243
Development output and customer output....243
Publish a local Mac app from the IDE....244
The Free Edition launch splash....244
Share a project....246
A simple publish checklist....246
Check your work....246
Tips and tricks....247
Chapter 18 — Capstone: build WeatherDesk....249
The user story....249
The two APIs....250
The database schema....250
Build order....250
The API request in BASIC....251
The database write in BASIC....251
The complete CRUD cycle....252
Create....252
Read....252
Update....252
Delete....253
Create a simple WeatherBadge component....253
Step 1: create the component project....253
Step 2: expose a small public contract....254
Step 3: check and package the component....255
Step 4: place the control and pass JSON results to it....255
Step 5: test the component boundary....255
Repository checkpoints....256
What to test....256
Final review....258
Tips and tricks....258
Chapter 19 — Import a Visual Basic 6 project....259

What the importer does 259
What the importer converts in code 261
Before you import 264
Import step by step 264
When the import stops 266
Read the migration report 267
Finish with AI 269
What to finish by hand 269
Check your work 271
Tips and tricks 271
Chapter 20 — Publish an app for Windows 273
What you get 273
Publish step by step 274
What works on Windows 274
What the Check says about Windows 276
The VideoPlayer on Windows 277
Print a report on Windows 278
Try every service: the WindowsServicesTest sample 279
When the publish stops: MCB2302 282
First launch on Windows 282
The launch splash on Windows 282
Publish from the command line 283
Check your work 283
Tips and tricks 283
Chapter 21 — Publish an app for Linux 285
What you get 285
Publish step by step 286
What works on Linux 286
What the Check says about Linux 288
Print a report on Linux 289
Try every service: the WindowsServicesTest sample 289
When the publish stops: MCB2302 290

First launch on Linux....290
The launch splash on Linux....291
Publish from the command line....291
Check your work....291
Tips and tricks....292
Appendix 1 — Quick reference....295
Common workflow....295
IDE command map....295
AI shortcuts....297
Common control names....297
Common event pattern....298
Common constants....299
Common questions....300
Why does Run stop?....300
Why does my code not find a control?....300
Why did my export not replace the old file?....300
Why is a feature marked Limited?....300
Glossary starter....300
Tips and tricks....301
Appendix 2 — Standard component reference....303
How to read this table....303
Form properties....305
Classic properties added in Build 817....307
TextBox....307
Label....308
CommandButton....308
ComboBox and ListBox....308
DataGrid....309
Classic properties added in Build 863....310
Timer....310
Image....310
Panel....311

ImageList....311
Choose by user need....311
The user must enter something....311
The user must choose something....312
The user must start an action....312
The user must understand state....312
The user must understand structure....312
Common properties....312
Event families at a glance....313
Methods and members in code....314
Naming examples....315
Tips and tricks....315
Appendix 3 — McBasic™ language quick reference....317
A form procedure....317
Common declarations....317
Common types....318
User-defined types (Type … End Type)....318
Operators used often....320
Decisions and loops....320
End....321
Strings....321
Dates....321
Format masks....322
Conversions....323
Object references....323
Control item lists....324
Control arrays and the Controls collection....325
Menus and PopupMenu....326
Asynchronous work....327
Procedure signatures....328
JSON values....330
HTTP requests....331

Common app services 331
App 331
Screen 332
Form members in code 333
Built-in constants 333
MsgBox 334
Text 334
Colors 335
Keys 336
Window states 338
Other constants 338
VB6 features and their McBasic™ replacements 339
When a statement does not work 341
Tips and tricks 341
Project files and folder layout 343
The project at a glance 343
How large a project can be 346
Files you edit and files McBasic™ creates 347
Where build output goes 348
AI chats and the project 348
A safe backup 348
Moving a project to another Mac 349
Check the moved copy 349
What not to share 349
Check your work 349
Tips and tricks 350
Diagnostics and troubleshooting 351
When something goes wrong 351
Common problems 352
The project does not open 352
Check reports an unknown name 352
The event does not run 352

The form looks right but the code uses the wrong control 353
A property value is rejected....353
Build works but Run does not 353
Data is missing or appears old....353
The Visual Basic 6 import reports a problem 353
The AI assistant reports a problem 354
Check messages and their fixes....357
A useful minimal test....361
When to ask for help....361
Check your work 362
Tips and tricks....362
Appendix 4 — Check messages by code....363
Glossary 377
A useful distinction....380
Check your understanding....380
Tips and tricks....380
Workshop 1 - Build a small app from an empty form 381
What you will build 381
Before you start 382
Step 1: Design the form 382
Checkpoint: design only 383
Step 2: Add the Preview event 383
Step 3: Add the Clear event....383
Step 4: Test the normal path 384
Step 5: Test the correction paths....384
Step 6: Make the result easier to read....385
After the workshop....385
Check your work 386
Next challenge 386
Tips and tricks....386
Component cookbook 387
Start with the user's job 387

Text input 388
TextBox 388
PasswordBox 388
MaskedTextBox 389
Commands and feedback 389
CommandButton 389
Label 389
Badge 389
StatusBar 390
Choices 390
CheckBox 390
OptionButton 390
ToggleSwitch 390
ComboBox 390
ListBox 391
Search and numeric or time input 391
SearchBox 391
NumericBox 391
DatePicker and TimePicker 391
Display and data 392
DataGrid 392
Image 392
ProgressBar, Gauge, and Sparkline 392
Layout containers 392
Panel and Frame 392
TabControl and TabPage 392
Expander 393
ScrollContainer and SplitContainer 393
Menus and nonvisual helpers 393
Menu, ToolBar, and ContextMenu 393
Timer 393
ValidationProvider 394

Check your component choice 394
The 51-control recipe cards 394
TextBox 394
PasswordBox 395
TabControl 395
TabPage 396
CommandButton 396
Label 396
ValidationProvider 397
Timer 397
Menu 398
CheckBox 398
Panel 399
FlowLayoutPanel 399
Frame 399
Expander 400
OptionButton 400
DataGrid 401
ComboBox 401
ListBox 402
ContextMenu 402
Badge 402
SearchBox 403
ToggleSwitch 403
ColorPicker 404
Image 404
ProgressBar 404
Slider 405
NumericBox 405
DatePicker 406
TreeView 406
StatusBar 406

ToolBar....407
ScrollContainer....407
TimePicker....408
SplitContainer....408
MaskedTextBox....408
Gauge....409
Sparkline....409
ImageList....409
ListView....410
ImageCombo....410
UpDown....411
MonthView....411
DTPicker....412
TabStrip....412
FlatScrollBar....412
Line....413
Shape....413
CommonDialog....413
Animation....414
AudioPlayer....414
VideoPlayer....415
Tips and tricks....415
Workshop 2 - Practice the language with small exercises....417
Exercise 1: make a clean greeting....417
Exercise 2: show a count....417
Exercise 3: keep the normal path short....418
Exercise 4: choose between two messages....418
Exercise 5: use a function for a rule....418
Exercise 6: count a small set of values....419
Exercise 7: choose a branch with Select Case....419
Exercise 8: keep database work behind a message....419
Exercise 9: make a second form honest....420

Exercise 10: write a diagnostic on purpose 420
Exercise 11: make a test table 420
Exercise 12: explain your code in plain English 421
Answer key: what good results look like 421
Final challenge 421
Tips and tricks 421
Workshop 3 - Review a project before delivery 423
Start with the user story 423
Make a feature map 423
Review the form as a first-time user 424
Review the source as a maintainer 424
Test data boundaries 424
Review the published output 425
Prepare a support note 425
Sign-off table 425
Expected result 426
Check your work 426
Next challenge 426
Tips and tricks 426
Practical workflow recipes 427
Recipe 1: validate before a slow operation 427
Recipe 2: show an empty state 427
Recipe 3: keep a selected record visible 428
Recipe 4: confirm a destructive action 428
Recipe 5: keep unsaved input safe 429
Recipe 6: make a search easy to undo 429
Recipe 7: keep a file export honest 429
Recipe 8: report a cancelled operation 430
Recipe 9: pass only what a second form needs 430
Recipe 10: use a timer for display, not hidden work 431
Recipe 11: make accessibility part of the design 431
Recipe 12: make a component replaceable 431

Recipe 13: design the error message first 432
Recipe 14: keep a release note beside the project 432
Recipe 15: turn a bug report into a test 433
Recipe 16: add a feature with the AI assistant 433
A recipe review checklist 434
Test workbook 435
How to use the workbook 435
Test 1: project opens 435
Test 2: first launch 436
Test 3: required input 436
Test 4: normal record 436
Test 5: unusual text 437
Test 6: duplicate or existing data 437
Test 7: search and empty results 438
Test 8: more than one form 438
Test 9: file export 439
Test 10: slow work and cancellation 439
Test 11: keyboard path 439
Test 12: published output 440
Test 13: accessibility review 440
Test 14: privacy review 441
Test 15: release record 441
A failure report template 442
Final test cycle 442
Topic finder 443
I want to... 443
Start and build 443
Work with data and delivery 443
Alphabetical index 444
When you come back to a topic 447

Chapter 1 — Welcome to McBasic™

McBasic™ is a Mac-first desktop development environment. You can design a window, open an event, write BASIC, and run a desktop application without having to learn every detail of how the application is built.

Think of McBasic™ as one project with three connected views:

```
Project files  <->  Form designer  <->  BASIC code
```

The form you draw is part of the project. The controls you name become members that your BASIC code can use. The event procedure you write becomes part of the application. When you change one side, McBasic™ validates the relationships before the next build or run.

Who this book is for

You will get the most from this book if you are one of these readers:

- you used Visual Basic, VBA, or another event-driven BASIC dialect;
- you know basic programming but have not built a desktop GUI before;
- you want to make small business tools, utilities, data-entry applications,

or internal desktop software on macOS;

- you prefer a visible designer and short event procedures to a large setup;
- you want to understand what your application is doing, even when an

assistant helps you write part of it. McBasic™'s optional AI assistant is covered in Chapter 16, "Work with the AI assistant".

You can learn McBasic™ through the IDE without first learning another programming language or a large build process. The book starts with the visible workflow: design a form, write an event, check the project, and run the app.

A pattern you will use often

For many McBasic™ tasks, the steps look like this:

1. **Design:** place a control or select an existing one.
2. **Name:** give it a stable, meaningful name such as `txtName` or

`cmdSave`.

3. **Describe:** set properties such as `Caption`, `Text`, `Width`, or `ReadOnly`.
4. **Behave:** open an event and write the smallest procedure that responds to the user's action.
5. **Check:** ask McBasic™ to validate the complete project.
6. **Run:** launch the application and test the behavior as a user.

The order matters. A button named `cmdSave` is only the beginning. Its Click event still needs a clear, testable job.

Figure 1.1: The McBasic mental model

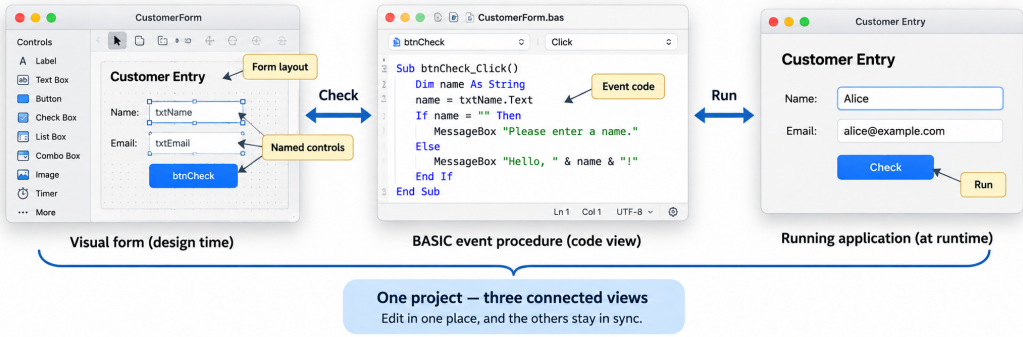


Figure 1.1 - The McBasic mental model

A first look at a real project

The supplied `HelloForms` sample contains two forms. The startup form has:

- a `TextBox` named `txtName`;
- a `CommandButton` named `cmdGreet`;
- a `Label` named `lblGreeting`;
- a second button named `cmdDetails` that opens the details form.

The greeting button is connected to this procedure:

```
Private Sub cmdGreet_Click()
    Dim name As String
    name = Trim$(txtName.Text)

    If Len(name) = 0 Then
        lblGreeting.Caption = "Enter a name."
        Exit Sub
    End If

    lblGreeting.Caption = "Hello, " & name
End Sub
```

Read it as a conversation:

1. When the button is clicked, McBasic™ enters `cmdGreet_Click`.
2. It reads the text from `txtName` and removes surrounding whitespace.
3. If there is no name, it changes the label and stops this procedure.
4. Otherwise, it builds a greeting and writes it to the label.

You will see the same small pattern throughout the book: a user action, some state to read, a decision, and a result to show.

What McBasic™ is not

McBasic™ is not a promise that a program written for another BASIC dialect can be opened unchanged. The language is its own supported dialect, and unsupported constructs should produce a diagnostic rather than being silently reinterpreted.

McBasic™ is also not a web page generator. The target is a desktop application with forms and project files that can be checked and built.

Finally, a proposal or large edit does not replace your review. Check and test the complete project before you trust the result.

Your first checkpoint

Before continuing, make sure you can explain these three statements:

- a control's **Name** is how code refers to that control;
- an **event** is a moment at which code can run;
- **Check** validates the project, while **Run** lets you observe behavior.

If those ideas are clear, you are ready to build the first application.

Expected result

You should now be able to look at a small McBasic™ project and answer three questions:

1. Which control will the user operate?
2. Which event will start the code?
3. Which property or output will show the result?

For the greeting example, the answers are `cmdGreet`, its Click event, and `lblGreeting.Caption`.

Check your work

Open the `HelloForms` sample and point to each part of that chain in the IDE. If you cannot find one part, do not write new code yet. Use Design view to select the control, read its Name, and then open its event in Code view.

Small challenge

Before moving on, predict what will happen if the user enters three spaces instead of a name. Run the sample and compare the result with your prediction.

Tips and tricks

- **Start from a sample instead of a blank form.** On the Welcome window choose **Open a Sample…**. It copies a finished project to a folder you choose, so you can run it, read it and change it without risk.
- **Choose Check before Run, every time.** Check lists any problem in the **Diagnostics** list before the app starts, which is quicker than finding the same problem while the app runs.
- **Treat a project as one item.** In Finder a `.mcbproj` looks like a single file. Copy, zip or back up that one item and you have everything.
- **Use the AI assistant for small steps.** Ask for one clear change, read the result, then ask for the next. Small requests are easier to check and to undo.

Chapter 2 — Install, open, and protect a project

Before the hands-on lessons, take a few minutes to make sure the project is installed, easy to find, and safe to change.

Check the release you are using

Write down the McBasic™ version and build number you are using. Screenshots and menu names can change between builds. If a menu or property differs from the example, check the release notes for that version before continuing.

The Welcome window

When McBasic™ starts, it shows only the **Welcome to McBasic** window. The IDE window, with its toolbox, project tree, properties and editor, appears when you open or create a project; the Welcome window then closes. Choose **Window > Welcome to McBasic** (Shift–Command–1) to show it again at any time.

The window has two layouts:

- **With recent projects**, it has two columns. The left column shows the McBasic™ icon, the name **McBasic** and the version line, then the commands; the right column lists your **Recent Projects** (below).
- **Without recent projects**, for example the first time you start McBasic™, it is a narrow window with one centred column: the McBasic™ icon, the name **McBasic** and the version line at the top, then the **Start** commands, a thin line and the **Learn** commands. The group names are not shown in this layout.

The version line shows the full version and the build, for example **Version 0.4.818 (Build 818)**. The window switches layout by itself: when you remove the last project from **Recent Projects**, it becomes the narrow window, and the next time it appears after you open or create a project, it has two columns again. The window keeps its top edge when it changes size, and you cannot resize it yourself.

The commands are in two groups. Each command has an icon in the accent colour; the keyboard shortcuts are shown at the right of their command, as in a menu.

Start

Command	What it does
New Project… (Command–N)	Opens the same template chooser as **File > New Project…**, including the plugin templates.
Create with AI…	Creates a new blank project and opens the AI chat beside it. If the open project has unsaved changes, McBasic™ first asks whether to save them.
Open a Project… (Command–O)	Opens the same Open panel as **File > Open Project…**.
Open a Sample…	Copies one of this manual's sample projects to a folder you choose and opens the copy.

Learn

Command	Where it goes
User Manual	The documentation on the website, `https://mcbasic.com/docs/`.
Tutorials	The tutorials on the website, `https://mcbasic.com/docs/tutorials/`.
What's New in 0.5	A **What's New in McBasic 0.5** window inside McBasic™ with the release notes of this build.

The bottom of the window holds the checkbox **Show this window when McBasic launches** and four links. With two columns they share one row; in the narrow window the links are centred on their own row under the checkbox.

Link	Where it goes
Website	The McBasic™ website, `https://mcbasic.com/`.
Community	The community page, `https://mcbasic.com/community.html`, with the invite to the McBasic™ Discord server: questions, bug reports, ideas and the apps other developers build.
Support	The contact page, `https://mcbasic.com/contact.html`, for private requests such as privacy or licensing.

Link	Where it goes
Acknowledgements	The **Licenses & Acknowledgements** window inside McBasic. It lists every third-party component with its license and the original license and notice texts, and it repeats the Visual Basic trademark notice. The same window opens from **McBasic > Licenses & Acknowledgements…**.

McBasic™ opens a web page only when you click its link. Nothing is downloaded in the background.

The same pages open from the **Help** menu of the IDE window, so you do not need the Welcome window while a project is open: **Help > User Manual**, **Help > Tutorials**, **Help > McBasic Community**, **Help > Support** and **Help > McBasic Website**.

The checkbox is a setting of this Mac and is on by default. When you turn it off, McBasic™ opens your most recent project at launch instead. If that project no longer exists, the Welcome window appears as usual.

Recent Projects

The right column, **Recent Projects**, lists the projects you opened or created most recently, newest first, up to ten. It is the same list as **File > Open Recent**. The list is kept when you quit McBasic™ and start it again, and it remembers several projects, not only the last one. In the sandboxed edition (the Mac App Store version), McBasic™ also keeps permission to reopen each project through a security-scoped bookmark, so a recent project opens without a new Open panel. Each row shows the project name and the folder that contains it. A project that was moved or deleted, or that is on a disk that is not connected, is shown in grey with **Not found** in place of its folder. In the sandboxed edition, a project outside the folders McBasic™ can see is listed normally; McBasic™ marks it **Not found** only when its permission no longer works. Opening a project that is **Not found** shows the alert **“Name” can’t be found.** with **Remove from Recent Projects** and **Cancel**. Move the project back, or open it again with **Open a Project…**, and it returns to the top of the list.

- Double-click a project, or select it and press Return, to open it.
- Control-click a project for **Show in Finder** (not available for a project marked **Not found**) and **Remove from Recent Projects**. Removing a project takes it off the list, in the Welcome window and in **File > Open Recent**, and the other projects stay listed (including those marked **Not found**). The project itself is not changed. When you open the project again, it comes back to the list. **File > Open Recent > Clear Menu** empties the whole list.
- When the list is empty, the **Recent Projects** column is not shown and the window uses its narrow layout.

While a project is opening, the commands and the list are unavailable. If a project cannot be opened or created, McBasic™ shows an alert with the reason (**The project could not be opened.** or **The project could not be created.**) and the Welcome window stays open.

Use the arrow keys to move through the list. Tab moves between the list, the checkbox and the commands when **Keyboard navigation** is turned on in macOS System Settings > Keyboard. Press Escape to close the Welcome window while a project window is open.

Close a project

Close the project window (its close button, **File > Close** or Command–W) to close the project. If it has unsaved changes, McBasic™ asks whether to save them first. The Welcome window then appears again. To quit McBasic™, choose **McBasic > Quit McBasic** (Command–Q).

Open a project

A McBasic™ project is one item with the extension `.mcbproj`. In Finder it behaves like a single file (a package), but it holds the forms, BASIC source, resources, and package references that belong together. Choose **File > Open Project…** (or **Open a Project…** in the Welcome window) and select the `.mcbproj` item, or pick a recent project from **File > Open Recent** or the Welcome window's **Recent Projects**, which remember the last ten projects you opened or created. Do not open an application folder.

Projects created by older builds were plain folders containing a `.mcproj` file. Those still open. To turn one into a `.mcbproj`, run `mcbasic convert <folder>` once (add `--output <Name>.mcbproj` to choose the name and place). It never overwrites anything and leaves the original folder unchanged.

Create a project

File > New Project… creates one `.mcbproj` in your projects folder. The default is `Documents/McBasic Projects`; change it in **Settings… > Projects** with **Choose…**. McBasic™ proposes a name that is not used yet: `Project1`, `Project2`, and so on (`Library1`, `Library2` for a plugin or control library). You can type another name in the Save panel.

If you choose a name that already exists, macOS asks whether to replace it and McBasic™ then shows its own alert saying where the existing project is. The buttons are **Cancel** (the default), **Move to Trash**, and **Show in Finder**. Replacing never deletes the old project: it goes to the Trash, and you can put it back from there.

In the Welcome window, **Create with AI…** creates a new blank project the same way and opens the AI chat beside it. It always creates a new project: if the open project has unsaved changes, McBasic™ first asks whether to save them, as **File > New Project…** does. The assistant is optional; see Chapter 16, "Work with the AI assistant".

Know the important project files

File or folder	What it contains
`project.mcproj`	project settings (edit them with **Project > Project Settings…**) and the list of forms (inside the `.mcbproj`)
`.bas`	McBasic™ source code
`.form.json`	form layout and control properties
`Resources/`	images and other project resources
`McBasic.lock`	exact offline package inputs, when used
`AI/Activities/`	saved AI chats, only when the project keeps them

A `.mcbproj` contains only files you author. Build output and application bundles are results and are never stored inside it; McBasic™ keeps its working build files in `~/Library/Caches/McBasic/Builds` and removes them automatically. Keep the project and the resources that you authored.

Set the app's name, version and credits

Choose **Project > Project Settings…** to edit the identity of the app that McBasic™ builds. The command is available for a desktop application; a Library has no project settings. The sheet has these fields:

Field	What to type
Bundle Identifier	The reverse-DNS identifier of the built app, for example `com.example.myapp`.
Version	The app version, shown by About and returned by `App.Version`.
Build	The build number, shown by About and returned by `App.Build`.
Copyright	Optional. The copyright line, for example `© 2026 Your Name`.

Field	What to type
Company	Optional. The name of your company; `App.CompanyName` returns it.

McBasic™ checks the fields while you type and keeps **OK** disabled, with the reason in red under the fields, until every field is valid:

- **Bundle Identifier** must have at most 200 characters, only letters A-Z, digits, "." and "-", and at least one ".". The message is "Bundle Identifier must have at most 200 characters, only letters A-Z, digits, "." and "-", and at least one "." (for example com.example.myapp)."
- **Version** needs exactly three numbers separated by dots, digits only. The message is "Version needs exactly three numbers separated by dots, digits only (for example 1.0.0)."
- **Build** must be a whole number, digits only, up to 4294967295. The message is "Build must be a whole number, digits only, up to 4294967295 (for example 1)."
- **Copyright** and **Company** may have at most 200 characters. A longer text shows "Copyright has N characters; the limit is 200. Shorten it, then press OK." (or the same for **Company**).
- Control characters, such as a tab or a line break, are not allowed. McBasic™ removes them as you type and tells you so, and it trims the spaces around every field.

Press **OK** to apply the settings, or **Cancel** to close the sheet without changing anything. The change is one step of **Edit > Undo Project Settings**, and it is saved with the project the next time you save: the status line says "Project settings changed. Save to keep them." The settings are stored in the `application` block of `project.mcproj`; **Copyright** and **Company** are the optional keys `copyright` and `company`, and McBasic™ leaves them out of the file when they are empty. Projects made before Build 865 have neither key and stay valid.

Every app made with McBasic™ shows who made it. **About** in the built app's application menu always shows the McBasic™ About box: the app name, "Version x (Build n)", your copyright line when you set one, and "Made with McBasic™". The **Copyright** also goes into the app's `Info.plist`, so Finder shows it in Get Info. See Chapter 8, "The Menu Editor", for how a form's own `mnuAbout` item relates to this box.

Choose the appearance

The **Appearance:** pop-up in the same sheet has two items, **macOS** and **Classic**. It chooses how the controls of the app look. The help text under it reads: "Classic draws controls like a Windows app of the Visual Basic 6 era; imported projects start Classic."

Item	The look
macOS	The default. Rounded, light or dark controls that follow the Mac.
Classic	Grey faces with 3D bevels, sunken edges on text boxes, lists and combo boxes, classic check and option glyphs, etched frames and the font Microsoft Sans Serif at size 11.

Only the look changes. Your code, the layout of every form and every event stay the same, so you can switch back and forth at any time.

- **New projects and projects made by the AI assistant start as macOS.** A project that you imported with **File ▸ Import Visual Basic 6 Project…** in Build 868 or later starts as **Classic**. Projects made before Build 868 stay macOS until you choose **Classic**.
- **Classic is always light.** It does not follow the Mac's dark mode, as Visual Basic 6 had no dark mode. Its palette is the Windows 10 system colours: a light grey face (`#F0F0F0`), white fields, black text and a blue selection (`#0078D7`).
- **The font.** The default font in Classic is Microsoft Sans Serif at size 11. A font whose family is `System` means this font in Classic. If your Mac does not have Microsoft Sans Serif, McBasic™ uses the normal theme font.
- **Your own choices win.** A `BackColor`, `ForeColor` or `Font` that you set on a form or a control is used as you set it, in both looks.

Classic draws these controls:

- CommandButton, TextBox, ListBox, ComboBox, CheckBox, OptionButton and Frame, and the border of a Label or an Image whose `BorderStyle` is Solid;
- scroll bars: the FlatScrollBar and every scroll bar inside a control (ListBox, multi-line TextBox, the list of a ComboBox, TreeView, ListView, DataGrid and ScrollContainer). They are 17 DIP thick, with grey arrow buttons, a dotted track and a raised thumb;
- TabControl, TabPage and TabStrip, with raised tabs that have cut (chamfered) corners. The selected tab is a little taller and joined to the page. The tabs are laid out as described in "Tabs in Classic" below;
- ProgressBar (blue blocks in a sunken well), Slider (a pointed thumb and ticks) and UpDown (a sunken field with two small buttons);
- StatusBar, ToolBar, TreeView, ListView and DataGrid, with sunken edges and raised column headers;
- DTPicker and MonthView, as described in "DTPicker and MonthView in Classic" below;

- ImageCombo and MaskedTextBox, which look like a ComboBox and a TextBox. The closed box of an ImageCombo shows the image and the text of the selected item, left-aligned. The items take their images from the ImageList named in the control's `ImageList` property.

Controls that exist only in McBasic™ (for example SearchBox, NumericBox and DatePicker) keep the macOS look. They do show the Classic scroll bars when they have one. The Form designer shows the same Classic look as the running app. A vertical Slider has its **Minimum** at the top in Classic, as in Visual Basic 6.

Tabs in Classic

Every tab of a TabControl in Classic has cut (chamfered) corners at the top. The selected tab is a little taller than the others and is joined to the page, with no line between them.

A TabControl in Classic shows its tabs in one of two ways, chosen by its `TabsPerRow` property (it changes only the Classic look; the macOS look ignores it):

- **`TabsPerRow` 0, the default.** Each tab is as wide as its caption, and all the tabs stand on one row, like the tabs of a Windows dialog box. A TabControl that you add in the designer starts this way.
- **`TabsPerRow` 1 or more: the SSTab layout.** The tabs are laid out like the SSTab control of Visual Basic 6. Each row holds `TabsPerRow` tabs, each an equal share of the control's width whatever its caption, so with `TabsPerRow` 3 and two pages the last third of the row stays empty, as in an SSTab. Further pages go in further rows; the first tabs are in the row farthest from the page. Each extra row is one tab taller and moves the page down by one tab. Clicking a tab in a back row selects its page and raises the tab, but the rows stay in place.
- **Imported SSTabs use the SSTab layout.** The importer copies the SSTab's `TabsPerRow` (3 when the form file leaves it out), so the rows match the Visual Basic 6 form.

A TabStrip in Classic follows the Windows tab control:

- **Each tab is as wide as its caption.** With `MultiRow` False all the tabs stay on one row.
- **With `MultiRow` True** the tabs wrap into as many rows as they need, and each row is stretched to the full width of the control.
- **The selected row comes forward.** When you click a tab in a back row, the rows rotate so that the row with the selected tab stands on the client area, and the row after it comes next. With `Alignment` Bottom the rows are under the client area and the selected row is the top one.

The Form designer draws the same rows. A TabControl shows the captions of its pages there.

DTPicker and MonthView in Classic

- **The time of a DTPicker is text with fields.** The hour, the minute, the second (when `UseSeconds` is True) and AM/PM (when `Clock` is 12Hour) are shown as "9:05:07 AM" or "09:05:07". Click a field to select it; it turns blue while the DTPicker has the focus (a spin button gives it the focus too). **Left Arrow** and **Right Arrow** move to the next field, **Home** and **End** to the first and last. **Up Arrow** and **Down Arrow** change the selected field by one (the minutes by `MinuteIncrement`), and wrap inside it: 59 minutes go to 0 without changing the hour. Type digits to enter a value, and A or P to choose AM or PM.
- **A DTPicker whose `Mode` is Time has two spin buttons.** They stand at the right, where a date picker has its drop-down button, as in a Visual Basic 6 DTPicker whose format is time. The upper button raises the selected field and the lower one lowers it; hold a button to repeat.
- **The value stays inside its limits.** A field never changes to a time outside `MinimumDate` and `MaximumDate`; the change is simply not made. Every change raises `Change`, as before.
- **A MonthView circles today.** Today's date has a red rounded frame, as in Visual Basic 6. The frame is always there.
- **`ShowToday` shows the "Today:" line.** With `ShowToday` True (the default) the bottom of the MonthView shows a small red circle, then "Today:" and today's date in bold. Click it to show today's month and select today. With `ShowToday` False the line is hidden; today keeps its red frame. The month that drops down from a DTPicker always shows the line. When a MonthView is too short for the line, the line is hidden so that every week stays visible.
- **The month that drops down is opaque.** The drop-down month of a DTPicker sits on a white field inside a black frame 1 pixel thick, as in Visual Basic 6.

Disabled controls in Classic

A control whose `Enabled` is False keeps its Classic edges and shows its text in grey (GrayText, `#6D6D6D`), as in Visual Basic 6:

- a TextBox, ComboBox, ListBox and UpDown keep their white field;
- a TreeView, ListView and DTPicker turn light grey (ButtonFace, `#F0F0F0`), as the Windows controls do; a DataGrid keeps its white cells;
- a disabled TreeView, ListView or DataGrid shows no blue selection, because the selection is shown only while the control can have the focus;
- a CommandButton keeps its bevel and greys its caption.

An authored `BackColor` stays as you set it.

These limits apply in Build 875:

- The list that drops down from a ComboBox keeps the macOS rows inside a thin black frame.
- Some Classic controls differ in small details from Visual Basic 6. A ToolBar does not grey a single disabled button. The row header of a DataGrid has a grey face but no bevel. A DTPicker whose Mode is DateTime shows the date with its drop-down button, then the time.
- A TabControl has no `TabMaxWidth` or `Style` property: SSTab tabs always share the row equally, and the selected tab's caption is not bold. Its tab rows are about 24 DIP tall whatever the SSTab's `TabHeight` was, so the controls of an SSTab with several rows sit a little higher than in Visual Basic 6.
- The fixed SSTab rows, the bold "Today:" text and the disabled look of the DataGrid follow Visual Basic 6 from memory; they have not yet been compared on screen.
- The title of a MonthView is grey (ButtonFace), with black text.
- A few properties read different values in Classic when you did not set them: `BorderWidth` of a TreeView and a ListView is 2, and `RowHeight` of a DataGrid is 18. The Inspector keeps showing the usual defaults.
- The component preview in the IDE uses the Classic styles for a Classic project. It has not been checked on screen yet.

System colors

A color property accepts three kinds of value: `#RRGGBB`, an empty text, or the name of a system color. This holds for `BackColor`, `ForeColor`, `BorderColor`, `AlternatingRowBackColor`, `FillColor` and the other color properties of a form or a control, in the layout, in the Inspector and in code. The 25 names are the Visual Basic 6 system color constants without the `vb`. They are not case sensitive, and McBasic™ writes them with the capitals shown here:

`ScrollBars`, `Desktop`, `ActiveTitleBar`, `InactiveTitleBar`, `MenuBar`, `WindowBackground`, `WindowFrame`, `MenuText`, `WindowText`, `TitleBarText`, `ActiveBorder`, `InactiveBorder`, `ApplicationWorkspace`, `Highlight`, `HighlightText`, `ButtonFace`, `ButtonShadow`, `GrayText`, `ButtonText`, `InactiveCaptionText`, `3DHighlight`, `3DDKShadow`, `3DLight`, `InfoText` and `InfoBackground`.

A name stands for a different color in each appearance. Classic uses the Windows 10 values that a Visual Basic 6 program shows today. The macOS look uses the same values, except for the text colors and `Highlight`:

Name	Classic	macOS
`MenuText`, `WindowText`, `TitleBarText`, `ButtonText`, `InfoText`	`#000000`	`#1C1C1C`

Name	Classic	macOS
`Highlight`	`#0078D7`	`#007AFF`
`ButtonFace`, `MenuBar`	`#F0F0F0`	`#F0F0F0`
`WindowBackground`, `HighlightText`, `3DHighlight`	`#FFFFFF`	`#FFFFFF`
`ButtonShadow`	`#A0A0A0`	`#A0A0A0`
`GrayText`	`#6D6D6D`	`#6D6D6D`
`3DLight`	`#E3E3E3`	`#E3E3E3`
`3DDKShadow`	`#696969`	`#696969`
`WindowFrame`	`#646464`	`#646464`
`ScrollBars`	`#C8C8C8`	`#C8C8C8`
`Desktop`, `InactiveCaptionText`	`#000000`	`#000000`
`ActiveTitleBar`	`#99B4D1`	`#99B4D1`
`InactiveTitleBar`	`#BFCDDB`	`#BFCDDB`
`ActiveBorder`	`#B4B4B4`	`#B4B4B4`
`InactiveBorder`	`#F4F7FC`	`#F4F7FC`
`ApplicationWorkspace`	`#ABABAB`	`#ABABAB`
`InfoBackground`	`#FFFFE1`	`#FFFFE1`

McBasic™ apps are always light, so there is no separate dark value. The name is stored as it is, so if you switch the **Appearance:** of the project, every color given by a name changes with it. Reading the property in code returns the name, not the `#RRGGBB` value. In code you can use the constants `vbButtonFace`, `vbWindowText`, `vbHighlight` and so on (and `vb3DFace` for `ButtonFace`, `vb3DShadow` for `ButtonShadow`): each one holds the name as text, so `cmdOk.BackColor = vbButtonFace` works. Text that is not `#RRGGBB`, a name or empty is an error: "MCB2101: BackColor takes "#RRGGBB" text, a system color name such as "ButtonFace", or "" for the theme default, not "Purple"; use one of these." The `CommonDialog` color and the `ColorPicker` value keep `#RRGGBB`, because they hold a color the user picked.

Press **OK** to apply the choice. It is one step of **Edit > Undo Project Settings**, like the other fields. The setting is stored in the `application` block of `project.mcproj` as `appearance` (see Chapter 22).

Choose the app icon

The **Icon:** row of the same sheet sets the icon of the app that McBasic™ builds: the icon in the Dock, in Finder, in the Windows taskbar and on the Windows `.exe`. It shows a 64-point preview of the icon and two buttons:

Control	What it does
Preview	The current icon. It shows the McBasic™ icon while the project uses the default.
Choose…	Opens a file panel, **Choose App Icon**, that accepts a PNG, ICNS or ICO image.
Use Default	Goes back to the McBasic™ icon. It is dimmed while the default is in use.

The help text under the row reads: "A square PNG (1024 × 1024 recommended, at least 256 × 256) gives the Mac and the Windows app their icon. An ICNS is used only on the Mac, an ICO only on Windows."

- **A square PNG is the best choice.** When you build, McBasic™ makes the Mac icon (`.icns`, 16 to 1024 pixels) and the Windows icon (`.ico`, 16 to 256 pixels) from it, the same way on every Mac and without a network. Use 1024 × 1024 pixels with a transparent background for the sharpest result.
- **An ICNS or ICO file is used as it is,** but only on its own system: an ICNS gives the Mac app its icon and the Windows app keeps the McBasic™ icon, and an ICO does the opposite. Publishing then shows the warning "MCB2407: the app icon AppIcon.icns is a macOS .icns file, so the Windows app shows the McBasic icon; choose a square PNG in Project > Project Settings to use one icon on both." (or the same for an ICO on the Mac). The app is still published.
- **The sheet checks the image when you choose it.** A wrong image is refused in red under the fields and the icon does not change. The messages are "The image is W × H pixels. An app icon must be square; crop it (1024 × 1024 recommended).", "The image is W × H pixels. Use a square PNG of at least 256 × 256 (1024 × 1024 recommended).", "The image is W × H pixels, larger than 4096 × 4096. Scale it down to 1024 × 1024.", "The image is an interlaced PNG. Save it again without interlacing.", "The image is larger than 8 MB. Use a smaller one; a 1024 × 1024 PNG is enough." and "This file is not a valid PNG image. Export it again as PNG or choose another image." (or "valid ICNS icon" and "valid ICO icon" for the other two formats).

Press **OK** to apply the choice. McBasic™ copies the image into the project as `AppIcon.png`, `AppIcon.icns` or `AppIcon.ico` (it replaces a file of that name) and writes its name into the `application` block of `project.mcproj` as `icon`. The copy and the setting are one step of **Edit > Undo Project Settings**, and both are written when you save; if the save fails, the project stays unsaved. You do not have

to save before you ask the AI chat for a change: the chat's Check sees the new icon as the IDE shows it, and nothing is written until you save. **Use Default** removes the `icon` key; the image file stays in the project, so you can delete it in Finder if you no longer want it. The icon is free in every edition: the launch splash and the About box still show "Made with McBasic™".

Make a working copy

The quickest way to start from one of this manual's samples is **Open a Sample…** in the Welcome window. Choose the sample in the **Open a Sample** dialog and click **Choose Location…**. In the **Save a Copy of the Sample** panel, type a name and choose a folder (your projects folder is proposed). McBasic™ copies the sample there and opens the copy. The sample inside McBasic™ is never opened or changed, and McBasic™ never replaces an existing project with a sample: if the name already exists, the Save panel says so and stays open so that you can choose another name. Each copy gets its own project identity, so two copies of the same sample never share AI chats.

To copy a project by hand before following a tutorial:

1. copy the sample `.mcbproj` to a folder you control;
2. give the copy a clear name;
3. open the copied `.mcbproj`;
4. run the unchanged copy once;
5. start the exercise.

This gives you a known-good starting point.

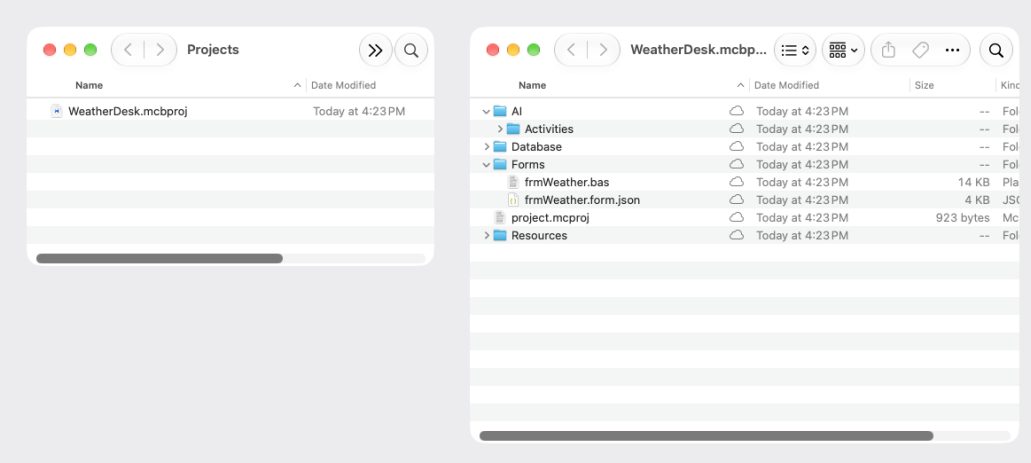


Figure 2.1: Project files

Check your work

- McBasic™ opens the copied project.
- The unchanged sample passes Check.
- The sample runs before you edit it.
- You know where the source and layout files are stored.

Expected result

You have a safe, working copy of the sample. The original is still available if you need to start over, and you know which file McBasic™ opens first.

Small challenge

Make a second copy with a different name. Open it without changing either copy. Copying is easy because the whole project is a single `.mcbproj`. This is a simple way to practise the backup routine before the project becomes important.

Tips and tricks

- **Check the version before you test.** After you rebuild or update McBasic™, look at the version line on the Welcome window or open **McBasic > About McBasic**. Testing an older build by mistake is the most common reason a fix seems not to work.
- **Look in Documents/McBasic Projects first.** New projects are saved there by default. To use another folder, change it in **Settings… > Projects** with **Choose…**.
- **Open the .mcbproj, not a folder inside it.** A project is a package that Finder shows as one item. Use **File > Open Project…** and pick that item, or choose it from **Recent Projects**.
- **Make a working copy before a big experiment.** Copy the `.mcbproj` in Finder and experiment on the copy. If the experiment goes wrong, the original is untouched.
- **Set the version and credits once, early.** Open **Project > Project Settings…** and fill in **Version**, **Build**, **Copyright** and **Company** before you publish; the built app's About box and Get Info show them.
- **Give the app its own icon early.** Choose a 1024 × 1024 PNG with **Icon: Choose…** in **Project > Project Settings…**. One PNG serves both the Mac and the Windows app, and every Run already shows it in the Dock.
- **Try Classic before you decide.** Open **Project > Project Settings…**, choose **Classic** in **Appearance:** and press **OK**. Run the app, then switch back if you prefer the macOS look. Your code does not change.

- **Want SSTab tabs?** Set `TabsPerRow` of a TabControl to 3 (or as many tabs as you want on a row). With more pages than that, leave room for a second row of tabs: the page starts one tab lower.
- **Step a time with the keyboard.** In a Classic DTPicker, click the minutes and press **Up Arrow** or **Down Arrow**; the hour does not change when the minutes wrap.
- **Use a system color name to follow the look.** Set `BackColor` to `ButtonFace` instead of `#F0F0F0`. The control then follows the project's **Appearance:** and your choice stays one name.
- **Keep projects where you can find them.** A project that you move or delete shows **Not found** in **Recent Projects**. Open it again from its new place and it returns to the list.

Chapter 3 — Build your first working app

Start with the supplied `HelloForms` sample. You will run it, change one line of code, and open a second window. Along the way, you will use this routine:

```
open -> look -> change -> check -> run -> test
```

Before you start

You need:

- a Mac with McBasic™ installed;
- the `HelloForms` sample that comes with McBasic™;
- a copy of the sample that you can edit.

Make a copy first. This lets you return to the original sample if you make a mistake. McBasic™ makes the copy for you in the next step.

Step 1: Open the sample

1. Start McBasic. The **Welcome to McBasic** window appears.
2. Click **Open a Sample…**, choose `HelloForms` and click **Choose Location…**. Type a name for your copy, choose a folder and click **Save**. McBasic™ opens the copy. (If you already have a copy, click **Open a Project…** or choose **File > Open Project…** and open your copy of `HelloForms.mcbproj` instead.)
3. Select `frmMain`, the first form, if it is not already selected.
4. Choose **Design**.

You should see a name box, a **Greet** button, a message label, and an **Open details** button. Click each control once. Notice its name in the Properties area. The name is how your code finds the control.

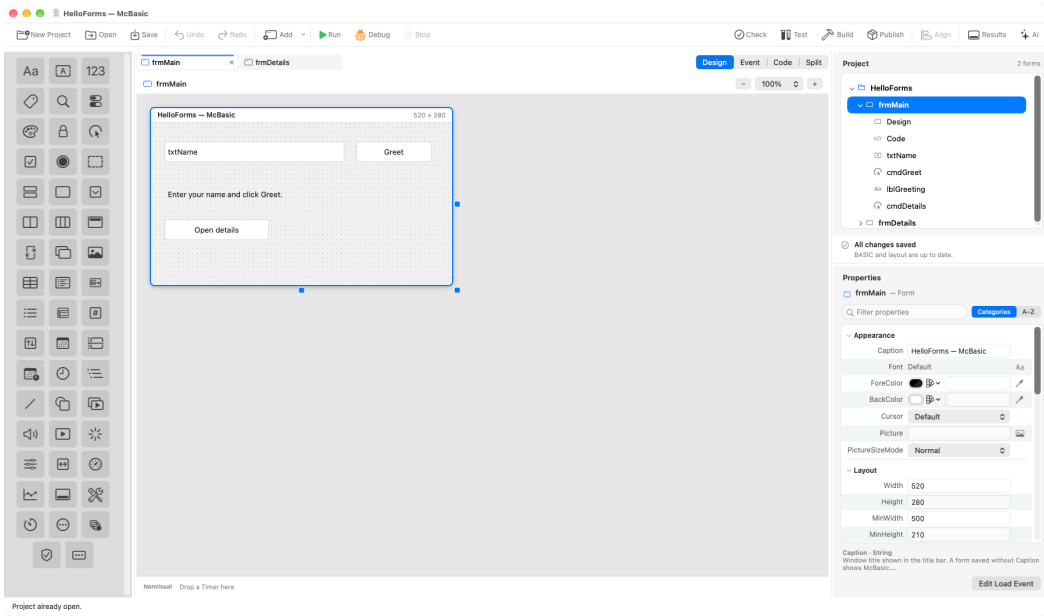


Figure 3.1: The HelloForms sample

Step 2: Run the sample

Choose **Run**. When the app opens, click **Greet** without typing a name.

You should see:

```
Enter a name.
```

Now type `Ada` in the name box and click **Greet** again. You should see:

```
Hello, Ada
```

This shows that the app is running the BASIC code connected to the button.

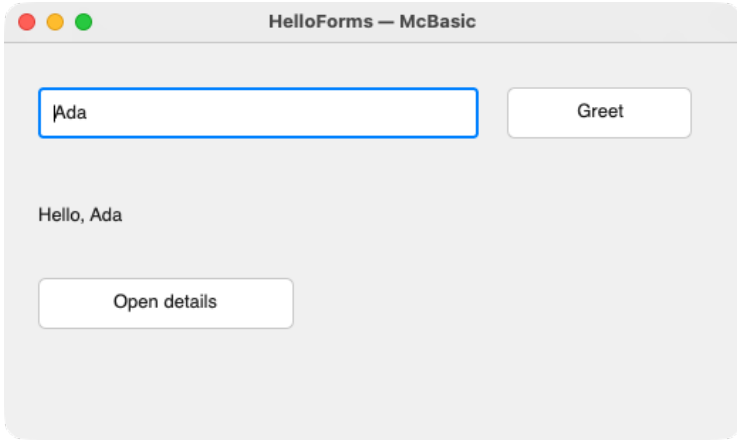


Figure 3.2: A successful greeting

Close the running app before you edit the project.

Step 3: Read the button code

Choose **Code** and open the `cmdGreet_Click` procedure. It looks like this:

```
Private Sub cmdGreet_Click()
    Dim name As String
    name = Trim$(txtName.Text)

    If Len(name) = 0 Then
        lblGreeting.Caption = "Enter a name."
        Exit Sub
    End If

    lblGreeting.Caption = "Hello, " & name
End Sub
```

Read it from top to bottom:

1. The code starts when the user clicks `cmdGreet`.
2. It reads the text from `txtName`.
3. It removes spaces at the start and end of the name.
4. If the name is empty, it changes the label and stops.
5. Otherwise, it builds a greeting and puts it in the label.

Step 4: Make a small change

Change the last assignment to add an exclamation mark:

```
lblGreeting.Caption = "Hello, " & name & "!"
```

Save the project. Then choose **Check**.

Check looks for problems in the project before McBasic™ tries to run it. If it reports an error, read the message and fix that error first.

Choose **Run**, enter `Ada`, and click **Greet**. The message should now be:

```
Hello, Ada!
```

Step 5: Open the second window

Close the running app. In Code view, read the `cmdDetails_Click` procedure:

```
Private Sub cmdDetails_Click()
    Dim details As frmDetails
    Set details = New frmDetails
    details.DisplayName = txtName.Text
    details.Show
End Sub
```

Run the app again, enter a name, and choose **Open details**.

The second window opens and shows the name. `New frmDetails` creates a new window. `DisplayName` gives it the text from the first window. `Show` makes the new window visible.

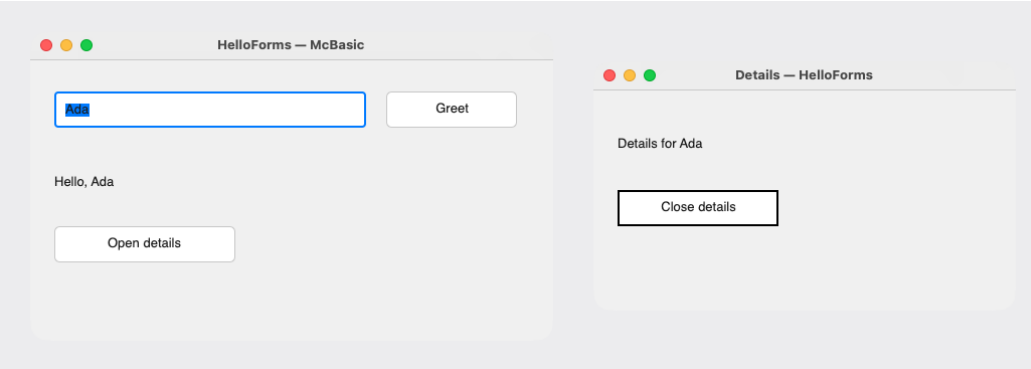


Figure 3.3: Two windows

Check your work

You are ready for the next chapter when:

- the sample opens and runs;
- an empty name shows `Enter a name.`;
- a name shows a greeting;
- your exclamation-mark change appears after a new run;
- the details window opens with the entered name;
- Check finishes without an error.

Small challenge

Change the button's Caption from `Greet` to `Say hello`, then run the app.

The text on the button changes, but the code name `cmdGreet` does not. This is an important difference:

- **Caption** is what the user sees;
- **Name** is what your code uses.

The project that grows with the lessons

From Chapter 5 onward, the larger example uses one project called **WeatherDesk**. It starts as a form, then gains a local SQLite database, an Open-Meteo geocoding request, a forecast request, JSON parsing, and CSV export. Each new chapter changes the same project. The working files are in the repository as `samples/WeatherDesk.mcbproj`.

Tips and tricks

- **Press Stop to end your app.** Choose **Stop** in the toolbar (Command–period) when you have finished trying the sample.
- **Change one thing, then Check and Run.** One small change at a time shows you exactly which edit caused a new result.
- **Undo is always there.** If a change goes wrong, use **Edit > Undo** to step back instead of retyping the old code.
- **Open the second window from the code you just read.** Following the button code to the second form is the quickest way to see how forms and code belong together.

Chapter 4 — Find your way around the IDE

The IDE is where you design forms, write code, and check your project. There is no need to learn every panel on the first day, so begin with the areas you will use most often.

McBasic™ uses one workspace for the whole job. You do not have to copy code into another program to move from design to source. The same project stays open while you switch between its form layout, BASIC code, diagnostics, and running result.

The Welcome window and the IDE window

McBasic™ starts with the **Welcome to McBasic** window (Chapter 2, "The Welcome window"). The IDE window described in this chapter appears when you open or create a project, from the Welcome window or from the **File** menu, and the Welcome window then closes.

- Close the IDE window (its close button, **File > Close** or Command–W) to close the project. McBasic™ asks to save unsaved changes, then shows the Welcome window again.
- Choose **Window > Welcome to McBasic** (Shift–Command–1) to show the Welcome window while a project is open, for example to open a recent project or a sample. Press Escape to close it again.
- Choose **McBasic > Quit McBasic** (Command–Q) to quit.

The main areas

Area	What it is for
Project tree	Find forms, modules, and other project files. Right-click (or Control-click) a form, module, class or test and choose **Add to Chat** to let the AI assistant change that whole document. A form that has a menu shows a **Menu** group with the menu items nested; double-click an item to open its `Click` handler, or the group to open the Menu Editor (Chapter 8).

Area	What it is for
Document tabs	One tab per open form or code file. Click a tab to show it; the **Design | Event | Code | Split** control at the right of the tabs chooses the view.
Design view	Place and arrange controls.
Code view	Write McBasic™ code.
Component rail	Choose a control to add to a form.
Canvas	The visible area where a form is designed.
Properties	Change the selected form or control.
Results	Read Check messages and errors.
AI	Ask the optional AI assistant about the open project.

The toolbar

The toolbar sits at the top of the project window. It comes in three styles with the same commands: **Ribbon** (the default, see The Ribbon), **Mac** and **Classic** (see Toolbar settings). In the **Mac** style the toolbar sits next to the window title and every command shows its icon with its name underneath. From left to right:

Command	Icon	What it does
Run	green play triangle	Builds and runs the current project (Command–R). In a Custom Control Plugin it reads **Run Test App**.
Debug	orange ladybug	Builds and runs the project with the debugger.
Stop	red square	Stops the running build, application or debugger (Command–.).
New Project	folder with a plus	Creates a new `.mcbproj` in your projects folder from a template (Command–N).
Open	folder with an arrow	Opens an existing project (Command–O). **File > Open Recent** lists the last ten.

Command	Icon	What it does
Save	arrow into a tray	Writes the current project changes (Command–S).
Undo / Redo	curved arrows	Move through recent editor and designer changes.
Add	window with a plus	Adds a Form (a Control in a Custom Control Plugin; a Module in a Library Plugin). The small arrow, or a long press, opens a menu: **Form**, **Module**, **Class**, **Existing File…**.
Align	left-aligned bars	Aligns, sizes or spaces the selected controls. It appears only while the form designer is showing.
Check	tick in a circle	Checks the BASIC code (Command–K).
Test	test tubes	Runs the project's unit tests (Command–U).
Build	hammer	Compiles the project (Command–B). In a plugin it reads **Build Package**.
Publish	box	Creates a standalone app on this Mac.
Certify	rosette	Runs the SDK certification. It appears only in plugin projects.
Results	window with a filled bottom band	Shows or hides the Results pane (Shift–Command–Y).
AI	sparkles	Shows or hides the AI assistant on the right (Shift–Command–L).

Only Run, Debug and Stop have coloured icons; the other icons follow light and dark mode. The icons of Results and AI are tinted with your accent colour while their pane is open. A command that cannot run right now is dimmed. Hover over any button to read its tooltip.

In the **Mac** style the commands sit in groups (on macOS 26 and later, each group shares one rounded background): **Run**, **Debug**, **Stop** | **New Project**, **Open**, **Save** | **Undo**, **Redo** | **Add**, **Align** | **Check**, **Test**, **Build**, **Publish**, **Certify** | **Results**, **AI**. You can rearrange the groups: Control-click the toolbar and choose **Customize Toolbar…**.

Toolbar settings

Choose **McBasic > Settings…**. The **Appearance** section has three toolbar settings:

- **Toolbar**: **Ribbon** (the default) shows tabs of grouped commands (see The Ribbon). **Mac** is the native toolbar described above. **Classic** shows one line of buttons inside the window (see The Classic toolbar).
- **Icon size**: **Small**, **Standard** or **Large** (the default). It applies to every style.
- **Show labels**: on by default. Turn it off to show the icons alone. It applies to every style.

Choose **Apply**; the toolbar changes at once, without restarting McBasic™. The settings belong to your Mac, not to a project.

The Classic toolbar

Choose **Toolbar: Classic** for one line of buttons at the top of the window, in the style of classic desktop development tools. Each button shows its icon with its name beside it; its rounded outline appears only while the pointer is over it. The line has two halves, with empty space between them:

- **Left half: New Project**, **Open**, **Save** | **Undo**, **Redo** | **Add** | **Run**, **Debug**, **Stop**
- **Right half**, against the right edge of the window: **Check**, **Test**, **Build**, **Publish**, **Certify** | **Align** | **Results**, **AI**

When the window is too narrow for both halves side by side, the right half moves to a second line, still against the right edge. This happens as you resize the window, and goes back to one line when there is room again. With every command showing (a plugin project with the form designer open), the labelled buttons need a window about 1560 points wide at **Large**, 1500 at **Standard** and 1430 at **Small**; in a desktop app, which has no **Certify**, **Standard** fits one line from about 1320 points and **Large** from about 1370. Without labels one line fits even the narrowest window.

The buttons work exactly like the Mac toolbar buttons, with the same icons, tooltips and rules:

- **Certify** appears only in plugin projects;
- **Align** appears only while the form designer is showing;
- a command that cannot run right now is dimmed.

The window title stays above the buttons. **Icon size** and **Show labels** apply here too. The Classic toolbar cannot be customised.

To work the toolbar from the keyboard, choose **View > Focus Ribbon** (Command–7). The left and right arrow keys move along a line, and, when the toolbar has wrapped onto two lines, the up and down arrow keys move between the lines. Press Space to run the button.

The Ribbon

Choose **Toolbar: Ribbon** for a tabbed toolbar at the top of the window. Click a tab to show its commands. Each tab arranges its commands in labelled groups. The main commands are large buttons with the name under the icon; the others are small buttons with the name beside it.

Tab	Groups and commands
Home	**Run**: Run, Debug, Stop · **Project**: New Project, Open, Save · **Edit**: Undo, Redo · **Insert**: Add · **Application**: Check, Build, Test, Publish · **Panels**: Results, AI
Design	**Insert**: Add, Form, Module, Class, Existing File… · **Arrange**: Align (only while the form designer is showing) · **Form**: Grid Settings · **Menu**: Menu Editor
Run	**Run**: Run, Debug, Stop · **Debugger**: Continue, Step Over (F10), Step Into (F11), Step Out (F12)
Publish	**Publish**: Build, Publish, Show App, Certify (plugin projects only), Plugin References, Package Inputs
View	**Panels**: Results, AI

Some commands appear on more than one tab, so the ones you use most are always one click away. The Ribbon also offers some commands that the Mac and Classic toolbars leave to the menus:

Command	Icon	Same as
Continue	triangle against a bar	**Project > Debug > Continue**
Step Over	arrow bouncing over a line	**Project > Debug > Step Over** (F10)
Step Into	arrow down to a line	**Project > Debug > Step Into** (F11)
Step Out	arrow up to a line	**Project > Debug > Step Out** (F12)
Show App	arrow out of an app icon	**Project > Show Published App** (in a plugin, **Show Built Package**)

Command	Icon	Same as
Plugin References	puzzle piece	**Project > Plugin References…**
Package Inputs	full tray	**Project > Offline Package Inputs…**
Grid Settings	grid	**Design > Grid Settings…**
Menu Editor	menu bar	**Design > Menu Editor…** (Command–E; unavailable in a UserControl designer)

The step commands work while a debug session is running. **Icon size** changes the small buttons; the large buttons keep their size, except at **Small**, where they shrink too. With **Show labels** off, every button shows its icon alone and the group names stay. The Ribbon cannot be customised.

The Ribbon opens on **Home** each time McBasic™ starts. It keeps the tab you chose only while the style stays Ribbon. If you switch to another style and back, the Ribbon returns to **Home**.

To work the Ribbon from the keyboard, choose **View > Focus Ribbon** (Command–7). The left and right arrow keys move along the tabs, and Space opens a tab. The down arrow key moves into the commands, and the up arrow key moves back to the tabs.

Adding documents to a project

Use **Add** in the toolbar, or the **Project** menu:

- **Add Form…** adds a window with its own code and layout.
- **Add Module…** adds a standard BASIC module.
- **Add Class…** adds a class module. McBasic™ asks for a name, such as `Customer`; **Save** writes `Classes/Customer.cls` and its project entry.
- **Add Existing File…** (Command–D) copies a `.bas` module, a `.cls` class, or a form's `.bas` and `.form.json` files into the project. For a form, select both files (Command-click); in the sandboxed edition both must be selected. The files are copied when you choose **Save**; the originals are never changed. If the project already has a document with the same name, nothing is added and nothing is overwritten: rename the file and try again.

Knowing that your work is saved

While the project has unsaved changes, the window title reads **HelloForms — Edited — McBasic** (with your project's name), the close button shows a dot, and the save state under the project files reads **Unsaved changes** with the line *Save Project to write BASIC and layout changes.*

When you choose **Save** (the toolbar button or Command–S) and the save succeeds:

- the **Save** button shows a green checkmark for about a second and a half, then its usual icon returns (with **Reduce Motion** turned on in System Settings the icon changes without fading; VoiceOver says “Saved”);
- the save state reads **Saved at** followed by the time, in your Mac's short time format (for example **Saved at 14:32**), with the line *BASIC and layout are up to date.* It keeps that text until your next edit;
- the title loses **— Edited** and goes back to **HelloForms — McBasic**.

If the save fails, none of this happens: McBasic™ shows an alert that explains the problem, the title keeps **— Edited**, and the save state stays **Unsaved changes**, so nothing is lost. Edits that are still pending in the Properties panel show **Properties pending** or **Rename pending** instead, until you apply or cancel them.

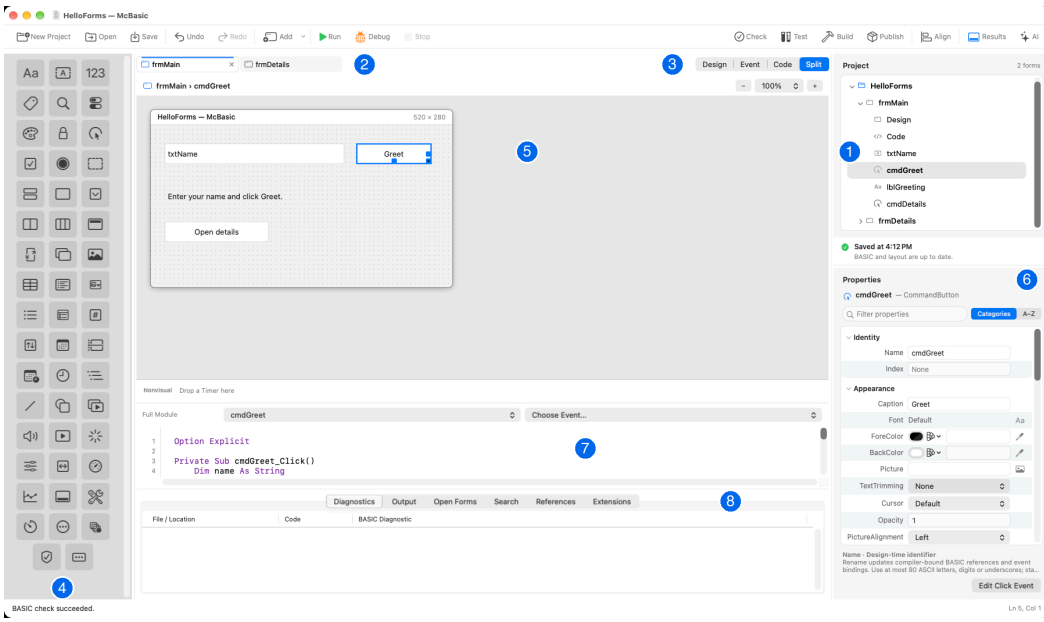


Figure 4.1: The McBasic workspace

Design and Code

When you select a form and choose **Design**, you see the form layout. When you choose **Code**, you see the BASIC code for that form. These are two views of the same project.

When you select a button and open its Click event, McBasic™ takes you to the procedure connected to that button. For example:

```
cmdGreet + Click  ->  cmdGreet_Click
```

The event view is a shortcut to the right piece of code. It does not create a separate copy of the code.

Documents, objects, and events

The document tabs keep several forms and source files open without losing your place. A form tab can show **Design**, **Event**, **Code**, or **Split**. Use them like this:

1. Open a form in **Design** when you need to place a control or change a

property.

2. Choose **Code** when you want to see the whole form module.
3. Choose **Event** when you want to work on one handler at a time.
4. Use the **Object** selector to choose the form or control.
5. Use the **Event** selector to choose an event declared for that object.

Event view is not a separate source file. Its line numbers are local to the displayed procedure so a short handler is easy to read. Diagnostics and breakpoints still point to the handler's real line in the form module.

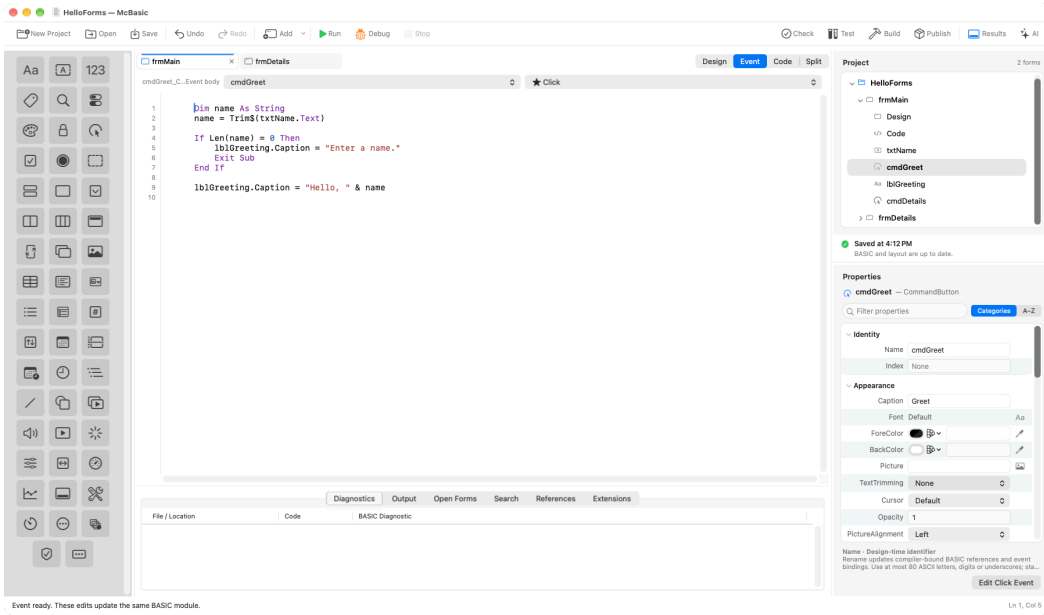


Figure 4.2: Design, Code, and Event

The document tab bar

The tab bar above the editor shows one tab for each open document. A tab shows the document's icon (a window for a form, a page for a module, a cube for a class) and its name. When the document has unsaved changes, a dot (•) follows the name. Hold the pointer over a tab to read the full name and the source path; for a

changed document the tooltip adds "Unsaved changes — Save writes all project documents."

- **Click a tab** to show that document. The active tab has a lighter fill and a short accent-coloured line on its top edge.
- **Tabs shrink to fit.** The tabs share the width of the bar, and a long name is shortened in the middle. Tabs do not get narrower than 120 points, so their names stay readable; only the active tab shrinks further in a very narrow window. When even the narrowest tabs do not fit, a » button (**More Documents**) appears at the right of the tabs. Click it for a menu of the hidden documents, with • after an unsaved one, and choose one to show it. The active tab is always visible.
- **Drag a tab** left or right to change the order. The order in the project tree does not change.
- **Move from tab to tab** with **Window > Show Next Tab** (Shift–Command–]) and **Window > Show Previous Tab** (Shift–Command–[). They follow the order of the bar, hidden tabs included, and wrap around at both ends. They need at least two open tabs.
- **Close a tab** with its **x** button (tooltip **Close Tab**), which appears on the active tab and on the tab under the pointer. Closing a tab only hides it from the bar: McBasic™ does not ask anything and discards nothing. Unsaved changes stay in the project, the project tree still shows the document as changed, and **Save** writes them. The last open tab cannot be closed.
- **Open a closed document again** by selecting it in the project tree. Its tab comes back next to the current tab.

The tab order and the closed tabs last while the project stays open. When you open the project again, every document has its tab again, in project order.

At the right end of the tab bar, the **Design | Event | Code | Split** control chooses the view of the current document:

View	What it shows
Design	The form designer.
Event	One event handler of the selected form or control (see the steps above).
Code	The whole BASIC module of the document.
Split	The form designer and the code side by side.

A module or class has no form, so its tab offers **Code** only.

The Properties area

The Properties area shows settings for the selected form or control. Common settings include `Name`, `Caption`, `Text`, `Width`, `Height`, and `Enabled`.

The top line names the selection, for example **cmdSave — CommandButton**. Below it, type in **Filter properties** to find a setting by name or category, and use **Categories I A–Z** to switch between grouped and alphabetical order. In **Categories**, click a group's triangle, or its title, to collapse or expand it; McBasic™ remembers which groups you closed. A filter still shows matching settings inside a closed group.

Each setting is one row: its name on the left, its value on the right. The **Aa** button beside **Font** opens the font chooser, and the eyedropper beside a color opens the color picker; the color swatch shows the current value. Beside the swatch, the **System colours** pull-down lists the 25 system color names (`ButtonFace`, `WindowText`, `Highlight` and so on), each with a swatch in the color it has for the project's **Appearance**. Choose one and it applies at once. A color row accepts `#RRGGBB`, one of these names (in any case) or nothing, which means the theme default. Any other text is refused with: "BackColor must be #RRGGBB, a system color name such as ButtonFace, or empty for the theme default." (the row shows the property's own name). See Chapter 2, "System colors", for the names and their values. A **Picture** row has a photo button that lists the project's image resources; choosing one applies it at once.

Select an empty part of the form to see the Form's own properties, grouped as Appearance, Layout, Form hosting, Keyboard actions, Window, Behavior, Data and Accessibility and help.

A change applies as soon as you finish it. Press Return or Tab, or click elsewhere, after typing a value; in the multi-line **Text** box, where Return and Tab type into the text, click elsewhere instead. A list choice, a check box, a color or a font applies the moment you pick it. Each applied setting is one step in **Edit > Undo**. Press Esc while typing to put back the value that was there before. If McBasic™ cannot use a value, it does not apply it: the reason appears in red under the row, and the text stays in the field so you can correct it. Typing a new **Name** and pressing Return, or leaving the field, renames the control; its event procedures and bindings change with it as one Undo step.

Some settings affect the design only. Some affect the running app. Some can be read but not changed while the app is running. If you are not sure, read the short description at the bottom of the Properties area. It describes the setting you are editing; point at a value to see the full description.

For example:

- `ReadOnly` stops the user from changing text in a running text box;

- `DesignLocked` stops you from moving or resizing that control in the designer.

These two settings sound similar, but they solve different problems.

Check before Run

Choose **Check** after a small change. It looks for problems in the complete project: code, names, forms, controls, and event connections.

Choose **Run** after Check succeeds. Then test the app as a user would. A successful Check means the project is valid enough to start; it does not mean you have tested every user action.

Test is the place for the project's automated checks when the project or release provides them. **Run** is the interactive application test. **Publish** is a separate delivery operation and should be tested outside the editor too. Keeping those words separate prevents a green syntax check from being mistaken for a finished product.

If you are not sure where to start

Use this five-step path:

1. Select the form in the project tree.
2. Choose **Design**.
3. Click the control on the canvas.
4. Read its **Name** in Properties.
5. Choose **Code** and open one of its events.

This path is often quicker than searching through files by hand.

Helpful editor tools

McBasic™ can help while you write code:

- completion suggests members after a control name or a known library value;
- **Edit > Complete BASIC** (Control–Space) requests suggestions when the popup is not already open;
- **Edit > Show BASIC Signature Help** (Control–Shift–Space) shows a callable's parameters and highlights the current one;
- **Edit > Find References** (Command–Shift–R) finds uses in the current project,
- Edit > Jump to Definition (Shift–F2, the VB6 shortcut), the first item of the code editor's right-click menu, or Command–click on a name opens the place where that name is declared: a procedure, Function or Property, a variable, constant, Enum or

Enum member, in any document of the project, also when you write it as Module.Proc or frmName.Proc. A module or class name opens that document; a Form or control name opens the form's designer with the control selected (for a control array, its member with the lowest Index). The status bar says where it went, for example "Function AcceptPuzzle · frmMain.bas:212", or why it could not go, for example "No definition found for X in this project. It may be built into McBasic™ or belong to a control or plug-in." or "X is declared in more than one place (A.bas, B.bas). Qualify it, for example A.X." Jump to Definition works even while the project has Check errors, but not while your app is running from the IDE;

including unsaved buffers;

- renaming a control in the Inspector can update its supported event procedure

and binding as one undoable change.

These tools help you write code. They do not save, Check, or run the project for you. A completion popup is a suggestion, not proof that the call is valid in your current project.

For larger edits, **Review Structured Form Patch** and **Review BASIC Procedure Patch** show a before/after preview and let you Apply or Cancel the change. McBasic™ validates the complete unsaved project before an approved patch is applied. Applying a patch still changes the editor buffer; choose **Save** when you want the change on disk.

The AI panel

The **AI** button sits at the right end of the toolbar, next to **Results**. Its icon is a group of sparkles. When the panel is open the button is tinted with the accent colour; choose it again, or press Shift–Command–L (**McBasic > Hide AI**), to hide the panel. You can drag the panel's edge to make it wider or narrower.

The panel is optional and sends nothing until you set up a provider and press Send. Its header holds the chat menu, the **Show exact content** button, and a gear that opens **AI Provider Settings**. The chips above the message field show which project documents the chat may send. Chapter 16, "Work with the AI assistant", explains the panel in full.

Add to Chat in the project tree's right-click menu adds the whole form, module, class or test to the chat's scope: the assistant may then change that document, and must ask your permission for anything else. The project row, the Design and Code rows and component rows have no such menu. The item is unavailable while the assistant works or while a Check, Build or Run is running.

Expected result

You should be able to complete this short path without searching through the file system:

1. select a form in the project tree;
2. find a control on the canvas;
3. read its Name in Properties;
4. open one event in Code view;
5. return to Design view.

Check your work

Choose a control in `HelloForms` and write down its Name and one event. Then open that event in Code view. If the procedure name matches the control and event, you have found the correct code path.

Small challenge

Select the greeting label and change a harmless design property, such as its Caption. Save, choose **Check**, and run the sample. Notice which changes are visible before code is involved.

Tips and tricks

- **Jump to where a name is declared.** Command–click a name, choose **Edit > Jump to Definition** (Shift–F2), or use the right-click menu in the code editor. It opens the procedure or variable in any file of the project, so you never have to search by hand.
- **Jump to Definition does not work while your app is running.** Press **Stop** first. It does work when the project has Check errors.
- **Find every use of a name.** Choose **Edit > Find References** (Command–Shift–R) before you rename or delete something, so you know what depends on it.
- **Switch between Design and Code to find a control's code.** Select the control in **Design**, read its **Name** in Properties, then choose **Code** and open one of its events.
- **Use Show exact content to see what the AI assistant will send.** The button is in the chat header. Nothing is sent until you press Send.
- **Want more room for your form?** Turn off **Show labels** in **McBasic > Settings… > Appearance**, or choose **Icon size: Small**.

- **Prefer big, labelled buttons on one line?** Choose **Toolbar: Classic** in **McBasic > Settings… > Appearance**. Switch back at any time; McBasic™ does not need to restart.
- **Step through your code with the mouse.** With **Toolbar: Ribbon**, the **Run** tab keeps **Continue**, **Step Over**, **Step Into** and **Step Out** next to **Run**, **Debug** and **Stop**.
- **Copy a form from another project** with **Project > Add Existing File…** (Command–D): select its `.bas` and `.form.json` together.
- **Duplicate Components** in the designer is Shift–Command–D.
- **Pick a color from System colours.** The pull-down beside a color swatch sets a name such as `ButtonFace`. The color then changes by itself when you switch the project between macOS and Classic.
- **Too many tabs?** Close the ones you are not using with their **x**: nothing is lost, unsaved changes stay and **Save** still writes them. Select a document in the project tree to bring its tab back, or use » to reach a hidden tab.

Chapter 5 — Design a form

A form is the window your user sees. A good form gives each control one clear job and makes the next action easy to find.

> ***WeatherDesk checkpoint***
> *Build this chapter's layout in the* `samples/WeatherDesk.mcbproj` *project. At this stage, the form has the city search, selected-place result, Save and Export buttons,*
> *and an empty grid. Do not add database or network code yet. Save, choose* ***Check****, and keep this as the first working version of the same project.*

Start with the user's task

Before you place controls, describe the task in one sentence:

> *The user enters a customer name, reads the feedback, and chooses Save or Cancel.*

That sentence suggests an input box, a message label, and two buttons.

Add a control

Use this simple order:

1. Select the form or container where the control should go.
2. Choose a component from the component rail.
3. Place it on the canvas.
4. Give it a useful **Name**.
5. Set the few properties needed for the current task.
6. Save and choose **Check**.

If you select a Panel first, the new control is placed inside that Panel. A Timer is different: it is not placed on the form canvas because it has no visible box.

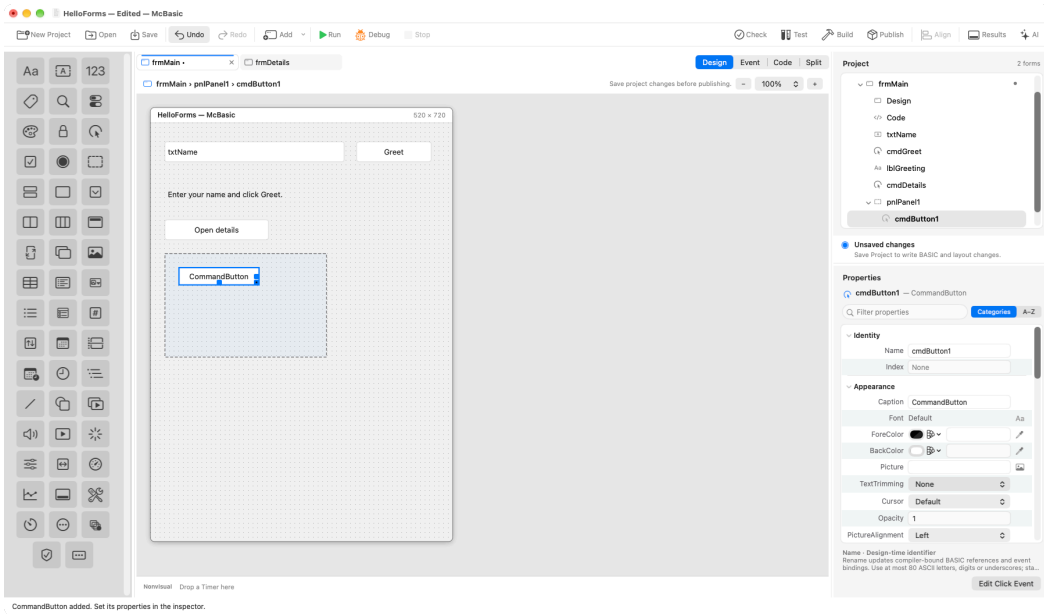


Figure 5.1: Add a control

The Form designer draws the controls in the look that the project uses. In a project whose **Appearance** is **Classic** (see Chapter 2), the designer previews the Classic look for buttons, text boxes, lists, combo boxes, check boxes, option buttons and frames, so the design looks like the running app.

Choose useful names

The Name is for your code. Choose a name that says what the control does:

Control	Example name	Meaning
TextBox	`txtCustomerName`	text entered by the user
CommandButton	`cmdSave`	starts the Save action
Label	`lblStatus`	shows a message
CheckBox	`chkIncludeInactive`	stores a yes/no choice
Timer	`tmrRefresh`	runs a repeated action

The Caption is different. The Caption is the text the user sees. You can change `Greet` to `Say hello` without changing the code name `cmdGreet`.

Set the first important properties

Start with `Name`, `Caption` or `Text`, size, position, and keyboard order. Later, add a tooltip, accessibility text, colors, borders, or a different font when those changes help the user.

You do not need to confirm a change. A typed value applies when you press Return or Tab or click elsewhere, and a choice from a list applies at once. If you change your mind, press Esc while typing, or choose **Edit > Undo** to step back one setting at a time.

Keep the layout easy to change

These properties solve different layout problems:

- **Anchor** keeps selected edges near the same edges of the parent;
- **Dock** places a control along one side or fills the remaining space;
- **Margin** adds space outside a control;
- **Padding** adds space inside a container.

For a first project, keep the rules simple. A small form can use top-left placement. A larger form might dock a navigation Panel on the left and let the main Panel fill the remaining space.

The Anchor editor

Select a control and open the **Layout** group of the Properties panel. The **Anchor** row shows a small picture of the control inside its container. Click a side to anchor that edge, and click it again to release it. An anchored side is drawn in the accent color. New controls are anchored Left and Top. At least one side must stay anchored; McBasic™ refuses to release the last one.

- Left and Top: the control keeps its position (the default).
- Left and Right: the control stretches horizontally with the window.
- Right and Bottom: the control follows the bottom-right corner, which suits OK and Cancel buttons.
- All four: the control grows and shrinks with its container.

VoiceOver reads each side as a checkbox ("Left anchor, checked").

The common visual contract

Many controls share a predictable group of properties. Learn the group once, then apply the same questions to a TextBox, a Button, a Panel, or a DataGrid.

Job	Properties to look for	What to ask
Size and position	`Left`, `Top`, `Width`, `Height`	Where does it belong, and how much room does it need?
Resize behavior	`Anchor`, `Dock`, `MinWidth`, `MaxWidth`, `MinHeight`, `MaxHeight`	What should change when the window changes size?
Visibility and input	`Visible`, `Enabled`, `HitTestVisible`, `AllowDrop`, `DragMode`	Should the user see it, use it, drop data on it, or drag it?
Spacing and drawing	`Margin`, `Padding`, `BackColor`, `ForeColor`, `BorderStyle`	Is the group readable and separated?
Keyboard use	`TabIndex`, `TabStop`, `Default`, `Cancel`	Can the user move through the task and escape safely?
Assistance	`ToolTipText`, `AccessibleName`, `AccessibleDescription`, `AccessibleRole`, `AutomationId`	Can a person understand the control without relying on color or position?

These properties have different jobs. `ReadOnly` protects text from editing; `Enabled` changes whether the control can be used; `Visible` removes it from the running view; `DesignLocked` only protects the control from accidental movement in the Designer. Do not use one as a substitute for another.

`Dock` is useful for a fixed strip or a main area that fills the remaining space. `Anchor` is useful when a control should keep its distance from selected edges. When both are present, Dock determines the active layout while Anchor is kept for later use. This lets you change the layout without losing the earlier anchor choice.

Containers are part of the design

The selected parent matters. The standard containers have different jobs:

- `Panel` groups ordinary controls;
- `Frame` groups them with a visible caption and boundary;
- `FlowLayoutPanel` places children in order, wraps them, and supports a

configurable direction, spacing, and cross alignment;

- `ScrollContainer` gives a larger content area a scrollable surface;
- `SplitContainer` gives two areas a resizable divider and can collapse either

pane;

- `TabControl` holds `TabPage` objects so related tasks can share a window;
- `Expander` hides optional detail without opening another window.

When you add a control, select its parent first. If the parent is wrong, move the control in the project tree or undo the operation before adding more children. A tidy hierarchy makes both the Designer and the BASIC source easier to understand.

Right-click menus

Right-click a control on the canvas, or click it with two fingers. If the control is not selected, McBasic™ selects it first. If you right-click a control inside a selection of several controls, the selection stays, and the commands act on all of it. The menu holds:

Item	What it does
Add to Chat	Adds the control, or the whole selection, to the AI chat's scope: the assistant may then change those controls and their event handlers, and must ask your permission for anything else. The chat panel opens. See "Add to Chat and the scope" in Chapter 16, "Work with the AI assistant".
Add Array to Chat	Shown only for a member of a control array: adds the whole array and its shared event procedures to the AI chat's scope. See "Control arrays: many controls, one name" below.
View Code	Opens the control's default event procedure, as a double-click does. For a control without one, it switches to **Code**.
Cut, **Copy**, **Paste**, **Duplicate**, **Delete**	The same commands as in the **Edit** menu.
Create Control Array…	Opens the sheet that turns the control into a grid of controls with one name. Available when you right-click one visual control that is not in an array. Not shown in a UserControl. See "Control arrays: many controls, one name" below.
Bring to Front, **Send to Back**	Changes the drawing order of the selected controls.
Lock or **Unlock**	Sets the `DesignLocked` property, which protects a control from accidental movement in

Item	What it does
	the Designer. A locked control shows **Unlock**. The item appears only for controls that have `DesignLocked`.
Properties	Moves the keyboard focus to the Properties area.

Right-click the empty part of the form to get the form's menu. It leaves the selection as it is:

Item	What it does
Add Form to Chat	Adds the whole form, its layout and its code, to the AI chat's scope.
Paste	Pastes the controls on the clipboard.
Select All	Selects every control on the form.
View Code	Switches to **Code**.

Control-click. A Control-click on a control opens its menu when you release the mouse button, if the control did not move. If you hold Control and drag the control instead, it still moves into the container you drop it on, such as a Frame or a Panel, and no menu opens. A Control-click on the empty part of the form opens the form's menu.

While McBasic is busy (the AI assistant works, or a Check, Build or Run is running), the menus still appear, but **Add to Chat** and **Add Form to Chat** are unavailable, and Cut, Copy, Paste, Duplicate, Delete, Bring to Front, Send to Back, Lock and Unlock follow the **Edit** menu, which disables them. **View Code** still works; while a Check, Build or Run is running it shows the code instead of the event. **Properties** works for a selected control. A right-click while McBasic™ is busy does not change the selection.

Components without a visible box, such as a Timer, have no right-click menu. To give one to the AI assistant, add the whole form with **Add Form to Chat**.

Control arrays: many controls, one name

A control array is a group of controls of one type that share a name and differ by their **Index**, a whole number from 0 to 32767. A 9×9 board is one array, `txtCell`, whose members are `txtCell(0)` to `txtCell(80)`. One event procedure serves every member and receives the Index of the control that raised the event (Chapter

6). The designer, the project tree and the code editor's object list show each member as `txtCell(37)`.

Create Control Array… builds a grid in one step. Select one control that is not already in an array, then choose **Design › Create Control Array…**, **Align ▾ › Create Control Array…** in the toolbar (on the Ribbon, the **Design** tab, **Arrange** group), or right-click the control and choose **Create Control Array…**. In the sheet:

Field	Meaning
Columns	members in each row, 1 to 32
Rows	rows of members, 1 to 32
Spacing	the gap between members, in points (0 to 1000)
Group every	a wider gap after every N rows and columns, such as 3 for the boxes of a Sudoku board; 0 for none
Group spacing	the extra gap between groups, in points (0 to 1000)
Start Index	the Index of the first member; the others count up row by row

Click **Create**. The selected control becomes the first member; the others copy its properties, take the next Index row by row, and follow its **TabIndex**. A form can hold at most 2048 controls and components after Create Control Array. A value out of range, or a grid that would pass Index 32767, shows a red message in the sheet, and nothing changes until you correct it.

You can also add members while the app runs, with `Load txtCell(n)` and `Unload txtCell(n)`. The new member copies the lowest member you designed, so set its size, anchors and container in the designer first. Chapter 6 gives the rules and an example.

If the control already has event procedures, such as `txtCell_Change()`, they get `Index As Integer` as their first parameter in the same step (`txtCell_Change(Index As Integer)`), so the project keeps compiling. A ContextMenu or ValidationProvider whose **Target** named the control now names the first member, such as `txtCell(0)`. When the change would break other code, for example a line that calls `txtCell_Change` directly, the sheet says why and creates nothing. The same happens when the control has event procedures or is used in code and the project does not compile yet. Fix that code first, then click **Create** again. One **Undo** removes the whole array, gives the first control back without an Index, and restores its event procedures and Targets.

Paste also makes arrays, as in Visual Basic 6. When you paste or duplicate one control whose name and type match an existing control, McBasic™ asks "A control named txtName already exists. Do you want to create a control array?". **Yes** makes the original `txtName(0)` and the copy `txtName(1)`, and the event procedures of the original, such as `txtName_Change()`, get `Index As Integer` as their first parameter automatically (`txtName_Change(Index As Integer)`), so the project keeps compiling. A ContextMenu or ValidationProvider **Target** that named the original now names `txtName(0)`. One **Undo** reverts the paste and the new headers together. **No** pastes the copy under a new name and changes no code.

McBasic™ checks **Yes** before it pastes. When code uses `txtName` in a way that would stop compiling once it is an array, such as `txtName.Text = ""`, or when the project does not compile yet, an alert lists the code to change and nothing is pasted. Update those lines (for example to `txtName(0).Text`), or paste again and answer **No**.

A pasted member of an array that is on the form joins it with the next free Index, without asking. When you paste several controls at once, members of an array join it, and the other controls get new names.

Rename. Typing an existing name of the same type in **Name** asks the same question. **Yes** makes the control a member with the next free Index, and, when the existing control was a single control, gives its event procedures `Index As Integer` in the same way as paste. **No** leaves the name unchanged. Renaming a member asks whether to **Rename Array** (every member and its event procedures get the new name) or to **Take Out of Array** (only this control, under the new name, without its Index and its event bindings); **Cancel** keeps the name. Either choice is one **Undo** step.

The Index row sits under **Name** in Properties. Type a number to make the control a member, or clear it to take the control out of its array; clearing works only when no other control shares the name (otherwise give it a new **Name**). When the number is already used, or out of range, a red message under the row names the problem and the Index stays as it was. The Index row does not change your code, as in Visual Basic 6: when a control becomes a member this way, give its event procedures `Index As Integer` yourself, and Check tells you which ones need it.

Delete. Deleting a member removes only that member; the others keep their Index, and gaps are allowed. When a ContextMenu targets a deleted control, it keeps its items but loses its **Target**: set a new Target before Check or Run.

Rules McBasic™ checks: every control that shares a name has an Index; no two members have the same Index (gaps are allowed); all members have the same type; members may sit in different containers. Non-visual components (Timer, ImageList, …) and controls from plugins cannot be arrays, and a UserControl cannot contain one: in a UserControl the **Index** row and the right-click **Create**

Control Array… are not shown, and **Design › Create Control Array…** is unavailable. A ContextMenu or ValidationProvider **Target** names one member, as `txtCell(5)`; the array name alone is refused.

Right-click a member and choose **Add Array to Chat** to give the AI assistant the whole array and its shared event procedures (Chapter 16). The right-click menu of a member shows **Add Array to Chat** after **Add to Chat**. **Create Control Array…** follows **Duplicate** in the same menu, and is available for one visual control that is not in an array.

Make a form usable with the keyboard

Set `TabIndex` in the order a person completes the task. Set `TabStop = False` for a control that should be visible but not entered during ordinary Tab navigation. Mark the main CommandButton `Default = True` when Enter should submit the form. Mark the safe escape action `Cancel = True` when Escape should activate it. Test these choices with the keyboard; do not assume that a form which looks correct is easy to use.

Plan for accessibility while you design

Give an input a meaningful `AccessibleName`, such as “Book title”, even if a nearby Label already says the same thing. Use `AccessibleDescription` for a short instruction, not for a duplicate paragraph. Use `AccessibleRole` when a generic visual arrangement needs a clearer role. `AutomationId` is a stable identifier for UI testing; it is not a replacement for the BASIC `Name`.

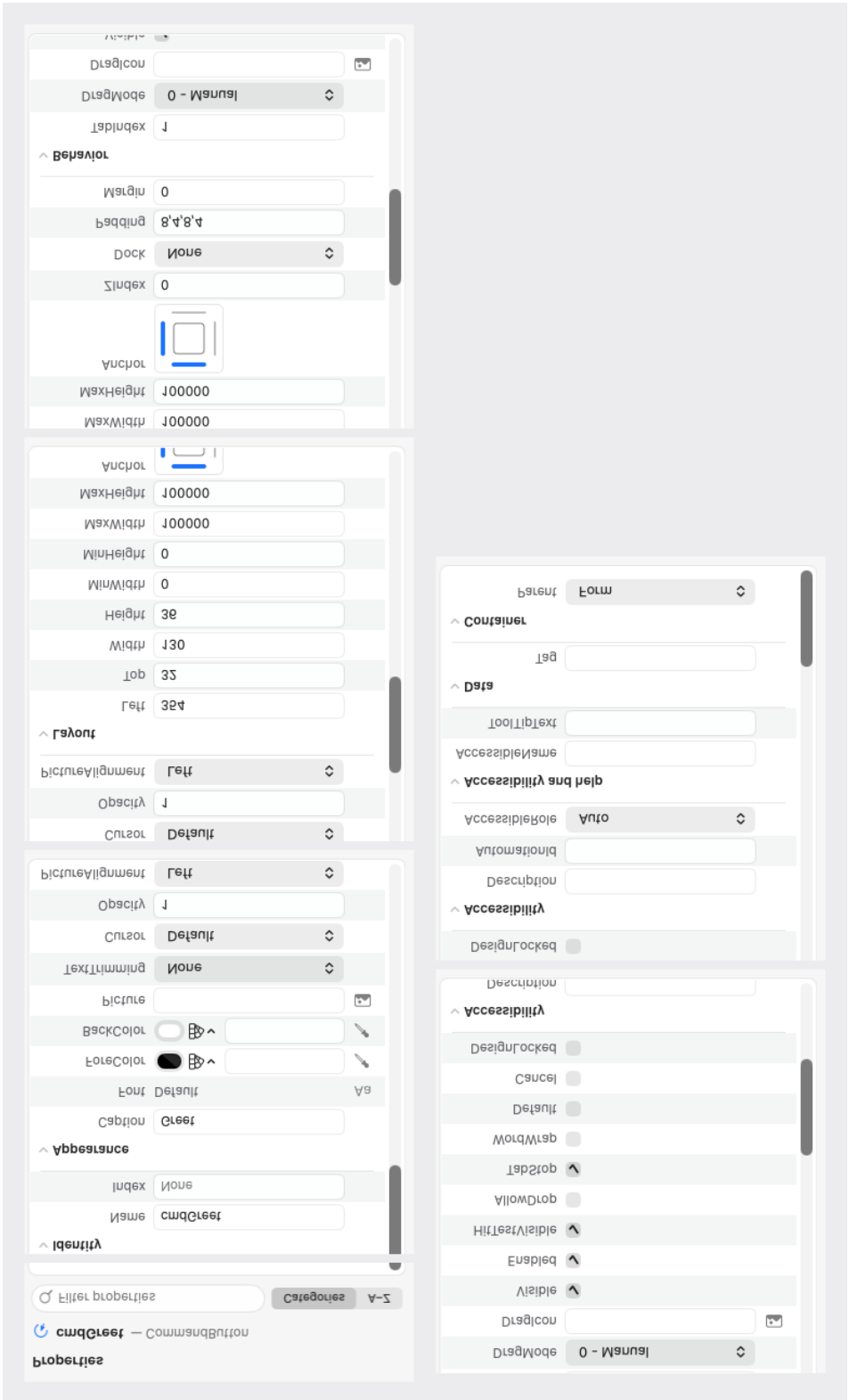


Figure 5.3: The common property contract

Practice form: customer entry

Create a form with these controls:

Type	Name	Job
TextBox	`txtName`	enter a customer name
Label	`lblStatus`	show feedback
TextBox	`txtNotes`	enter optional notes
CommandButton	`cmdSave`	try to save the entry
CommandButton	`cmdCancel`	close the entry form

Set `txtNotes.MultiLine` to `True`. Give the controls a TabIndex order that follows the task.

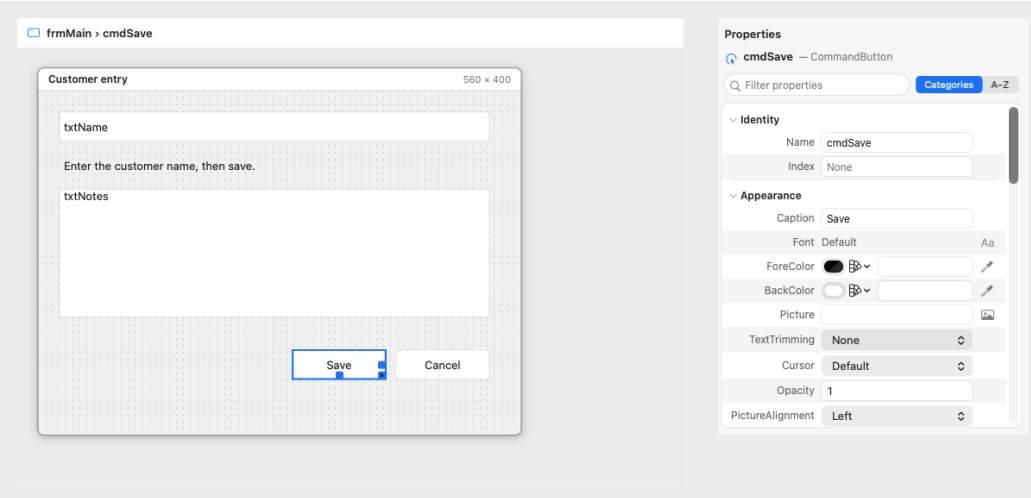


Figure 5.2: Customer entry form

Check your work

- Every control has one clear job.
- Every control has a useful Name.
- The controls are inside the parent you intended.
- Tab moves through the form in a sensible order.
- Check succeeds before you begin writing event code.

Tips and tricks

- **Use Create Control Array… for a grid of cells.** Select one control and choose **Design > Create Control Array…**, or right-click the control and choose **Create Control Array…**. All the cells share one name and one event procedure, and the `Index` parameter tells you which cell was used.
- **Use Group every for boxes like a Sudoku board.** Set **Group every** to 3 and give **Group spacing** a few points, so the nine boxes are easy to see.
- **Design in the look you will ship.** Choose the **Appearance** in **Project > Project Settings…** before you place many controls, so the designer shows the real look and the sizes you pick fit it.
- **Name controls before you write code.** Names such as `txtCell` or `cmdSave` tell you the control type when you read the code later, and the AI assistant uses them too.
- **Set the Anchor properties early.** The Anchor editor decides which controls stay put and which stretch when the window is resized. It is easier to set while the form is still small.

Chapter 6 — Write McBasic™ event code

An event turns a user action into a result. We will use the greeting from Chapter 3 to look at the pieces of an event procedure and then change it.

> ***WeatherDesk checkpoint***
> *Add the first event procedures to* `samples/WeatherDesk.mcbproj`*:* `cmdClear_Click`*,* `cmdSave_Click`*, and a temporary* `cmdFind_Click` *that only checks for an empty city. The complete project stays in the repository; the code in this chapter explains the change made at this checkpoint.*

Start with `Option Explicit`

Put this at the top of a form module:

```
Option Explicit
```

It asks McBasic™ to require declarations for the names you use. This helps find spelling mistakes before they become confusing bugs.

Declare and change values

Use `Dim` to create a variable:

```
Dim customerName As String
Dim saveCount As Long

customerName = Trim$(txtName.Text)
saveCount = saveCount + 1
```

A `String` holds text. A `Long` is useful for a whole-number count. Use a `Boolean` for true/false choices and a `Date` for dates.

The `&` operator joins text:

```
lblStatus.Caption = "Ready to save " & customerName
```

Choose a scope for each value

Start with the smallest scope that solves the problem:

Declaration	Where the value is available	A good use
`Dim` inside a procedure	That procedure	A temporary name or count
`Private` at form level	Procedures in that form	The open database or a busy flag
`Public` at form level	Other forms that have a reference to it	A value passed to a details form
`Const`	Wherever the declaration is visible	A fixed limit or message

This choice affects how long the value stays alive. A local variable is created when the procedure runs. A form-level variable remains available while the form is open. Do not make every value public just because it is convenient; it makes it harder to tell which procedure can change it.

```
Option Explicit

Private mSaveCount As Long
Private Const MaxNameLength As Long = 80

Private Sub cmdSave_Click()
    Dim customerName As String
    customerName = Trim$(txtName.Text)

    If Len(customerName) > MaxNameLength Then
        lblStatus.Caption = "Use 80 characters or fewer."
        Exit Sub
    End If

    mSaveCount = mSaveCount + 1
    lblStatus.Caption = "Save attempt " & CStr(mSaveCount)
End Sub
```

The two `Dim` variables in this example have different lifetimes. The text is needed only while the Click event runs; `mSaveCount` must survive until the next click, so it belongs at form level.

Separate assignment from comparison

An equals sign can assign a value or compare two values. The surrounding statement tells you which job it has:

```
customerName = "Ada"              ' assignment
If customerName = "Ada" Then      ' comparison
    lblStatus.Caption = "Known customer"
End If
```

Use = for equality, <> for “not equal”, and <, >, <=, or >= for ordered comparisons. Use `And`, `Or`, and `Not` when a decision has more than one condition. Parentheses make a mixed condition easier to read; do not rely on the reader remembering precedence rules.

```
If Len(Trim$(txtName.Text)) > 0 And chkActive.Value Then
    cmdSave.Enabled = True
Else
    cmdSave.Enabled = False
End If
```

Use `If` for a decision and `Select Case` for choices

Use `If` when you are asking one question or when the second test depends on the first. Use `Select Case` when one value has several named ranges or exact choices:

```
Private Function ScoreMessage(ByVal score As Long) As String
    Select Case score
    Case Is < 0
        ScoreMessage = "Score cannot be negative."
    Case 0 To 49
        ScoreMessage = "Try again."
    Case 50 To 79
        ScoreMessage = "Passed."
    Case 80 To 100
        ScoreMessage = "Passed with distinction."
    Case Else
        ScoreMessage = "Enter a score from 0 to 100."
    End Select
End Function
```

The function evaluates the input once and returns one message. Keeping the decision in a function makes the button event short and gives you one place to test the boundary values `-1`, `0`, `49`, `50`, `79`, `80`, `100`, and `101`.

Check a value before using it

Put the invalid case first. This keeps the normal case easy to read:

```
Private Sub cmdSave_Click()
    Dim customerName As String
    customerName = Trim$(txtName.Text)

    If Len(customerName) = 0 Then
        lblStatus.Caption = "Enter a customer name."
        txtName.SetFocus
        txtName.SelectAll
        Exit Sub
    End If

    lblStatus.Caption = "Ready to save " & customerName
End Sub
```

If the name is empty, the procedure shows a message, returns the cursor to the input box, selects its current text, and stops. It does not continue to the success message.

Use control properties and methods

The dot joins a control name to one of its members:

```
txtName.Text = "Ada"
txtName.ReadOnly = True
txtName.SetFocus
lblStatus.Caption = "Name accepted."
cmdSave.Enabled = False
```

Properties describe or change a control. Methods ask a control to do something. Completion can show the members available after the dot.

Events start procedures

An event is something that happens. A Click event happens when a button is activated. A Change event happens when input changes. The event is connected to a procedure:

```
cmdSave + Click  ->  cmdSave_Click
txtName + Change ->  txtName_Change
```

Keep each event procedure focused on its own job. Move shared work into a separate procedure when several events need it.

One event procedure for a control array

The members of a control array (Chapter 5) share one procedure per event. Its first parameter is `Index`, the member that raised the event:

```
Private Sub txtCell_Change(Index As Integer)
    lblStatus.Caption = "Cell " & Index & " is now " &
txtCell(Index).Text
End Sub
```

Write `Index As Integer` or `ByVal Index As Integer`; `Long` works too. An event with its own parameters keeps them after `Index`: `Private Sub txtCell_KeyPress(Index As Integer, ByVal Character As String, ByRef Handled As Boolean)`. For the ListView `ItemClick` and `ItemDoubleClick` events, which already carry an index, name the second one `ItemIndex`. An `Async Sub` handler takes `ByVal Index As Integer`. Double-click a member in the designer to open this procedure, or to create it with the right parameters, such as `Private Sub txtCell_Change(ByVal Index As Integer)`.

In code, `txtCell(i)` (or `txtCell.Item(i)`) is the member with Index `i`, and every property and method of a single TextBox works on it. `txtCell.Count`, `txtCell.LBound` and `txtCell.UBound` give the members' number and range, and `For Each cell In txtCell` visits them in Index order:

```
Dim cell As TextBox
For Each cell In txtCell
    cell.Text = ""
Next
```

An Index the array does not have raises error 340, "Control array element doesn't exist", with the array and the Index, such as `txtCell(99)`.

Adding and removing members while the app runs

`Load txtCell(i)` adds a new member with Index `i` to a control array of the same form; `Unload txtCell(i)` removes it again. The new member is a copy of the array's lowest member, with the layout values you set in the designer (not values that code changed later). It lands in the same container, at the same position, as that member, and it gets the next TabIndex after the form's highest. It is invisible until your code sets `Visible = True`, so move it with `Left` and `Top` first, then show it. A copied OptionButton starts unchecked. The shared handlers serve the new member and receive its Index, as for any other member, and it counts in `Count`, `LBound`, `UBound` and `For Each`.

This example adds a button to a row each time **Add** is clicked. `cmdItem(0)` is a button in the designer that has Index 0:

```
Private Sub cmdAdd_Click()
    Dim n As Integer
    n = cmdItem.UBound + 1
    Load cmdItem(n)
    cmdItem(n).Left = cmdItem(n - 1).Left + cmdItem(n - 1).Width + 8
    cmdItem(n).Caption = "Item " & n
    cmdItem(n).Visible = True
End Sub

Private Sub cmdItem_Click(Index As Integer)
    lblStatus.Caption = "You clicked item " & Index
End Sub

Private Sub cmdRemove_Click()
    If cmdItem.UBound > 0 Then Unload cmdItem(cmdItem.UBound)
End Sub
```

`Unload` also accepts a control variable (`Unload t`) and `Unload Me.Controls("cmdItem(3)")`. When the unloaded member had the focus, the focus moves to the next tab stop. Only a member that code loaded can be unloaded: a member you placed in the designer stays, so hide it with `Visible = False`.

`For Each` over an array whose members the form loads, and over `Me.Controls` in such a form, visits the controls that exist when the loop starts. Loading or unloading inside the loop is safe, and the members you add do not appear in that loop.

These run-time errors can be trapped like any other (see the error handling section of this chapter):

Error	Message	Fix
340	Control array element doesn't exist	Use an Index the array has; `Unload` of an Index that is not loaded raises it too.

Error	Message	Fix
360	Object already loaded	`Load` only an Index the array does not have, or `Unload` it first.
361	Can't load or unload this object	The template is docked or is a container that Check could not see, or you call `Load` outside the form that holds the array. Write the call in that form's code.
362	Can't unload controls created at design time	Hide the member with `Visible = False`, or unload a member that `Load` created.
364	Object was unloaded	A variable still points at a removed member. Load it again or use another member.
365	Unable to unload within this context	A member cannot unload itself inside its own event (including the first part of an `Async` handler). Unload it from another control's event, or after the event ends.
	That procedure	A temporary name or count
Check also reports mistakes before you run. It refuses `Load` of a control that	That procedure	A temporary name or count
is not in an array, of a bare array name (`Load txtCell`), of a control	That procedure	A temporary name or count
variable, of a control on another form, and of a literal Index outside 0 to	That procedure	A temporary name or count
32767; the messages name the fix, such as "Load txtCell needs the Index of the	That procedure	A temporary name or count
new member: write Load txtCell(index), such as Load txtCell(3)." It also refuses	That procedure	A temporary name or count

Error	Message	Fix
arrays of containers or TabPages, and templates that are docked or inside a	That procedure	A temporary name or count
SplitContainer. A warning, MCB2404, appears for a literal Index that always	That procedure	A temporary name or count
fails: "Load txtCell(0) always raises error 360: txtCell(0) was placed in the	That procedure	A temporary name or count
designer and is already loaded; use an Index the array does not have, such as	That procedure	A temporary name or count
Load txtCell(3)." and "Unload txtCell(0) always raises error 362: txtCell(0) was	That procedure	A temporary name or count
placed in the designer, and only members loaded with Load can be unloaded; hide	That procedure	A temporary name or count
it with txtCell(0).Visible = False."	That procedure	A temporary name or count
	That procedure	A temporary name or count
Limits in this version: validation providers and context-menu targets are	That procedure	A temporary name or count
resolved when the form opens, so they never name a loaded member; a loaded	That procedure	A temporary name or count
member can be reached with Tab but never gets the initial focus; and `Load`	That procedure	A temporary name or count
works only in the code of the array's own form.	That procedure	A temporary name or count

Subs and Functions

A `Sub` performs an action. A `Function` returns a value:

```
Private Function CleanName(ByVal value As String) As String
    CleanName = Trim$(value)
End Function

Private Sub cmdGreet_Click()
    Dim name As String
    name = CleanName(txtName.Text)
    lblGreeting.Caption = "Hello, " & name
End Sub
```

The function returns its result by assigning to the function name.

Use parameters to make a procedure independent of a particular control. Use `ByVal` when the procedure should receive a value to read. Use `ByRef` only when the procedure is meant to change the caller's variable:

```
Private Sub AddOne(ByRef value As Long)
    value = value + 1
End Sub

Private Sub cmdCount_Click()
    Dim count As Long
    AddOne count
    lblStatus.Caption = "Count: " & CStr(count)
End Sub
```

After `AddOne count`, the local variable contains `1`. That is the point of `ByRef`: the procedure works with the caller's storage. If a procedure only needs to read a value, prefer `ByVal` so the intention is clear.

Learn the event families

The Event selector shows the events that the selected object really supports. The following families are useful in everyday applications:

Form life cycle

`Initialize`, `Load`, and `Shown` are useful for setup. `Activate` and `Deactivate` tell you when a window becomes or stops being the active window. `Resize` is useful for a status message or a layout rule. `Closing`, `QueryUnload`, `Unload`, and `Closed` belong to the closing path.

Use the `ByRef` flag to stop a close before you have saved work:

```
Private Sub Form_Closing(ByRef Cancel As Boolean)
    If Len(Trim$(txtNotes.Text)) > 0 Then
        Cancel = True
        lblStatus.Caption = "Save or clear the note before closing."
    End If
End Sub
```

If `Cancel` becomes `True`, the form remains open. If closing is accepted, the later close events can run. Keep the check quick; a closing event is a poor place for a long network request.

Pointer and keyboard input

`MouseDown` and `MouseUp` receive a button, modifier keys, and local `X` and `Y` coordinates. `MouseMove` receives a `MouseButtons` set because more than one button can be held. `MouseEnter`, `MouseLeave`, `MouseWheel`, and `DoubleClick` have their own simple signatures.

```
Private Sub pnlCanvas_MouseMove( _
    ByVal Button As MouseButtons, _
    ByVal Shift As KeyModifiers, _
    ByVal X As Double, _
    ByVal Y As Double)
    If (Button And MouseButtons.Left) = MouseButtons.Left Then
        lblStatus.Caption = "Pointer: " & CStr(X) & ", " & CStr(Y)
    End If
End Sub
```

`KeyDown` and `KeyUp` describe a key. `KeyPress` provides committed Unicode text. Their handlers can set `Handled = True` when the control should not carry out its normal action. Set `Form.KeyPreview = True` when the form must see a key before the focused control. When no control has the focus, for example because every control is hidden or disabled, the form receives `KeyDown`, `KeyUp` and `KeyPress` itself, whatever `KeyPreview` says, as in Visual Basic 6.

`GotFocus` and `LostFocus` are parameterless. Use them to update a hint or validate a field after the user leaves it. Do not put a database save in every focus event; focus can change for ordinary navigation.

Value, selection, and popup events

Use `Change` for a value change, `SelectionChanged` for a selection change, `DropDown` and `DropDownClosed` for a ComboBox popup, and `CalendarOpened`/`CalendarClosed` for a DatePicker. Slider has `DragStarted` and `DragCompleted` in addition to its value event. Opening a popup is not itself a value change.

Drag and drop

Set `AllowDrop = True` on the receiving control. `DragEnter` and `DragOver` receive `DragData`, a `ByRef DragEffect`, and local coordinates. `Drop` receives the accepted effect and data. A file drop can be handled without knowing which desktop pasteboard supplied it:

```
Private Sub pnlDrop_DragEnter( _
    ByVal Data As DragData, _
    ByRef Effect As DragEffect, _
    ByVal X As Double, _
    ByVal Y As Double)
    If Data.Files.Count > 0 Then
        Effect = DragEffect.Copy
    Else
        Effect = DragEffect.None
    End If
End Sub
```

```
Private Sub pnlDrop_Drop( _
    ByVal Data As DragData, _
    ByVal Effect As DragEffect, _
    ByVal X As Double, _
    ByVal Y As Double)
    If Data.Files.Count > 0 Then
        lblStatus.Caption = Data.Files.Item(1)
    End If
End Sub
```

The `DragEffect` values are `None`, `Copy`, `Move`, and `Link`. Keep the drop handler small: inspect the file, then call a separate procedure to load it.

Drag a control inside your app

The events above are for data dragged in from outside the app. To let the user drag one of your own controls onto another control or onto the form, as in Visual Basic 6, use `DragMode`, `DragIcon`, the `Drag` method and the `DragOver` and `DragDrop` events. They work on every visual control.

- `DragMode = 1` (Automatic) starts a drag as soon as the user presses the left mouse button on the control. While `DragMode` is Automatic the control does not raise `Click`, `MouseDown` or `MouseUp` for that press. With `DragMode = 0` (Manual, the default) you start the drag yourself with `Drag`, for example from `MouseDown`.
- `DragIcon` names a project image. The drag shows it under the pointer. Without one, a grey outline the size of the control follows the pointer. The control itself does not move.
- `ctl.Drag` (or `ctl.Drag vbBeginDrag`) begins a drag, `ctl.Drag vbEndDrag` drops it now, and `ctl.Drag 0` cancels it. The user can also press Escape to cancel: no `DragDrop` is raised. The drag is also cancelled when the app loses the focus, or when the dragged control is hidden, disabled or unloaded or its form closes.
- The target is the topmost visible, enabled control under the pointer, or the form when there is none. A control can be its own target, and a drop on another form of the app reaches that form's controls.

The target receives `DragOver` while the pointer is over it and `DragDrop` when the button is released. `Source` is the dragged control, so `Source.Name`, `TypeOf Source Is Panel`, `Source.Left` and `Source.Move` work. `X` and `Y` are relative to the target. `State` tells `DragOver` what happened: `vbEnter` (0) when the pointer enters the target, `vbOver` (2) on each move and `vbLeave` (1) when it leaves or the drag is cancelled.

A control has two kinds of `DragOver` handler: the one for files and text dragged from outside the app, and the one for a control dragged inside the app. They have the same name, `ctl_DragOver`, and McBasic™ tells them apart by the parameter list. Use the first for data from outside and the second for a control drag:

```
' Files or text dragged from outside the app
Private Sub pnlDrop_DragOver( _
    ByVal Data As DragData, _
    ByRef Effect As DragEffect, _
    ByVal X As Double, _
    ByVal Y As Double)

' A control dragged inside the app (DragMode or Drag)
Private Sub pnlDrop_DragOver( _
    ByVal Source As Control, _
    ByVal X As Single, _
    ByVal Y As Single, _
    ByVal State As Integer)
```

A control can have only one `<Name>_DragOver` Sub, so it uses either the control-drag form or the outside-drop form, not both. The code editor's event list shows the two as "DragOver (Source, X, Y, State)" and "DragOver (Data, Effect, X, Y)"; choosing the second when the first exists reports the existing-handler error. Any other parameter list for `<Name>_DragOver` is reported by Check with both accepted forms. A control array adds a leading `Index As Integer`. `DragDrop` has only one form, `Sub pnlDrop_DragDrop(ByVal Source As Control, ByVal X As Single, ByVal Y As Single)`, and the form has `Form_DragOver` and `Form_DragDrop` with the same parameters. A `DragDrop`, `Form_DragOver` or `Form_DragDrop` Sub with any other parameter list stays an ordinary Sub, as in projects written before these events existed.

This example drags a `Panel` named `pnlCup` (`DragMode = 1`) onto a target panel `pnlTable`, and drops it into place:

```
Private Sub pnlTable_DragOver( _
    ByVal Source As Control, _
    ByVal X As Single, _
    ByVal Y As Single, _
    ByVal State As Integer)
    If State = vbEnter Then pnlTable.BackColor = RGB(220, 240, 220)
    If State = vbLeave Then pnlTable.BackColor = RGB(255, 255, 255)
End Sub

Private Sub pnlTable_DragDrop( _
    ByVal Source As Control, _
    ByVal X As Single, _
    ByVal Y As Single)
    pnlTable.BackColor = RGB(255, 255, 255)
    Source.Move pnlTable.Left + X - Source.Width / 2, pnlTable.Top + Y -
Source.Height / 2
End Sub
```

To "shake the cup", drag the cup over itself: the cup is its own target, so `pnlCup_DragOver` and `pnlCup_DragDrop` run for it.

DataGrid and TreeView events

These controls have events that carry useful row, column, or node information. Typical DataGrid events include `CellClick`, `CellDoubleClick`, `CellBeginEdit`, `CellEndEdit`, `CellValueChanged`, `RowEnter`, `RowLeave`, `ColumnHeaderClick`, `Sorting`, `Sorted`, and `DataError`.

```
Private Sub grdBooks_Sorting( _
    ByVal ColumnIndex As Integer, _
    ByRef Cancel As Boolean)
    If ColumnIndex = 0 Then
        Cancel = True
        lblStatus.Caption = "Book ID is fixed order."
    End If
End Sub
```

TreeView supplies `TreeNode` references for `BeforeSelect`, `AfterSelect`, `BeforeExpand`, `AfterExpand`, `BeforeCollapse`, `AfterCollapse`, `NodeClick`, and `NodeDoubleClick`. A `Before...` event can cancel a transition with its `ByRef Cancel` value. Do not confuse a zero-based DataGrid row index with a database key, and do not use a TreeView's displayed position as a permanent ID.

If you are unsure of a signature, select the object and event in the editor and let McBasic™ create the procedure. This is safer than copying a similarly named event from another BASIC dialect.

Give a procedure one job

An event procedure should usually connect the control to a small helper. A useful split is:

1. read and clean the control value;
2. validate it;
3. call a function or procedure that does the work;
4. show success or failure.

For example, the button below does not know how a name is cleaned. That rule lives in `CleanName`, where it can be reused and tested:

```
Private Function CleanName(ByVal value As String) As String
    CleanName = Trim$(value)
End Function

Private Sub cmdGreet_Click()
    Dim name As String
    name = CleanName(txtName.Text)

    If Len(name) = 0 Then
        lblGreeting.Caption = "Enter a name."
        Exit Sub
    End If

    lblGreeting.Caption = "Hello, " & name
End Sub
```

Give one procedure to the AI assistant

A small procedure with one job is also the easiest thing to hand to the optional AI assistant. Right-click (or Control-click) anywhere inside a `Sub`, `Function` or `Property` in the code editor. When the click is inside a procedure, the menu starts

with **Add Procedure to Chat**. Choose it to add that procedure to the AI chat's scope, for example ◎ `CleanName (frmMain)`: the assistant may then change that procedure's body and signature, it reads the declarations the procedure uses, and it must ask your permission before it changes anything else. The item also works in the **Event** view, where it adds the handler you clicked in. Each part of a property (`Get`, `Let` or `Set`) is added separately. The assistant cannot yet replace a property part on its own: to change one it rewrites the whole document, so McBasic™ asks your permission for that document with a card.

The item is unavailable while the assistant works or while a Check, Build or Run is running. Outside a procedure the menu shows only the usual text commands. Chapter 16, "Work with the AI assistant", explains the scope in "Add to Chat and the scope".

Open another form

Create a new form instance with `New` and assign it with `Set`:

```
Dim details As frmDetails
Set details = New frmDetails
details.DisplayName = txtName.Text
details.Show
```

`Show`, `Hide`, and `Unload` do different jobs. Learn the form rules before you build an app with many windows.

Handle failures with `Try` and `Catch`

Current database, file, and network samples use `Try ... Catch` around an operation that can fail. This keeps the error close to the action and works well with `Await`:

```
Private Async Sub cmdLoad_Click()
    Dim loaded As Boolean

    Try
        loaded = Await LoadCustomerAsync(txtId.Text)
        If loaded Then
            lblStatus.Caption = "Customer loaded."
        Else
            lblStatus.Caption = "No customer was found."
        End If
    Catch failure As McException
        lblStatus.Caption = "Could not load the customer. " & _
            "Check the ID and try again."
    End Try
End Sub
```

Validate predictable input before the operation. In the `Catch` block, give the user a next step and leave their unsaved input intact. Use the error-handling form documented for your McBasic™ release.

Find a bug with five questions

When code does not work, ask:

1. Did the event run?
2. What value did the input have?
3. Which `If` branch ran?
4. Which property changed?
5. Did I run the project I just checked?

You can print a small message while you investigate:

```
Debug.Print "save:" & saveCount
```

Use the language reference carefully

McBasic™ has its own supported BASIC dialect. Similar-looking syntax from another language is not automatically supported. If a feature is not in the McBasic™ language reference, test it in a small project and read the diagnostic before building more code around it.

Expected result

You should now be able to write a small event procedure that reads one value, checks it, and changes one visible control. Keep that first procedure small. The goal is to prove the event and the data path before adding a database, file, or network call.

Check your work

Create a button and a status label, then use this test handler:

```
Private Sub cmdTest_Click()
    lblStatus.Caption = "The event works"
End Sub
```

Choose **Check**, run the project, and click the button. The label should show the exact message. If it does not, check the control Names before changing the code.

Small challenge

Add a text box named `txtName`. Change the handler so that the label displays `Hello, ` followed by the text in the box. Then add the empty-input check from this chapter.

Tips and tricks

- **Start every file with Option Explicit.** A misspelled name then becomes a Check error instead of a silent new variable.
- **Use one handler for a whole grid of controls.** When you have a control array, write one procedure such as `txtCell_Change(Index As Integer)` and use `Index` to know which cell changed.
- **Show a loaded member last.** `Load` creates it hidden: set `Caption`, `Left` and `Top`, then `Visible = True`. Unload only members that `Load` created.
- **Declare an array without bounds if a procedure resizes it.** An array declared with bounds, for example `Dim a(10)`, has a fixed size and cannot be resized. Write `Dim a() As Long` when you will use `ReDim`.
- **Jump to the code behind a call.** Command–click a procedure name, or press Shift–F2, to open where it is declared, in any file.
- **Drop a dragged control where the pointer is.** In `DragDrop`, move it with `Source.Move X - offset, Y - offset` plus the target's `Left` and `Top`, using half of `Source.Width` and `Source.Height` as the offset so the centre lands under the pointer.
- **Look at the Output pane when a Try … Catch hides a problem.** A friendly message in your app may hide a real bug. When you Run from the IDE, the **Output** pane shows each handled error as "Handled runtime error …" with the file and line.

Chapter 7 — Choose the right component

Components are the controls and services you add to a project. Choose a component because it matches the user's task, not because it has the longest list of properties.

Component groups

The standard catalog includes:

Text and input

`TextBox`, `PasswordBox`, `MaskedTextBox`, `SearchBox`, `NumericBox`, `DatePicker`, and `TimePicker`.

Buttons and choices

`CommandButton`, `CheckBox`, `OptionButton`, `ToggleSwitch`, `ComboBox`, `ListBox`, `ColorPicker`, and `Slider`.

Display and feedback

`Label`, `Badge`, `Image`, `ProgressBar`, `Gauge`, `Sparkline`, and `StatusBar`.

Containers

`Panel`, `Frame`, `Expander`, `TabControl`, `TabPage`, `ScrollContainer`, and `SplitContainer`.

Data and menus

`DataGrid`, `TreeView`, `Menu`, `ContextMenu`, `ToolBar`, `Timer`, and `ValidationProvider`.

The installed release reference describes each component's exact properties and events. This book uses examples instead of a long list of names with no explanation.

The complete standard catalog

The current standard catalog contains 51 entries. This is a useful map when a component appears in the rail but its purpose is not obvious:

Group	Components
Text and input	`TextBox`, `PasswordBox`, `MaskedTextBox`, `SearchBox`, `NumericBox`, `DatePicker`, `TimePicker`, `UpDown`, `MonthView`, `DTPicker`
Buttons and choices	`CommandButton`, `CheckBox`, `OptionButton`, `ToggleSwitch`, `ComboBox`, `ListBox`, `ColorPicker`, `Slider`, `ImageCombo`, `TabStrip`
Display and data	`Label`, `Badge`, `Image`, `ProgressBar`, `Gauge`, `Sparkline`, `StatusBar`, `DataGrid`, `ListView`, `TreeView`, `FlatScrollBar`
Containers	`Panel`, `FlowLayoutPanel`, `Frame`, `Expander`, `TabControl`, `TabPage`, `ScrollContainer`, `SplitContainer`
Commands and support	`Menu`, `ToolBar`, `ContextMenu`, `Timer`, `ValidationProvider`, `CommonDialog`, `ImageList`
Graphics and media	`Line`, `Shape`, `Animation`, `AudioPlayer`, `VideoPlayer`

The catalog's category is a hint, not a programming rule. `CommonDialog`, `Timer`, `Menu`, `ContextMenu`, `ValidationProvider`, and `ImageList` are nonvisual: they appear in the component area and are used from code or by other controls. `AudioPlayer` and `VideoPlayer` are visual controls: place them on the form, size them, and give them a declared source before you run the app.

A component's four parts

When learning a new component, look for four things in this order:

1. **Value or state** — for example `Text`, `Value`, `SelectedIndex`, `IsExpanded`, or `Playing`.
2. **Action** — for example `SetFocus`, `Load`, `Play`, `Pause`, `Stop`, or a dialog method such as `ShowOpen`.
3. **Event** — the moment your code should react, such as `Change`, `Click`, `SelectionChanged`, `Completed`, or `Failed`.
4. **Failure or empty state** — the message, error property, or no-selection case the user must see.

For example, an AudioPlayer normally has `Source`, `AutoPlay`, `Loop`, `Volume`, `Position`, `Duration`, `Playing`, `Paused`, `State`, and `Error`, plus `Loaded`, `Completed`, `Failed`, and `StateChanged`. A VideoPlayer adds `ControlsVisible` and `Stretch` but follows the same basic pattern.

`Source` should be a declared project resource, a direct HTTPS/HLS URL, or a supported public video URL according to the release contract. Do not place a machine-specific file path in a sample and assume another reader has it.

A few controls deserve special handling

CommonDialog

`CommonDialog` is a service component rather than a box on the form. Its parameterless methods are `ShowOpen`, `ShowSave`, `ShowFolder`, `ShowColor`, `ShowFont`, `ShowPrinter`, and `ShowPageSetup`. Set `Title`, `Filter`, or the other matching properties first, call one method, and then read `Cancelled`, `LastAction`, and the selected result.

Line and Shape

Use `Line` for a separator or diagram edge. Its `Angle`, `Direction`, `StrokeThickness`, `LineStyle`, and `LineCap` describe the drawing. Use `Shape` for a filled or outlined geometric surface. `ShapeKind` supports `Rectangle`, `Square`, `Ellipse`, `Circle`, `RoundedRectangle`, and `RoundedSquare`; `Radius` matters for rounded shapes.

TreeView and DataGrid

TreeView is for parent-child structure. Its `Items` collection contains `TreeNode` objects, and `SelectedNode` is the stable selection reference. DataGrid is for rows and columns. Bind a `Recordset` through its documented binding object, keep a stable key column, and use specialized events for cell editing, sorting, and errors. Neither control should use a visible row number as a permanent database identity.

ValidationProvider and Timer

`ValidationProvider` attaches a validation message to a target visual control; use it for field-level guidance rather than a single generic error label. `Timer` is nonvisual and exposes `Interval`, `Enabled`, and `Timer`. Keep the handler short and stop the timer when the repeated work is no longer needed.

An `Interval` of 0 is valid, as in Visual Basic 6: the timer never fires while `Interval` is 0, whatever `Enabled` says. Assigning a positive `Interval` starts it again when `Enabled` is True.

Ask four questions

Before adding a component, ask:

1. What will the user see or do with it?
2. What Name will my code use?
3. Which property holds its value or visible state?
4. Which event tells my code that something important happened?

For a `TextBox`, the value is usually `Text` and the main event is `Change`. For a `CommandButton`, the main event is `Click`. A `Timer` has no box on the form; it sends a `Timer` event while it is enabled.

Example: a simple search box

Add a `SearchBox` named `txtSearch` and a `Label` named `lblResults`. In the search event, show a simple message:

```
Private Sub txtSearch_Search()
    If Len(Trim$(txtSearch.Text)) = 0 Then
        lblResults.Caption = "Type a search term."
        Exit Sub
    End If

    lblResults.Caption = "Searching for: " & _
        Trim$(txtSearch.Text)
End Sub
```

This is not a full-text search feature. It is a small example that shows how the component, its value, its event, and the result fit together.

Shared properties

Many visual components use the same basic properties:

- size and position: `Left`, `Top`, `Width`, `Height`;
- layout: `Anchor`, `Dock`, `Margin`, `Padding`;
- state: `Visible`, `Enabled`, `Opacity`;
- appearance: `Font`, `ForeColor`, `BackColor`;
- keyboard use: `TabIndex`, `TabStop`;
- accessibility: `AccessibleName`, `AccessibleDescription`, `AccessibleRole`,

and `AutomationId`.

Controls that accept files also expose `AllowDrop` and the drag events. Every visual control can be dragged inside the app with `DragMode`, `DragIcon`, the `Drag` method and the `DragDrop` and `DragOver` events (see "Drag a control inside your app" in

chapter 6). Controls with a popup or a value may add events such as `DropDown`, `CalendarOpened`, `DragStarted`, or `SelectionChanged`. Read the event signature created by the IDE instead of guessing from a similarly named control.

Learn the common meaning once. Then read the selected component's description for the details that are different. The Properties area shows it at the bottom for the setting you are editing, and **Categories** keeps the shared settings in the same groups for every component.



Figure 7.1: The component rail

Check your work

- The component has one clear job.
- Its Name says what it does.
- You know its main property and event.
- Its parent and layout are intentional.
- You have checked that the needed members are supported in your release.

Tips and tricks

- **Put a picture on a CommandButton.** Set **Picture** to an image resource of the project and choose a side with **PictureAlignment**. The photo button in the Properties panel lists the images you added.
- **The AI assistant cannot create image files.** Add the image to the project yourself, then ask the assistant to use it by name.
- **Ask the four questions before adding a component.** What does the user do, what does the control show, which event matters, and which properties must be set. The answers usually point to the right component.
- **Read the shared properties once.** Name, Enabled, Visible and the other shared properties work the same on most components, so you only learn them once.
- **Refill a list from its own event with a loading flag.** A ListBox or ComboBox handler (SelectionChanged or Click) may clear and refill its own list. Setting `SelectedIndex` afterwards raises the event again, so guard the handler: set a module-level `mLoading = True` before you refill, `mLoading = False` after, and start the handler with `If mLoading Then Exit Sub`. Without the flag the handler calls itself again and again, as in Visual Basic 6.
- **Pause a Timer with Interval 0.** `tmrClock.Interval = 0` stops the Timer events even while `Enabled` is True; assign the old `Interval` again to resume.

Chapter 8 — Work with more than one form

Many desktop apps use more than one window. The main window might show a list, while a second window edits one item. A details window might show information without allowing changes.

McBasic™ lets you create form instances in code. Keep track of which window you are creating, showing, hiding, or closing.

Create a second form

Suppose the project contains `frmMain` and `frmDetails`. A button on the main form can open the details form like this:

```
Private Sub cmdDetails_Click()
    Dim details As frmDetails
    Set details = New frmDetails
    details.DisplayName = txtName.Text
    details.Show
End Sub
```

Read the four lines as four actions:

1. Make a variable that can hold a `frmDetails` form.
2. Create a new form and put it in that variable.
3. Pass a value to the new form.
4. Show the new form.

The new form is independent. Closing it does not automatically close the main form.

`Show`, `Hide`, and `Unload`

These commands are not synonyms:

Command	Simple meaning
`Load`	create a form and run its `Form_Load`, without showing it
`Show`	make a form visible
`Hide`	make a form invisible but keep it available

Command	Simple meaning
`Unload`	close and release a form instance

Use `Hide` when you expect to show the same window again with its current state. Use `Unload` when the window is finished and should be released.

`Load frmOptions` prepares a form before anyone sees it, as in Visual Basic 6: the form is created and its `Form_Load` runs, but the window stays hidden until `frmOptions.Show`. `Load Me` works inside a form, and `Load f` works on a form variable (`Dim f As frmOptions` or `As Form`). A second `Load` of a form that is already loaded does nothing. After `Unload`, the next `Load` creates a new instance, so `Form_Load` runs again with fresh values.

Pass information to a form

Give the details form a public value or procedure that receives the data. Then set that value before calling `Show`:

```
details.DisplayName = txtName.Text
details.Show
```

Keep the order clear. Set the value first, then show the form. That makes it possible for the details form to display the correct value when it opens.

A procedure in a module can also work on any form. Declare its parameter `As Form` and pass `Me` from the form:

```
' In a standard module
Public Sub MarkBusy(ByVal f As Form, ByVal busy As Boolean)
    f.Enabled = Not busy
    f.Tag = busy
End Sub

' In a form
Private Sub cmdLoad_Click()
    MarkBusy Me, True
End Sub
```

An `As Form` parameter reaches only what every form has: `Caption`, `Left`, `Top`, `Width`, `Height`, `Visible`, `Enabled`, `WindowState`, `Tag`, `Show`, `Hide`, `SetFocus` and `Unload`. To read `details.txtName.Text` or call a `Public` procedure of the form, declare the form's own type instead (`ByVal details As frmDetails`). Check names that fix if you forget.

Close a form from its own button

A Cancel button can close its own form:

```
Private Sub cmdCancel_Click()
    Unload Me
End Sub
```

`Me` means the form that is running the current procedure. This is often safer than using a form name when several instances can exist.

Plan your windows

Before you add controls, decide which window does which job. A Mac app is easier to use when each window has one clear purpose.

Window	Job	How you open it
Main window	shows the list or overview and the commands that act on it	it is the startup form
Dialog	creates or edits one item; the user finishes it with **OK** or **Cancel**	`Await f.ShowDialogAsync(Me)`
Settings window	changes app settings; each change applies at once	`frmSettings.Show`
About window	shows the app name and version	`frmAbout.Show`

A good rule: if a group of fields edits one record and needs **OK** and **Cancel**, it belongs in a dialog, not in the main window. Do not put the list, the editor, the filters and every command on one form.

The examples below come from a small task list: a main window `frmMain` with a DataGrid of tasks and the buttons **Add…**, **Edit…** and **Delete**, and a dialog `frmTaskEditor` with a title, a due date, a **Completed** check box, and **OK** and **Cancel**. The words in bold are the captions of that example, not IDE commands.

Open a dialog and wait for the answer

A dialog is a form that the user must finish before going back to the main window. Open it with `ShowDialogAsync` and wait for it with `Await`:

```
Private Async Sub cmdAdd_Click()
    Dim editor As frmTaskEditor
    Set editor = New frmTaskEditor
    editor.Edit "New Task", "", "", False
    Await editor.ShowDialogAsync(Me)
    If editor.Accepted Then
        ' The user clicked OK: use editor.TaskTitle, editor.TaskDue and
editor.TaskDone.
    End If
End Sub
```

Read it as four steps:

1. Create a new dialog, `editor`.

2. Give the data to the dialog with its `Edit` procedure.
3. Show the dialog and wait. `Me`, the main window, owns the dialog, and the dialog opens centred on it.
4. After the dialog closes, look at `editor.Accepted` and the other values it kept to see what the user decided.

The handler is declared `Private Async Sub`, because only an `Async` procedure can use `Await`. Button clicks, menu clicks and `Form_Shown` can be `Async`. `Form_Load` cannot, so never open a dialog from `Form_Load`. An event handler with parameters cannot be `Async` either: a DataGrid's `CellDoubleClick` has row and column parameters, so keep an **Edit…** button for opening the editor.

The older `Show vbModal` form does not exist in McBasic. Use `Await ShowDialogAsync(Me)` instead. Without an owner, `Await f.ShowDialogAsync()` shows the dialog on its own; it works only while no other form is visible, as at the start of `Sub Main` (see Chapter 9). `Hide` cannot close a dialog: it stops the app with error MCB4403. Close a dialog with `Unload Me`.

Pass information to the dialog

`Set editor = New frmTaskEditor` already runs the dialog's `Form_Initialize` and `Form_Load`. Any value you pass after that line is not there yet when `Form_Load` runs. Give the dialog one `Public Sub` that receives the data and fills its own fields:

```
Public Sub Edit(ByVal windowTitle As String, ByVal title As String, ByVal
due As String, ByVal done As Boolean)
    Me.Caption = windowTitle
    txtTitle.Text = title
    txtDue.Text = due
    chkDone.Value = done
End Sub
```

The main window can also fill a control of the dialog directly, as in `editor.txtTitle.Text = "x"` (see "Use a form's controls from other code" in Chapter 9). A `Public Sub` such as `Edit` is still the better design for a dialog: the dialog decides how its own fields are filled, and the caller passes everything in one call.

Get the answer back

The dialog keeps its answer in `Public` variables, declared at the top of its code:

```
Public Accepted As Boolean
Public TaskTitle As String
Public TaskDue As String
Public TaskDone As Boolean
```

When `Await editor.ShowDialogAsync(Me)` finishes, the dialog has already been unloaded, but, as in Visual Basic 6, its `Public` variables keep their last values.

The main window reads them right after the `Await`. Do not call a `Public Sub` or `Function` of a dialog that has closed, and do not read its controls (`editor.txtTitle.Text`): only its variables are still there, and a closed dialog's control stops the app with error MCB4402. The **OK** button copies the control values into the `Public` variables before `Unload Me`.

For a large answer, you can instead pass in an object of a class module (a `.cls` file with `Public` fields) that the dialog fills, and read that object afterwards.

OK and Cancel

The **OK** button checks the input, copies it into the `Public` variables, marks the answer as accepted and closes the dialog. **Cancel** only closes it:

```
Private Sub cmdOK_Click()
    Dim problem As String
    problem = TaskStore.CheckTask(txtTitle.Text, Trim(txtDue.Text))
    If Len(problem) > 0 Then
        lblError.Caption = problem
        txtTitle.SetFocus
        Exit Sub
    End If
    TaskTitle = Trim(txtTitle.Text)
    TaskDue = Trim(txtDue.Text)
    TaskDone = chkDone.Value
    Accepted = True
    Unload Me
End Sub

Private Sub cmdCancel_Click()
    Unload Me
End Sub
```

Keep `Unload Me` as the last line. If the input is wrong, the procedure shows the message in a red label and leaves the dialog open.

If the user closes the dialog with the window's close button, `Accepted` stays `False`. The main window treats that exactly like **Cancel**.

To make Return click **OK** and Escape click **Cancel**, name the two buttons in the dialog's layout:

```
"actions": { "defaultButtonId": "editor-ok", "cancelButtonId": "editor-
cancel" }
```

The **OK** button is then filled with the accent colour. You can also set `cmdOK.Default = True` and `cmdCancel.Cancel = True` in code. `Default` and `Cancel` are not layout properties: putting them under `"properties"` is a Check error (MCB2301). Inside a multi-line text box, Return still starts a new line.

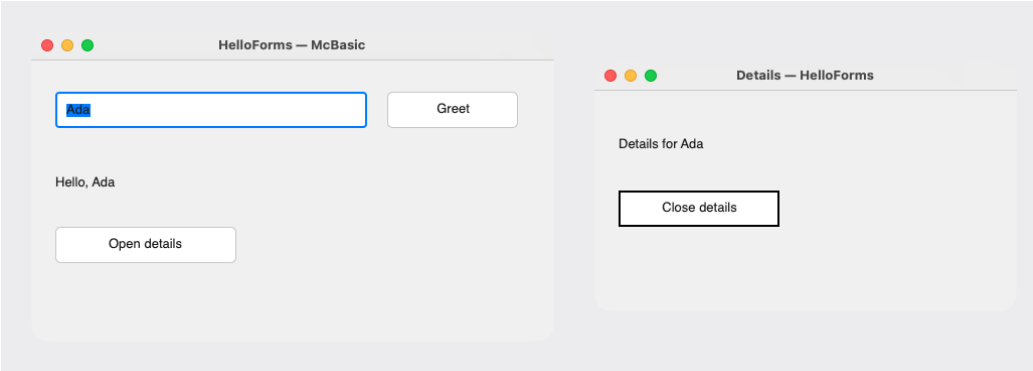


Figure 8.2: Task list and Edit Task dialog

Lay out a dialog the Mac way

These numbers follow Apple's Human Interface Guidelines. All of them are in DIP, the same unit as `Left`, `Top`, `Width` and `Height`:

- Keep 20 between the window edges and the controls, 8 between related controls, and 20 between groups.
- Text boxes, check boxes, option buttons, buttons and single-line labels are 32 high. A single-line text box and a command button centre their text vertically in that height. Do not go below 28: in a lower control the text of a field or a check box is cut. Every control uses the 13-point system font, so leave **Font** empty.
- A single-line label centres its text vertically, so give a field's label the same **Top** and **Height** as the field. Right-align it and end it with a colon: `Due date:`.
- Put **OK** in the bottom-right corner, 20 from both edges, and **Cancel** 8 to its left, both the same width (at least 96).
- Give a dialog a fixed size: **MinWidth** and **MaxWidth** equal to **Width**, **MinHeight** and **MaxHeight** equal to **Height**.
- End a button caption with … when it opens a window that asks for more input: **Add…**, **Edit…**.

Confirm before deleting

A command that destroys data asks first. Make **No** the default answer, so that pressing Return does not delete anything:

```
If MsgBox("Delete the selected task? This cannot be undone.", vbYesNo +
vbExclamation + vbDefaultButton2) = vbYes Then
    ' Delete the task.
End If
```

A **Delete** button is never the default button of a window.

Settings and About windows

A Settings window or an About window can stay open while the user keeps working. Open it with `Show` on the form's name:

```
Private Sub cmdSettings_Click()
    frmSettings.Show
End Sub
```

`frmSettings.Show` always means the same window: a second click brings the open window to the front instead of opening another one.

In a Settings window, apply each change at once. Save it with `Settings.Set` and tell the main window, for example `frmMain.ApplySettings`. A Mac Settings window has no **OK** and **Cancel** buttons, only a **Close** button that is both the default and the cancel button. An About window shows `App.Name`, `App.Version`, `App.Build` and `App.LegalCopyright` and has a **Close** button. Your own About window is optional: the application menu already shows the McBasic™ About box.

What happens when windows close

What happens	Result
a dialog or secondary window closes	only that window closes; the app keeps running
`Form_QueryUnload` sets `Cancel = 1`	the window stays open; a waiting `Await` keeps waiting
the main window closes, or the user chooses Quit	every other window is asked first; if none refuses, all unload and the app quits
a window refuses to close during Quit	the app keeps running and no window closes

To ask "Save changes?" before the app quits, do it in the main window's own `Form_QueryUnload`, which runs first.

The main window is the startup form. An app that starts from `Sub Main` (**Project > Startup Object > Sub Main**, see Chapter 9) has no main window: it keeps running while any form is loaded and quits when the last one unloads.

Keep the work out of the forms

Put the app's data and rules in a module, such as `TaskStore`, and let the forms only move values between their controls and that module. Then you can test the rules with a unit test, which runs without opening a window:

```
Public Sub Test_CheckTask()
    Assert.AreEqual "Enter a title.", TaskStore.CheckTask("   ", "")
```

```
    Assert.AreEqual "", TaskStore.CheckTask("Call Ada", "2026-10-05")
End Sub
```

A UI test can call only the `Public` procedures of a form. It cannot type into a text box or click a button. Do not add extra `Public` procedures to a form just so that a test can drive it. Test the module the form uses instead.

A small two-form exercise

1. Copy the `HelloForms` sample.
2. Open `frmMain` and type `Ada` in the name box.
3. Click **Open details**.
4. Confirm that the details form shows `Ada`.
5. Close the details form.
6. Confirm that the main form is still open.

> **[IMAGE PLACEHOLDER — Figure 8.1: Main and details forms]**
> Show: both windows open, with the same name visible in the details window.
> Purpose: show that the second form is a separate window owned by the app.
> Labels: main form, details form, value passed to the details form.
> Crop: both title bars and the important fields.
> Alt text: “The main form remains open while a separate details form is shown.”

Where a window opens

Select the form (click an empty part of the canvas) and use the **Window** group of the Properties panel.

Property	Values	What it does
`StartUpPosition`	`Manual`, `CenterScreen`, `CenterOwner` (default)	Where the window appears the first time it is shown.
`Left`, `Top`	DIP, default 0	Used by `Manual`: the distance from the top-left corner of the main screen's visible area (below the menu bar). The designer does not move the form.

- `CenterOwner` centres a dialog over the form that opened it, and any other window over the active window. The first form of the app has no owner, so it opens centred on the screen.
- `CenterScreen` centres the window on the screen.

- `Manual` uses `Left` and `Top`. McBasic™ keeps at least 40 points of the title bar on the screen, so a window can never be lost off-screen.
- A form shown again after `Hide` opens where the user left it.
- In code, `Me.StartUpPosition` can be read but not assigned. Assign `Me.Left` and `Me.Top` to move the window at any time. Assigning one of them keeps the other where the window is. Assigned in `Form_Load`, before the window first appears, they decide where it opens, whatever `StartUpPosition` says.
- Positions are measured from, and kept on, the main screen. A window that the user moved to another display moves back to the main screen when code sets `Left` or `Top`.
- An MDI child uses `Left` and `Top` inside the MDI area when it is `Manual`, and is centred there with `CenterScreen`. With `CenterOwner` it cascades like before.

To place a window in code, measure the screen with `Screen`. `Screen.Width` and `Screen.Height` are the main screen's size in points, the same unit as `Left`, `Top`, `Width` and `Height`:

```
Private Sub Form_Load()
    ' Open the window in the upper third of the screen, centred across it
    Me.Left = (Screen.Width - Me.Width) / 2
    Me.Top = (Screen.Height - Me.Height) / 3
End Sub
```

`Screen.Height` includes the menu bar, while `Top` counts from below it, so a window placed this way sits slightly lower than the arithmetic says. VB6 code that divides by `Screen.TwipsPerPixelX` or `Screen.TwipsPerPixelY` keeps working: both are 1. Chapter 21 lists every `Screen` member.

Earlier McBasic™ releases opened the first window at a fixed position near the top-left corner. Forms saved by those releases now open centred, because they use the default `CenterOwner`.

The window frame and its buttons

Property	Values	What it does
`BorderStyle`	`Sizable` (default), `Fixed`, `Dialog`, `None`, `ToolWindow`	The kind of window frame.
`ControlBox`	True (default) / False	The close button.
`MinButton`	True (default) / False	The minimize button, which sends the window to the Dock.
`MaxButton`	True (default) / False	The green zoom button.

- `Sizable` can be resized.
- `Fixed` cannot be resized. macOS cannot zoom such a window, so `MaxButton` is greyed out.
- `Dialog` cannot be resized and has no minimize or zoom button. Use it for dialogs opened with `ShowDialogAsync`.
- `None` has no title bar. The three button properties are ignored and greyed out. Such a window closes only from your code (`Unload Me`) or when the app quits (**Quit**, Cmd-Q), so give it its own close button.
- `ToolWindow` behaves like `Fixed` with only the close button. macOS utility panels are not available to McBasic™ apps yet.
- `MaxButton = False` greys out only the green zoom button. It does not turn off macOS full screen: the window still supports it. McBasic™ apps have no **Enter Full Screen** menu item, so McBasic™ offers the user no other way into full screen.
- `ControlBox = False` disables the close button and refuses any other close the user starts. Your code can still close the form with `Unload Me`, and **Quit** (Cmd-Q) still quits the app after `QueryUnload`. Give the user another way to close such a window, such as a Close button.
- These properties can also be changed in code: `Me.BorderStyle = "Fixed"`.
- In an MDI child they drive the child's own title bar inside the MDI area.

Window state, floating and transparency

Property	Values	What it does
`WindowState`	`Normal` (default), `Minimized`, `Maximized`	`Minimized` puts the window in the Dock; `Maximized` zooms it.
`TopMost`	False (default) / True	Keeps the window above ordinary windows.
`Opacity`	0 to 1 (default 1)	0 is invisible, 1 is opaque.

- `Maximized` is the macOS zoom, not full screen. The user can still choose full screen with the green button; code then reads `Maximized`.
- `Maximized` needs `BorderStyle = Sizable` and `MaxButton = True`.
- `Minimized` needs `BorderStyle Sizable` or `Fixed` and `MinButton = True`.
- Check reports an error for a layout that breaks these rules. Code that breaks them gets error 5.

- A `Maximized` or `Minimized` request that has not reached the window yet, for example one set in `Form_Load` or on a hidden form, makes assigning `BorderStyle`, `MinButton` or `MaxButton` raise error 5 when the new frame would not allow that state.
- A form whose `WindowState` is `Minimized` starts in the Dock; one whose `WindowState` is `Maximized` opens zoomed.
- `WindowState` follows the user: when they minimize the window, `Me.WindowState` reads `Minimized`.
- The constants `vbNormal`, `vbMinimized` and `vbMaximized` hold these values: `Me.WindowState = vbMaximized`.
- `TopMost` is ignored for an MDI child. `Opacity` fades an MDI child inside the MDI area.

Enabled, Visible, Tag, Cursor and AccessibleName

Property	Values	What it does
`Enabled`	True (default) / False	Whether the form's controls accept input.
`Visible`	True / False, code only	Shows or hides the form.
`Tag`	Text in the Properties panel, any value in code	Keeps a value of yours.
`Cursor`	`Default` and the controls' Cursor values	The pointer over the form.
`AccessibleName`	Text, empty by default	The window name for VoiceOver.

- `Enabled = False` makes every control on the form unavailable. The window can still be moved and closed. Use it while a long task runs. The form's Menu commands stay active, so code that a menu command starts should check that the task is not already running.
- A form with `BorderStyle None` has no title bar, so while it is disabled only **Quit** (Cmd-Q) or your code closes it: its own close button is one of the disabled controls.
- `Visible` exists only in code: `Me.Visible = False` is `Me.Hide`, and `Me.Visible = True` shows a hidden form as `Me.Show` does. On a form that is already shown, `Me.Visible = True` does nothing: unlike `Show`, it does not bring the form to the front. The first form of an app is always shown, so the Properties panel has no Visible row. Like `Show` and `Hide`, it cannot be assigned while `Form_Load` runs (MCB4403).

- `Tag` keeps any value of yours. The Properties panel stores text; code can store a number, a date or text (`Me.Tag = 42`).
- `Cursor` is the pointer over the form where a control does not set its own. It takes the same values as the controls' Cursor.
- `Screen.MousePointer` overrides the pointer of every window at once, over the controls too: `Screen.MousePointer = vbHourglass` shows the wait pointer and `Screen.MousePointer = vbDefault` gives each form and control its own `Cursor` back. A window that opens while the override is on gets it too, and so does a control created later, such as a control-array member made by `Load`.
- `Screen.ActiveForm` is the form whose window is active, or `Nothing` when none is (for example while the app is in the background).
- On an MDIForm, `Enabled` and `Cursor` cover the MDIForm's own controls. Each child form has its own.
- `AccessibleName` is the window name VoiceOver reads. Leave it empty to use the Caption. An MDI child has no window of its own, so its AccessibleName names its area inside the MDI window.

A background picture

Set `Picture` to one of the project's image resources. Click the photo button at the end of the row and choose a name, or **None**. The picture is painted over `BackColor`, behind the controls.

PictureSizeMode	Result
`Normal` (default)	Natural size at the top-left corner.
`Stretch`	Fills the form; proportions may change.
`Zoom`	As large as fits, keeping its proportions.
`Center`	Natural size, centred.
`ZoomCrop`	Fills the form, keeping its proportions; edges are cropped.

The designer does not draw the picture; run the app to see it. In code, `Me.Picture = "Logo"` changes it. A name that is not an image resource of the project raises error 481 (Invalid picture).

On an MDIForm, the gray MDI area covers the picture, so it shows only in the strip above the child area (`MDIClientTop`).

Not available on macOS

- `ShowInTaskbar`*: the Mac has no taskbar. Every window appears in Mission*

Control and in the Dock.
- A per-form `Icon`*: macOS windows have no icon. The app icon belongs to the project and appears in the Dock and the Finder: set it with* ***Icon:*** *in* ***Project > Project Settings…*** *(see Chapter 2, "Choose the app icon").*
- `HelpContext`*: there is no context-help file system. Add a Help button or a* ***Help*** *menu item that opens your help.*

Menus

Every window of an app now gets menus without any work from you.

Automatic menus. The menu bar of a form window holds these menus.

- **Edit** has **Undo**, **Redo**, **Cut**, **Copy**, **Paste** and **Select All**. They act on the text field that has the focus and are greyed out when the focus is elsewhere. They carry no shortcuts of their own: the text fields keep handling their usual Command keys, so a list or a grid never loses them.
- **Window** has **Minimize** (Command–M), **Zoom**, **Bring All to Front** and the list of the open windows. In an MDI app the list shows the MDI children. Command–M minimizes the window, unless the form cannot minimize (for example `BorderStyle` is not `Sizable` or `Fixed`, or `MinButton` is False); then the item is greyed out.
- **The application menu** has **About**, which always shows the McBasic™ About box: the app's name, "Version x (Build n)", the copyright line when **Project > Project Settings…** sets one, and "Made with McBasic™". This is how apps made with McBasic™ are credited. A menu item named `mnuAbout` does not replace it: `mnuAbout` is an ordinary menu item that stays in the menu where you put it, runs its own `Click` handler, and can be a menu title or hold items. A menu item named `mnuSettings` moves into the application menu as **Settings…** and keeps its own `Click` handler. Without such an item, **Settings…** stays hidden.

The legacy Menu component. A form's Menu component still works, but its items now appear in the **Dock menu**: right-click the app's icon in the Dock. They no longer appear in the menu bar. The toolbox no longer offers it, and projects that already use it keep working. To move its items into the menu bar, use **Convert to Menu Editor…** (see "Convert an old Menu component" below).

The Menu Editor

The Menu Editor builds the menu bar of one form: menu titles, submenus, keyboard shortcuts, `Checked`, `Enabled` and `Visible` per item, and a `Click` event for each item. Chapter 21 documents the language side, in "Menus and PopupMenu".

To open it, select the form and choose **Design › Menu Editor…** (Command–E). On the Ribbon, use the **Design** tab, group **Menu**, button **Menu Editor**. You can also

double-click the menu strip in the designer (see below). The command is unavailable in a UserControl designer: a UserControl cannot have menus.

The window

The window has a list of the form's menu items on top and a section **Item** below it. Select an item in the list to edit it.

Field	Meaning
Caption	The text the user sees. An `&` before a letter marks the access key, as in VB6 (`&File`). A single `-` makes the item a separator line.
Name	The name your code uses, such as `mnuFileOpen`. Names follow the rules of control names (Chapter 5).
Index	Leave it empty for an ordinary item. A whole number from 0 up makes the item a member of a menu array (see "A menu array" below).
Shortcut	A pop-up with **None** and the 79 VB6 shortcuts. Each entry shows the Mac keys next to it, for example `Ctrl+S  (⌘S)`: Ctrl becomes Command (⌘) on the Mac.
Checked, **Enabled**, **Visible**, **WindowList**	The starting values of the item's check mark, enabled state and visibility. **WindowList** makes the menu title list the open windows.

The buttons work like this.

Button	What it does
← and →	Change the level of the selected item. The items under it move with it.
↑ and ↓	Move the item up or down among the items at the same level, together with its sub-items.
Next	Select the next item.
Insert	Add a new item before the selected one.
Delete	Remove the selected item and everything under it.

The list shows the level of each item with dots (· · · · for each level) and the shortcut at the right. A menu has at most 5 levels.

Problems appear live under the list. **OK** is disabled while the list shows an error. A warning, such as a shortcut that macOS keeps for itself (Command–Q or Command–H), does not block **OK**. **OK** applies all your changes to the form as one step, so a single **Undo** takes them back; **Cancel** discards them.

If you rename an item in the editor, McBasic™ renames its event handlers too (`mnuOld_Click` becomes `mnuNew_Click`) and the references in your code, as it does when you rename a control. Giving an item the name of a menu array, with an **Index**, makes it a member of that array, and its handler gains `Index As Integer`; clearing a member's **Index** and giving it a new name takes it out of the array. Renaming needs a project that passes **Check**, as renaming a control does. If McBasic™ cannot apply a rename, for example because an item that joins an array still has handlers of its own, nothing changes and a message says "The menu was not changed" with the reason: move that code into the array's handler, then try again.

Rules the editor checks

Message (shortened)	What to do
the name is missing, or is not a valid name	Type a **Name** that starts with a letter and has only letters, digits and underscores.
the name is used twice	Give each item its own **Name**. Two items may share a name only as a menu array: they have the same parent and each has an **Index**.
the name is already used by a control	Choose another **Name**; menu items and controls share one set of names.
more than 5 levels	Move the item nearer the menu bar with ←.
a title cannot be a separator	A top-level item needs a real **Caption**, not -. Indent the separator with →.
a separator cannot hold items	Move the items out from under the - line.
only an item under a title can have a shortcut	Set the **Shortcut** to **None**, or move the item under a menu title.
the shortcut is used twice	Choose another **Shortcut** for one of the items.
only a menu title can show the window list, and only one	Keep **WindowList** on one menu title.
`mnuSettings` cannot be a menu title	Indent it under a menu such as **Help**. On the Mac this item moves to the application menu.

Message (shortened)	What to do
the **Caption** is empty (warning)	The item shows nothing. Type a caption or make it a separator.

The menu strip in the designer

When a form has a menu, a strip above the canvas shows its menu titles. The strip is not part of the form's client area, so control coordinates do not change. When the form has no menu, the strip is hidden.

- Click a title to see its drop-down.
- Click an item, or a title that has no items, to open its `Click` handler in the code editor. McBasic™ creates the handler if it is missing, as when you double-click a control.
- Hidden items show in italic grey, with the tooltip "Hidden at run time". Disabled items are dimmed.
- Double-click the strip to open the Menu Editor.
- The strip shows only your own menus. The automatic **Edit**, **Window** and application menus are not previewed.

Menus in the project tree and the code editor

Under each form that has a menu, the project tree shows a **Menu** group with the items nested as in the menu. Double-click an item to open its `Click` handler. Double-click the group to open the Menu Editor. In the code editor, the object pop-up lists the menu items; choose one, then choose its **Click** event.

A menu array

Give several items the same **Name**, the same parent, and different **Index** values to make a menu array, for example `mnuRecent(0)` to `mnuRecent(3)`. They share one handler, `Sub mnuRecent_Click(Index As Integer)`, and you can `Load` and `Unload` members at run time (Chapter 21).

Convert an old Menu component

An old Menu component becomes a real menu with **Convert to Menu Editor…**. The button is in the Inspector of the Menu component, and in the context menu of its button in the non-visual tray under the designer.

- The old Menu, which only showed in the Dock menu, becomes one menu title with the Caption `&Menu` and one item per line. Separator lines stay separators. An `&` in an old line is kept as a literal `&` (written `&&` in the new Caption). A hidden or disabled old Menu gives a hidden or disabled title.

- Its `Click` handler is split into one `mnuX_Click` per item when it is exactly a `Select Case` on `ClickedIndex` with whole-number `Case` labels. In any other case the old handler stays as it is, with a comment line that lists the new items and their old indexes. That happens when there is a comment outside the `Case` bodies, any other code, a call to the handler from elsewhere, or a `Case Else` with code.
- An alert lists the references to the old Menu that remain in your code, so you can update them.
- One **Undo** reverts the whole conversion. Then the Menu Editor opens.

If the conversion cannot be done, the IDE shows an MCB4606 message that names the fix (Chapter 23).

Current limits

- The legacy Menu component is one flat list of commands, and its items now appear in the Dock menu, not in the menu bar (see "Menus" above). Convert it with **Convert to Menu Editor…**.
- A menu has at most 5 levels. A UserControl cannot have menus.
- The designer strip does not preview the automatic menus.
- An event handler with parameters, such as a DataGrid's `CellDoubleClick`, cannot open a dialog with `Await`. Use a button.
- A form cannot use `Show vbModal`, and `Hide` cannot close a dialog (MCB4403). Open dialogs with `Await f.ShowDialogAsync(Me)` and close them with `Unload Me`.
- MDI containers have their own rules; see the MdiForms sample.
- The designer does not draw a form's `Picture`, and does not resize an `AutoSize` Label; run the app to see both.

Check your work

- You can create and show a second form.
- The second form receives its data before it is shown.
- Closing the second form leaves the main form available.
- You know when to use `Hide` and when to use `Unload`.
- Each window has one job: list, dialog, Settings or About.
- Dialogs open from an `Async Sub` with `Await f.ShowDialogAsync(Me)`.
- The dialog returns its answer in `Public` variables that the main window reads after the `Await`.
- Return clicks **OK**, Escape clicks **Cancel**, and closing the window counts as **Cancel**.

- **Delete** asks first, with **No** as the default answer.
- A dialog uses `BorderStyle = Dialog` and opens over its owner.
- A window that must not be closed directly offers its own Close button.
- The app's rules live in a module that has unit tests.

Tips and tricks

- **Hide is not Unload.** `Hide` keeps the form and its values in memory; `Unload` throws them away. Pick `Unload` when the next visit should start from scratch.
- **Load a slow form early.** `Load frmReport` in the main form's `Form_Load` runs the report's own `Form_Load` while nobody waits for it; `frmReport.Show` later opens it at once.
- **Set Height 32 on buttons and fields.** It is the McBasic™ height, and text stays centred. The AI layout check (MCB4790, Chapter 16) flags a button, single-line field, check box, option button or toggle switch below 28.
- **Pass data in before you show the form.** Give the second form a public procedure that receives the data and call it before `Show`, so the form never appears with empty fields.
- **Make No the default when you confirm a deletion.** Then pressing Return by mistake does not delete anything.
- **Do not look for the legacy Menu in the menu bar.** Its commands are in the Dock menu. Use **Convert to Menu Editor…** to move them into the menu bar.
- **Press Command–E to open the Menu Editor.** Or double-click the menu strip above the form.
- **Use a menu array for a Recent Files list.** Make `mnuRecent(0)`, `mnuRecent(1)` and so on, and `Load` more members as files are opened.
- **Put Settings under Help or Edit, and keep your own About where you want it.** Name the item `mnuSettings` and it moves to the application menu on the Mac, where users expect it. `mnuAbout` is an ordinary item: it stays in its menu and runs its `Click` handler, for example to show your own About window, while the application menu keeps the McBasic™ About box.
- **Wrap long work in `Screen.MousePointer = vbHourglass` … `vbDefault`.** Set it back in every path, including the error handler. macOS redraws the pointer between events, so put long work in an `Async Sub` that `Await`s between steps; a loop that never waits can finish before the pointer changes.
- **Share code between forms with `As Form`.** A module procedure that takes `ByVal f As Form` works for every window, as long as it uses only what every form has, such as `Caption` or `Enabled`; for a form's own controls, take that form's type.

- **Use a dialog when the user must answer first.** Open it with `ShowDialogAsync` and wait with `Await`, so the code after it runs only when the dialog has closed.

Chapter 9 — Organize your code

As an app grows, one form can end up holding too many procedures. McBasic™ projects can contain form code, shared modules, and class code. Put each piece where it is easiest to find and reuse.

Form code

Put code that belongs to one form in that form's `.bas` file. Event procedures usually belong here because they work with that form's controls.

Keep this code close to the form it controls. A short procedure such as `cmdClear_Click` should be easy to find when you select the Clear button.

Shared modules

Put general-purpose procedures in a module when more than one form needs them. For example, a name-cleaning or formatting function may belong in `Utilities.bas`.

Shared code should receive values as arguments and return results. When it does need a control of a form, it names the form, as the next section shows.

Use a form's controls from other code

A module, a class or another form reaches a form's controls through the form, as in Visual Basic 6. Write the form's name, a dot, and the control's name:

```
' In a standard module
Public Sub ShowScore(ByVal points As Long)
    frmMain.lblScore.Caption = CStr(points)
    frmMain.cmdReset.Enabled = points > 0
End Sub
```

`frmMain` here is the form's default instance: the window the app opened at start when `frmMain` is the startup form. It follows the same rules as `frmMain.Caption`. If the default instance does not exist yet, the first use creates it without showing it.

Everything a control can do in its own form works the same way from outside: read and change its properties, call its methods, and pass it on.

A form variable reaches the controls of that window:

```
Dim editor As frmTaskEditor
Set editor = New frmTaskEditor
editor.txtTitle.Text = "Write the report"
editor.chkDone.Value = False
```

A control array works through the form too, with its index, its bounds and `For Each`:

```
Dim i As Long
For i = frmMain.txtCell.LBound To frmMain.txtCell.UBound
    frmMain.txtCell(i).Text = ""
Next

Dim c As Control
For Each c In frmMain.txtCell
    c.Enabled = True
Next
```

Components such as a Timer, a CommonDialog or an ImageList work the same way, through their properties and methods:

```
frmMain.tmrClock.Enabled = False
frmMain.dlgOpen.ShowOpen
```

Menu items are controls of the form as well:

```
frmMain.mnuFileSave.Enabled = False
frmMain.mnuRecent(0).Caption = "Report.txt"
```

`With` saves repeating the form's name:

```
With frmMain
    .lblStatus.Caption = "Saved"
    .txtName.SetFocus
End With
```

A control can also be passed to a procedure or kept in a variable of its type. Declare the parameter `ByVal`: a control placed in the designer cannot be replaced, so a `ByRef` parameter (the default) gives the Check error MCB2102.

```
Public Sub Highlight(ByVal box As TextBox)
    box.SelectAll
    box.SetFocus
End Sub

' Elsewhere
Highlight frmMain.txtName
```

Some things stay inside the form that owns the control:

- **A control name alone works only in its own form's code.** In a module, `lblScore.Caption = "0"` is a Check error (MCB2001) that names the fix: write `frmMain.lblScore`.
- **Load and Unload.** Check refuses `Load frmMain.txtCell(5)` and `Unload frmMain.txtCell(5)` outside `frmMain`, and `Unload frmMain.lblScore` anywhere, as for any control placed in the designer. Add a `Public Sub` to `frmMain` that loads the member, and call it.
- **Event procedures.** `txtName_Change` belongs in the code of the form that holds `txtName`.

- **One name, one meaning.** A form cannot declare a `Public` variable, procedure, property or event with the name of one of its controls, because `frmMain.name` would be ambiguous: Check reports MCB2002 and asks you to rename one of them. A `Private` one is allowed: other code cannot see it, and inside the form it hides the control of the same name.
- **A closed form's controls.** Read a dialog's controls before it unloads. After `Unload`, reading `editor.txtTitle.Text` stops the app with error MCB4402; read the dialog's `Public` variables instead (see Chapter 8).

A UserControl keeps its inner controls private. On a form that hosts the UserControl `ucClock1`, `frmMain.ucClock1.Time` works when the UserControl declares a `Public Property Time`. A control inside it, such as `ucClock1.lblTime`, cannot be reached: Check says to use a `Public Property` of the UserControl instead.

When a Public Sub is the better choice. For one value, the qualified name is the shortest route. When several controls change together, add a `Public Sub` to the form, such as `Public Sub ShowScore(ByVal points As Long)`, and call `frmMain.ShowScore 42`. The form stays in charge of its own controls, and the callers stay short.

Start the app from Sub Main

A desktop app normally starts by showing its startup form. As in Visual Basic 6, it can start from a procedure instead: `Sub Main` in a standard module. Use it to show a splash or login window first, to choose which window opens, or to load data before any window appears.

1. Add a module with **Project > Add Module…** if the project has none.
2. In that module, write a `Public Sub Main()` without parameters. Write `Public Async Sub Main()` when `Main` waits for a window with `Await`.
3. Choose **Project > Startup Object > Sub Main**. The menu lists the forms that can open first and **Sub Main**; the current choice has a check mark. An MDI child is not listed, because it needs its MDIForm first.
4. Save the project.

The project still needs at least one form: `Main` decides which one opens.

A splash window

A window only appears on screen when McBasic™ has time to draw it. Code that runs without a pause, such as `frmSplash.Show`, then a long `LoadSettings`, then `Unload frmSplash`, can finish before the splash is ever drawn. Let the splash do the slow work itself and wait for it:

```
' In a standard module, for example modStart
Public Async Sub Main()
    Await frmSplash.ShowDialogAsync()
    frmMain.Show
End Sub

' In frmSplash, with a Timer tmrStart (Interval 100, Enabled True)
Private Sub tmrStart_Timer()
    tmrStart.Enabled = False
    LoadSettings
    Unload Me
End Sub
```

`ShowDialogAsync()` without an owner shows the splash on its own and waits until it unloads. `Main` pauses at the `Await`, so McBasic™ can draw the splash before the Timer runs, and the splash stays visible while `LoadSettings` works. When it unloads, `Main` goes on and shows the main window.

A login window

The VB6 `frmLogin.Show vbModal` becomes an awaited dialog without an owner. Read the dialog's `Public` variables after it closes; they keep their values after `Unload`:

```
Public Async Sub Main()
    Dim login As frmLogin
    Set login = New frmLogin
    Await login.ShowDialogAsync()
    If Not login.Accepted Then Exit Sub
    frmMain.Show
End Sub
```

To stop here, use `Exit Sub`: when `Main` ends and no form is loaded, the app ends. `End` also stops the app at once, without `QueryUnload` or `Unload` (see Chapter 21).

`ShowDialogAsync()` without an owner works only while no other form is visible: show the splash or login before any other window. If another window is already open, the call stops with run-time error MCB4403; pass an owner instead, as in `Await f.ShowDialogAsync(Me)`. `MsgBox` also works in `Main` before any window is open: it shows its alert on its own and `Main` waits for the answer.

`Hide` on a modal dialog is refused (run-time error MCB4403): end the dialog with `Unload Me`, and `ShowDialogAsync()` returns.

When the app ends

McBasic™ calls `Main` before any form exists, so `Main` decides what opens. The app then keeps running while any form is loaded, shown or hidden, and quits when the last one unloads. A form hidden with `Hide` still counts as loaded: with only hidden forms left, the app stays in the Dock with no window until the user chooses Quit. Unload a form when you are done with it. If `Main` ends without loading any form, the app ends at once.

Quit (Command–Q or `App.Quit`) asks every loaded form in `Form_QueryUnload` first, MDI children before their MDIForm; if one refuses, no window closes. `UnloadMode` is 1 for every form during Quit. Once every form has agreed, the app ends: if `Main` is still waiting, for example for a login that Quit just closed, it resumes but cannot load a new form, so `frmMain.Show` after the `Await` opens nothing. Code that an `Await` resumes while the app is already ending, after its last form unloaded, cannot keep it running either.

Check

Check makes sure the project can start: exactly one standard module must declare `Sub Main`, as a `Public Sub` (or a plain `Sub`) or a `Public Async Sub`, without parameters. A `Sub Main` in a form or class does not count, and `Async Sub Main` is refused in a project that starts from a form. If `Sub Main` is missing or declared twice, Check names the fix: add `Public Sub Main()` to a module, or choose a form in **Project > Startup Object**.

To start from a form again, choose that form in **Project > Startup Object**. A module's `Sub Main` is then an ordinary procedure.

A `"ui"` test module needs a startup form (see Chapter 15).

Classes

Use a class when you need an object with its own data and procedures. A class can represent something such as a Customer or an Invoice.

Create a class only when it makes the program easier to understand. A small helper function does not need a class just because classes exist.

The exact class features supported by McBasic™ are part of the language reference. Check that reference before using interfaces, events, inheritance, or advanced object behavior.

When a class starts and ends

A class can run code of its own when an object is created and when it ends, as in Visual Basic 6. Write two Subs without parameters in the class:

```
' In the class clsLog
Option Explicit
Private mOpenedAt As Date

Private Sub Class_Initialize()
    ' Runs once, when New creates the object
    mOpenedAt = Now
End Sub

Private Sub Class_Terminate()
    ' Runs once, after the object is no longer used
    Debug.Print "Log open for " & DateDiff("s", mOpenedAt, Now) & "
```

```
seconds"
End Sub
```

- `Class_Initialize` runs at the end of `New clsLog`, before the object is assigned to your variable. It runs exactly as in VB6. Do not call it yourself after `New`: it would run twice.
- `Class_Terminate` runs **once, but not at a fixed moment**. VB6 runs it as soon as the last variable lets go of the object. McBasic™ cannot know that moment: it runs `Class_Terminate` some time after the object is no longer used, when the memory is reclaimed, or, for every object still alive, when the app ends normally (the last window closes, `App.Quit`, or `Sub Main` returns with no window open).
- `End` stops the app at once, without `Class_Terminate`, as in VB6.
- An error in `Class_Terminate` shows the run-time error dialog, like an error in an event procedure.
- Both may be `Private` or `Public`. With parameters, as a `Function`, or as `Async`, Check reports an error that names the fix (MCB2301).
- In a form or module, a Sub named `Class_Initialize` is an ordinary Sub.

Classes written for McBasic™ before Build 860 may need a change. Earlier releases ran neither Sub on their own, so a class that called its own `Class_Initialize` after `New` now initialises twice: remove that call. A `Class_Initialize` or `Class_Terminate` declared with parameters now fails Check with MCB2301: remove the parameters, or rename the Sub (for example `Setup`) and keep calling it yourself.

A simple rule for moving code

Ask this question:

> *Does this procedure need a control from this form?*

- If yes, keep it in the form or pass the needed value clearly.
- If no, consider moving it to a shared module.
- If it needs its own state and several related operations, consider a class.

Check your work

- Event code is close to the form it controls.
- Shared procedures name the form when they use its controls (`frmMain.lblScore`), and a form with several related changes offers a `Public Sub` for them.
- Names describe the job of each module or class.
- You have checked advanced language features before depending on them.

Tips and tricks

- **Jump between files with Jump to Definition.** Command–click a procedure name, or press Shift–F2. It opens the module or class where the procedure lives, so a project with many files stays easy to explore.
- **Move code that more than one form needs into a module.** If you copy the same lines into two forms, they belong in a shared module. Then there is one place to fix.
- **Use a class when data and its rules belong together.** A customer with a name and a rule for checking it is a good class.
- **Find where a procedure is used before you move it.** Use **Edit > Find References** (Command–Shift–R) so you do not miss a caller.
- **Show a splash window with Sub Main.** Choose **Project > Startup Object > Sub Main**. In `Public Async Sub Main()`, write `Await frmSplash.ShowDialogAsync()` and then `frmMain.Show`; let a Timer on the splash do the slow work and `Unload Me`.
- **Do cleanup that must happen now in a Public Sub you call, not in Class_Terminate.** Give the class a `Public Sub Close()` that releases what it holds, and call it where you are done with the object; keep `Class_Terminate` as a safety net.
- **Name the form to reach its controls from a module.** Write `frmMain.lblStatus.Caption = "Saved"`. Renaming `lblStatus` in the designer also updates `frmMain.lblStatus` in every module and form.

Chapter 10 — Handle input and validation

Good input handling answers three questions:

1. What does the user need to enter?
2. What counts as a valid value?
3. What should the user do when the value is not valid?

> ***WeatherDesk checkpoint***
> *Apply the rules to the same project in* `samples/WeatherDesk.mcbproj`*: an empty city*
> *must stop before either HTTP call, and an empty selected place must stop before the SQLite insert. The status label should say what to correct.*

Validate before saving

Start with the input. Trim extra spaces, check the value, and show a useful message. If the value is not valid, move the focus back to the field and stop the procedure.

The pattern is:

```
name = Trim$(txtName.Text)
If Len(name) = 0 Then
    lblStatus.Caption = "Enter a name."
    txtName.SetFocus
    txtName.SelectAll
    Exit Sub
End If
```

The user learns what is wrong and where to fix it. The code does not continue until the required value exists.

React when text changes

Use a TextBox Change event when feedback should appear as the user types. For example, show `A name is required` while the field is empty and `Name entered` after the user types something.

Do not use Change for expensive work on every keystroke. Wait for a button or a dedicated search event when the operation may take time.

Focus and selection

`SetFocus` moves the logical focus to a control. `SelectAll` selects its text. Together they help the user correct an error quickly.

Every control that can take keyboard focus (every component with a **TabStop** property: text boxes, buttons, check and option buttons, lists, pickers, the grid, sliders, tabs and the toolbar) has `SetFocus`. Call it as a statement, `txtName.SetFocus`, or with `Call txtName.SetFocus()`. It needs a shown form and a visible, enabled control whose **TabStop** is `True`; otherwise it stops with error MCB4408. `SelectAll` belongs to `TextBox` and `MaskedTextBox`.

Use these methods with a clear message. A focused field is helpful, but the user should still know why it needs attention.

Read-only input

Use `ReadOnly` when the user may read a value but must not change it. It affects the running control. It is not the same as locking a control in the designer.

Keyboard actions

For a form with a primary and a secondary button, assign a default button and a cancel button when the release supports those form actions. Test Return and Escape with each important input control. A multiline text box may use Return for a new line instead of the form's default action.

Practice exercise

Use the Customer Entry sample and test these cases:

Action	Expected result
Click Save with an empty name	A message asks for a name.
Type a name and click Save	The status message contains the name.
Turn on ReadOnly	The user cannot edit the field.
Press Escape	The Cancel action runs when the form supports it.

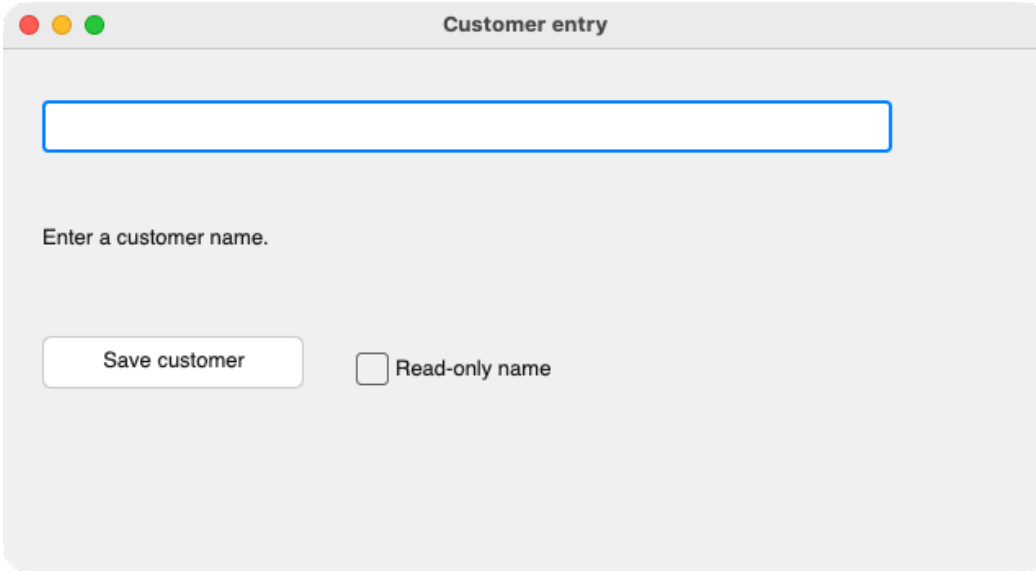


Figure 10.1: Validation feedback

Check your work

- Empty and valid input both have a tested result.
- The message says what the user should do next.
- Focus moves to the field that needs correction.
- Read-only fields cannot be changed by accident.
- Mouse and keyboard paths produce the same intended result.

Tips and tricks

- **Validate first, then act.** Trim the text, check it, and stop with a clear message before the slow part of the procedure starts.
- **Send the user back to the problem.** After a message, call `SetFocus` on the field that needs correction and `SelectAll` to select its text.
- **Use ReadOnly to show a value that must not be edited.** It is for the running app. It is not the same as locking the control in the designer.
- **Test Return and Escape.** Press both in every important field. A multiline TextBox may need Return for a new line.

Chapter 11 — A small McBasic™ language toolkit

You do not need the whole language at once. The ideas in this chapter cover many everyday apps; use the full language reference when you need an exact rule or a list of limits.

> ***WeatherDesk checkpoint***
> *The same project now uses the language features in this chapter: local and form-level variables, a helper function,* `If`*, a loop over returned JSON results, and a small error path. Work on* `samples/WeatherDesk.mcbproj` *so each*
> *exercise changes the application you have already built.*

The BASIC habits that prevent most bugs

McBasic™ code is easiest to understand when every value has a clear type, every name has a clear scope, and every procedure has one job.

Choose a type deliberately

Type	Good first use
`Boolean`	a yes/no state such as `mBusy`
`String`	user text, labels, paths, and messages
`Integer`	a small whole-number index or count
`Long`	a larger whole-number count, database ID, or row number
`Single` / `Double`	measured or calculated values with a fractional part
`Decimal` / `Currency`	values that need a decimal or money-oriented type in a supported API
`Date`	a calendar date or time value
`Variant`	a value whose shape is supplied by a supported API, such as JSON or a data row

Type	Good first use
object reference	a form, control, database, recordset, or service object

Do not use `String` for every value just because text is easy to display. If a value is a number, keep it numeric until the moment you show it. If it is a date, keep it a `Date` until you format it for a label.

```
Dim quantity As Long
Dim unitPrice As Double
Dim total As Double

quantity = 3
unitPrice = 4.5
total = quantity * unitPrice
lblTotal.Caption = CStr(total)
```

`CStr`, `CInt`, `CLng`, `CDbl`, `CDate`, and the other conversion functions make a conversion visible. Validate text before converting it; do not assume that a blank box is a valid number.

McBasic™ never turns text into a number by itself. `quantity = txtQuantity.Text` is a Check error; write `quantity = CLng(txtQuantity.Text)`. Text that is not a plain number raises error 13 and a value that does not fit raises error 6, so convert inside `Try` or check the text first. The same applies to functions that take numbers, such as `DateSerial`: `DateSerial(CLng(Mid(s, 1, 4)), CLng(Mid(s, 6, 2)), CLng(Mid(s, 9, 2)))`. The language reference appendix lists every conversion.

Declare first, then assign: `Dim total As Long`, then `total = 5` on its own line. McBasic™ has no initialisers (`Dim total As Long = 5`), no `As New` (write `Dim items As Collection`, then `Set items = New Collection`) and no type suffixes (`name$`). Check names the fix for each of them.

Keep scope small

Use `Dim` inside a procedure for temporary work. Use a `Private` form or module field for state that must survive between events. Use `Const` for a named value that does not change. `Option Explicit` at the top of a module makes a misspelled name a Check error instead of a hidden new variable.

```
Option Explicit

Private mBusy As Boolean
Private Const MaxSearchLength As Long = 120

Private Sub cmdSearch_Click()
    Dim query As String
    query = Trim$(txtSearch.Text)
    If Len(query) > MaxSearchLength Then
        lblStatus.Caption = "Use 120 characters or fewer."
```

```
        Exit Sub
    End If
End Sub
```

Be explicit about object references

Use `Set` for an object and `Nothing` for an empty reference. Use `Is` when you want to compare object identity, not the text or value displayed by an object.

```
Dim editor As frmDetails
Set editor = New frmDetails
If editor Is Nothing Then Exit Sub
editor.Show
```

The reference must remain valid while you use it. A control or form that has been unloaded cannot be used as if it were still open.

Use arrays and collections for groups of values

An array has numeric positions and a declared shape. A `Collection` is an ordered group that can grow. Array bounds are worth reading with `LBound` and `UBound`; collection positions are one-based.

```
Dim names As Collection
Dim item As Variant

Set names = New Collection
names.Add "Ada"
names.Add "Grace"

For Each item In names
    Debug.Print CStr(item)
Next item
```

Use `ReDim` and `ReDim Preserve` only when a dynamic array is really the right shape. A database result, JSON array, or live control collection should be handled through its documented object rather than copied into an unrelated array without a reason.

Optional parameters, named arguments, and `ParamArray`

A procedure can let callers leave out its last arguments. Declare those parameters `Optional` and give each one a default value:

```
Private Function Greeting(ByVal name As String, _
        Optional ByVal word As String = "Hello", _
        Optional ByVal excited As Boolean = False) As String
    Greeting = word & ", " & name
    If excited Then Greeting = Greeting & "!"
End Function
```

`Greeting("Ada")` returns `Hello, Ada`, and `Greeting("Ada", "Welcome")` returns `Welcome, Ada`. To pass a later argument without the one before it, leave its position empty, as in `Greeting("Ada", , True)`, or name it, as in `Greeting("Ada", excited:=True)`. Both return `Hello, Ada!`.

When a procedure accepts any number of values, end its parameter list with a `ParamArray`. The values arrive as a `Variant` array that starts at 0:

```
Private Function Total(ParamArray values() As Variant) As Double
    Dim item As Variant
    Dim sum As Double
    For Each item In values
        sum = sum + item
    Next item
    Total = sum
End Function
```

`Total(4, 5.5, 10)` returns `19.5`, and `Total()` returns `0`.

Check enforces a few limits. Required parameters come first. A typed `Optional` parameter needs a constant default; only an `Optional` parameter `As Variant` can leave it out, and then `IsMissing` tells whether the caller passed it. A `ParamArray` is always the last parameter, and its procedure takes neither `Optional` parameters nor named arguments. `Async` procedures accept neither `Optional` nor `ParamArray` parameters. The language reference appendix lists every rule.

Conditions

Use `If` when a value changes what the app should do. Keep each branch clear. If a condition becomes hard to read, move part of the decision into a function with a clear name.

A decision with more than one rule

Write each rule so that a reader can tell what makes the action available. In this example, a customer can be saved only when the name is present and the form is not already saving:

```
Private Function CanSave( _
    ByVal name As String, _
    ByVal busy As Boolean) As Boolean
    If busy Then
        CanSave = False
        Exit Function
    End If

    If Len(Trim$(name)) = 0 Then
        CanSave = False
        Exit Function
    End If

    CanSave = True
End Function

Private Sub UpdateSaveButton()
    cmdSave.Enabled = CanSave(txtName.Text, mBusy)
End Sub
```

The early exits make the two reasons for `False` visible. Test both invalid cases separately, then test a name such as `Ada` while `mBusy` is `False`.

Choose between several values

`Select Case` reads well when the same value has several possible meanings:

```
Private Function PriorityLabel(ByVal priority As Long) As String
    Select Case priority
    Case 1
        PriorityLabel = "High"
    Case 2
        PriorityLabel = "Normal"
    Case 3
        PriorityLabel = "Low"
    Case Else
        PriorityLabel = "Unknown"
    End Select
End Function
```

Keep the `Case Else` branch when data may come from a file or database. It gives the user a sensible result when a future record contains a value you did not expect.

Loops

Use a loop when the same action must happen more than once. Always make it clear how the loop ends. A loop that never ends can make an app look frozen.

When learning loops, start with a small fixed count and print the value you are checking. Then test an empty case and a one-item case.

`For` for a known range

Use `For` when you know the counter range. The `Step` is optional; it defaults to one:

```
Private Function TotalRange( _
    ByVal firstValue As Long, _
    ByVal lastValue As Long) As Long
    Dim total As Long
    Dim value As Long

    For value = firstValue To lastValue
        total = total + value
    Next value

    TotalRange = total
End Function
```

`TotalRange(1, 4)` returns `10`. Also test `TotalRange(4, 1)` and decide what your application should do before you use a range supplied by a user.

`Do` when the end depends on the work

Use a `Do` loop when the number of attempts is not known in advance. Give the loop a visible exit condition and, for a retry, set a maximum:

```
Private Function ReadFirstNonEmpty( _
    ByVal firstText As String, _
    ByVal secondText As String) As String
    Dim attempt As Long
    Dim value As String
```

```
    Do
        attempt = attempt + 1
        If attempt = 1 Then
            value = Trim$(firstText)
        Else
            value = Trim$(secondText)
        End If

        If Len(value) > 0 Then Exit Do
        If attempt = 2 Then Exit Do
    Loop

    ReadFirstNonEmpty = value
End Function
```

The limit matters. A loop that waits forever for valid input makes the window look frozen. When the input comes from a control, it is often clearer to stop the procedure and ask the user to try again instead of looping inside the event.

`For Each` for a collection

Use `For Each` when the job is “do this for every item” and the position does not matter:

```
Private Function JoinNames(ByVal names As Collection) As String
    Dim item As Variant
    Dim result As String

    For Each item In names
        If Len(result) > 0 Then result = result & ", "
        result = result & CStr(item)
    Next item

    JoinNames = result
End Function
```

For a `Collection`, positions start at 1. If the code must update a specific row or use an index, use `For` and `Item(index)` instead.

Arrays and collections

An array is a fixed group of values addressed by number. A `Collection` is an ordered group that can grow or shrink. Choose by asking whether the number of items is part of the design:

```
Private Function AverageScore() As Double
    Dim scores(0 To 2) As Long
    Dim index As Long
    Dim total As Long

    scores(0) = 80
    scores(1) = 75
    scores(2) = 90

    For index = LBound(scores) To UBound(scores)
        total = total + scores(index)
    Next index
```

```
    AverageScore = total / 3
End Function
```

Use `LBound` and `UBound` instead of assuming that an array starts at zero. That makes the procedure safer when a module uses `Option Base 1` or when a lower bound is declared explicitly.

For data that arrives one item at a time, a collection is often clearer:

```
Private Sub ShowNames()
    Dim names As Collection
    Dim name As Variant

    Set names = New Collection
    names.Add "Ada"
    names.Add "Grace"

    For Each name In names
        Debug.Print CStr(name)
    Next name
End Sub
```

The output is `Ada` followed by `Grace`. Do not use a collection as a hidden database: if the values must survive when the app closes, save them in a file or database.

Dates and times

Use `Date` for a date or time value. Capture the value once when several lines must refer to the same moment.

`Format(d, mask)` accepts only these Date masks: `yyyy-MM-dd`, `dd/MM/yyyy`, `MM/dd/yyyy`, `HH:mm:ss`, `yyyy-MM-dd HH:mm:ss`, `dd/MM/yyyy HH:mm:ss` and `MM/dd/yyyy HH:mm:ss`. Build any other layout from parts, for example `MonthName(Month(d)) & " " & CStr(Day(d))`. Number masks use only `0`, `#`, `,`, `.` and a final `%`: `"0"`, `"0.00"`, `"#,##0"`, `"#,##0.00"` and `"0%"` are typical. The language reference appendix shows what each mask prints.

Built-in constants

The VB6 constants are built in, with their VB6 values, so a confirmation reads the way it does in VB6:

```
Private Sub cmdDelete_Click()
    Dim answer As Integer

    answer = MsgBox("Delete this customer?", _
        vbYesNo + vbQuestion + vbDefaultButton2, "Delete")
    If answer = vbYes Then
        lblStatus.Caption = "Deleted." & vbCrLf & "Choose Undo to restore
it."
    End If
End Sub
```

`vbDefaultButton2` makes **No** the button that Return chooses, which protects the user from deleting by accident. `vbCrLf`, `vbTab` and the other text constants are

`String`; the `MsgBox`, key and comparison constants are `Long`. `KeyDown` code compares the key with `vbKeyReturn`, `vbKeyEscape` or another key constant. Colors are the exception to VB6: a McBasic™ color is `"#RRGGBB"` text, so `vbRed` and `RGB(255, 0, 0)` are `String` and assign directly to `ForeColor` or `BackColor`. The language reference appendix lists every constant with its value.

Arrays and collections

An array is useful when values belong together and you use numeric positions. A collection is useful when you need an ordered group that can grow or shrink.

Choose the structure that makes the data easier to read. Do not use an array only because another language example used one.

Error handling with `Try` and `Catch`

An error message is useful only when the program can tell the user what to do next. Check predictable problems before they happen. For a file, database, or network operation that can fail while it runs, keep `Try ... Catch` close to that operation:

```
Private Async Sub cmdLoad_Click()
    Dim loaded As Boolean

    Try
        loaded = Await LoadCustomerAsync(txtId.Text)

        If Not loaded Then
            lblStatus.Caption = "No customer was found."
        Else
            lblStatus.Caption = "Customer loaded."
        End If
    Catch failure As McException
        lblStatus.Caption = "Could not load the customer. " & _
            "Check the ID and try again."
    End Try
End Sub
```

The `Await` keeps the form responsive while the operation is in progress. It is allowed only inside an `Async` procedure, which is why the handler is `Private Async Sub`. In a real form, disable the Load button while it runs and enable it again in both the success and failure paths. Do not hide an error by continuing with empty data.

The current practical samples use `Try` and `Catch` for these service calls. Do not copy an error-handling form from another BASIC dialect unless it appears in the McBasic™ language reference for your release.

Worked example: choose a message

The following function keeps a decision separate from the button that uses it:

```
Private Function CustomerMessage(ByVal name As String) As String
    If Len(Trim$(name)) = 0 Then
        CustomerMessage = "Enter a customer name."
    Else
        CustomerMessage = "Ready for " & Trim$(name)
    End If
End Function

Private Sub cmdCheck_Click()
    lblStatus.Caption = CustomerMessage(txtName.Text)
End Sub
```

Try it with an empty box, `Ada`, and `  Ada  `. The first input should ask for a name. The other two should produce the same clean result.

Expected result

You now have a working example for conditions, loops, arrays, collections, dates, and failure messages. Each one has a result you can observe.

Check your work

Change one input at a time and write down the expected result before you run the project. If the result is surprising, reduce the example until only the condition or loop remains.

Small challenge

Extend `CustomerMessage` so that a very long name produces a clear message. Choose the length limit deliberately, document it beside the code, and test the value just below and just above the limit.

A safe learning method

When you want to try a new language feature:

1. create a small copy of the project;
2. write the smallest example that uses the feature;
3. choose **Check**;
4. run one visible test;
5. keep the feature only if the result is clear.

This keeps an uncertain language experiment out of a large application.

Tips and tricks

- **Choose the type on purpose.** Write `Long` for whole numbers, `String` for text, and so on. Check can then catch a mix-up for you.

- **Use a fixed array only for a fixed number of values.** If the number of values changes while the app runs, declare it without bounds (`Dim a() As Long`) and use `ReDim`.
- **Prefer a Collection for a list that grows.** You add to it without resizing anything.
- **Use Try … Catch around work that can fail.** In the IDE, a handled error still shows in the **Output** pane as "Handled runtime error …", so you can see it even when the app hides it.
- **Try a new idea in a small project.** An experiment in a copy cannot break the application you rely on.

Chapter 12 — Work with SQLite data

SQLite is a small database stored in a file. It suits many desktop apps because the user does not need to install a separate database server.

> ***WeatherDesk checkpoint***
> *Add the* `saved_locations` *and* `forecast_snapshots` *migration from* `samples/WeatherDesk.mcbproj/Database/migrations`*. Open the per-user database, query*
> *the saved rows, bind them to the grid, and test the empty database before saving the first location.*

Plan the data first

Before writing database code, write down:

- what one row represents;
- which fields are required;
- which field identifies a row;
- what the app should do when two users change the same record.

For WeatherDesk, one row contains an ID, a user label, a city, a country, coordinates, a timezone, a tracking flag, and save/update times. A second table records the current temperature, wind speed, and weather code returned by the API.

Open a per-user database

Use a path owned by the application, such as `AppPaths.DataFile("weatherdesk.db")`. Do not write user data into the application bundle. A bundle is the installed program; user data needs a safe writable location.

The WeatherDesk sample opens SQLite when the form is shown, applies a migration, and then loads the rows. The screen stays clear about what is happening by showing messages such as “Opening the local weather database...” and “4 saved location(s)”.

`Set mDb = Await Sqlite.OpenAsync(path)` opens the file and creates it, with its folder, when it does not exist yet. Keep one `Database` per file in a module-level

variable such as `Private mDb As Database` and reuse it. Each `Database` runs its operations one after another on its own background worker, never on the window's thread. `Sqlite.OpenReadOnlyAsync` opens a file that the app must not change.

Close the database with `Await mDb.CloseAsync()` and `Set mDb = Nothing`, for example before you delete the file. New database files use SQLite's WAL journal, so `weatherdesk.db-wal` and `weatherdesk.db-shm` can appear next to the database; close it first, then delete all three files. Foreign keys are enforced.

Use migrations

A migration is a small, numbered database change. It creates the tables the app needs and lets a later version add changes in a known order.

Keep migrations in source control. Do not silently rebuild a user's database when the app starts.

Each migration is a `.sql` file inside the project, declared in `project.mcproj` with a number and a path:

```
"migrations": [
  { "id": 1, "path": "Database/migrations/0001_initial.sql" },
  { "id": 2, "path": "Database/migrations/0002_visits.sql" }
]
```

- Only a `DesktopApp` project can declare migrations; other kinds fail Check with MCB2301.
- `id` is a unique positive whole number. `path` is a project-relative `.sql` file; `Database/migrations/0001_name.sql` is the usual convention.
- A project declares at most 64 migrations. Each file must exist, must not be empty, and can be at most 256 KiB. One file may contain several statements separated by semicolons.
- The build copies the SQL into the app. Nothing is read from the project folder when the app runs.

The current SQLite path is deliberately small and readable:

```
Private Async Function OpenWeatherDatabase() As Boolean
    Dim version As Long

    Try
        Set mDb = Await Sqlite.OpenAsync( _
            AppPaths.DataFile("weatherdesk.db"))
        version = Await mDb.ApplyMigrationsAsync(App.Migrations)
        OpenWeatherDatabase = True
    Catch failure As McException
        lblStatus.Caption = "Database error " & _
            CStr(failure.Number) & ": " & failure.Message
        OpenWeatherDatabase = False
    End Try
End Function
```

`App.Migrations` returns the declared migrations in `id` order. `ApplyMigrationsAsync` runs, in one transaction, every migration that the database has not recorded yet, records each one with its SHA-256 checksum in the table `__mcbasic_migrations`, and returns the highest applied `id` (0 when there is none). Running it again changes nothing. If any statement fails, nothing is applied.

Never edit a migration after it has run on any database. If the SQL of an applied migration changes, `ApplyMigrationsAsync` stops with error 17 and applies nothing. Add a new migration instead, for example `ALTER TABLE saved_locations ADD COLUMN note TEXT NULL`.

`Sqlite.CreateMigrationSet()` and `migrations.Add id, sql` build a set in code; the numbers must increase (error 5 otherwise). Use `App.Migrations` in the app so the schema has one source of truth, and a set built in code in tests (see “Test database code” below).

When the AI assistant proposes a migration, the change adds the `.sql` file and the manifest entry, and waits for your separate approval in the card **Approve database migration files separately?**. The card shows `Migration <id>: <path>` and the SQL, and states that approval adds the files only: “It does not execute a migration or access a database.” The migration runs later, when the app calls `ApplyMigrationsAsync`. Like the rest of an applied stage, the `.sql` file is written to disk when you save the project; later stages of the same plan already see it, and undoing the stage before you save removes the file and its manifest entry together.

Use parameters

Put user values in parameters instead of joining them into SQL text. This keeps the query easier to read and avoids treating user text as SQL instructions.

Parameters are named: `@name`, `:name`, or `$name`. Positional `?` placeholders are not supported and fail with error 17. Every name in the SQL needs exactly one `AddParameter`; a missing value raises error 5 when the command runs, and a name that the SQL does not use, or a second value for the same name, raises error 5 at once.

Pass `Null` for a missing value. `Empty` is stored as empty text, not as NULL. Booleans are stored as 1 and 0, dates as text, and `Currency` and `Decimal` values as exact text. `AddTypedParameter "@price", value, DbType.Currency` also checks the value against a `DbType`.

A common sequence is:

1. create a command with SQL containing names such as `@name`;
2. add a parameter for each name;
3. await the database operation;

4. show a success or error message;
5. refresh the screen when the stored data changes.

Here is the complete shape of a parameterized query and a grid refresh. The `BindingSource` keeps the returned rows available to selection code:

```
Private Async Function RefreshBooks() As Boolean
    Dim command As DbCommand
    Dim rows As Recordset

    Try
        Set command = mDb.CreateCommand( _
            "SELECT ID, Title, Author FROM Books " & _
            "WHERE Title LIKE @needle ORDER BY Title")
        command.AddParameter "@needle", "%" & txtSearch.Text & "%"
        Set rows = Await command.QueryAsync()
        Set mRows = rows.Bind()
        Set grdBooks.DataSource = mRows
        grdBooks.SelectedIndex = -1
        lblStatus.Caption = CStr(mRows.RowCount) & " book(s) loaded"
        RefreshBooks = True
    Catch failure As McException
        grdBooks.ErrorText = failure.Message
        lblStatus.Caption = "Could not load books."
        RefreshBooks = False
    End Try
End Function
```

The SQL has a named parameter, not text joined from the user's input. The grid has a deliberate empty state and the previous rows are not quietly replaced by an error message. If you need pages rather than all rows, `QueryPageAsync` can return a page and a `HasMore` value; see the `DataGridPaging` sample.

`ExecuteAsync` runs `INSERT`, `UPDATE`, `DELETE`, or `CREATE` and returns the number of changed rows as a `LongLong`; declare the receiving variable `LongLong` or `Variant`. `ScalarAsync` returns the first column of the first row, or `Null` when there is no row. Use it for `SELECT COUNT(*) ...`, or for `SELECT last_insert_rowid()` right after an `INSERT` to get the new row's ID.

Read query results

A `Recordset` is a complete copy of the result in memory. Rows and columns are numbered from zero. There is no `EOF`, `MoveNext`, `Fields`, or `RecordCount`; loop over the row numbers and read values by column name:

```
Private Function ListTitles(ByVal rows As Recordset) As String
    Dim index As Long
    Dim result As String

    For index = 0 To rows.RowCount - 1
        If Not IsNull(rows.Value(index, "Title")) Then
            result = result & CStr(rows.Value(index, "Title")) &
Environment.NewLine
        End If
    Next
```

```
    ListTitles = result
End Function
```

- `rows.Value(index, "Title")` and `rows.Rows(index).Value("Title")` return a `Variant`. Column names ignore case.
- `rows.ColumnCount` and `rows.ColumnName(i)` describe the columns. To read a column's declared type, assign `Set column = rows.Columns(i)` and use `column.Name` and `column.StoreType`.
- `INTEGER` values arrive as `Long` (`LongLong` when larger), `REAL` as `Double`, and `TEXT` as `String`. Columns declared `BOOLEAN`, `DATE` or `DATETIME`, `DECIMAL` or `NUMERIC`, and `CURRENCY` arrive as `Boolean`, `Date`, `Decimal`, and `Currency`.
- SQL NULL arrives as `Null`. Test it with `IsNull` before calling `CStr` or `CLng`; there is no `Nz`.
- A row number outside the result raises error 9, an unknown column name raises error 5, and a BLOB value raises error 13: BLOB columns cannot be read from McBasic™ code.

Change several rows together

Use a transaction when changes must succeed or fail together. While a transaction is open, create every command with `tx.CreateCommand`:

```
Private Async Function SaveTwoBooks(ByVal first As String, ByVal second
As String) As Boolean
    Dim tx As DbTransaction
    Dim command As DbCommand
    Dim affected As LongLong
    Dim failed As Boolean

    Try
        Set tx = Await mDb.BeginTransactionAsync()
        Set command = tx.CreateCommand("INSERT INTO Books(Title)
VALUES(@title)")
        command.AddParameter "@title", first
        affected = Await command.ExecuteAsync()
        Set command = tx.CreateCommand("INSERT INTO Books(Title)
VALUES(@title)")
        command.AddParameter "@title", second
        affected = Await command.ExecuteAsync()
        Await tx.CommitAsync()
        SaveTwoBooks = True
    Catch failure As McException
        lblStatus.Caption = "Not saved: " & failure.Message
        failed = True
    End Try
    If failed Then
        If Not tx Is Nothing Then Await tx.RollbackAsync()
        SaveTwoBooks = False
    End If
End Function
```

- While the transaction is open, the database accepts nothing else: a command from `mDb.CreateCommand`, or a second `BeginTransactionAsync`, raises error 17.

- A failed statement does not end the transaction. Call `RollbackAsync` to undo everything since `BeginTransactionAsync`; calling `CommitAsync` after a failure would keep the statements that succeeded.
- The first `CommitAsync` or `RollbackAsync` ends the transaction; later calls do nothing. `tx.Dispose`, or closing the database, rolls back a transaction that is still open.
- A single `ExecuteAsync` outside a transaction is already all-or-nothing.

Keep the window responsive

Database operations may take time. Use the asynchronous McBasic™ database APIs so the window can still show progress and respond correctly. Disable buttons that would start the same operation twice. Re-enable them when the operation finishes, including after an error.

The rules for asynchronous code are short:

- Every call whose name ends in `Async` needs `Await` and parentheses: `Await tx.CommitAsync()`.
- `Await` is allowed in an `Async Function`, and in an `Async Sub` that handles an event without parameters, such as `Form_Shown` or `cmdLoad_Click`. An `Async Function` may return a `Database` or a `Recordset`.
- Write `ByVal` on every parameter of an `Async Function`: `Private Async Function FindBooks(ByVal title As String) As Recordset`. Parameters are `ByRef` unless you write `ByVal`, and `ByRef`, `Optional`, `ParamArray` and array parameters are not allowed in async code.
- `Await` stands alone or is the whole right side of an assignment: `Set rows = Await command.QueryAsync()`. Split `CStr(Await ...)` into two lines.
- Async procedures handle errors with `Try`, `Catch`, and `Finally`; `On Error` is not allowed there. `Await` works inside `Catch` and `Finally`.

Set the grid's `Loading` property to `True` before a long query and back to `False` in `Finally`, so an error never leaves the window looking busy.

Show data in a grid

The usual flow is:

1. run a query;
2. bind the returned rows to a `DataGrid` or another suitable control;
3. clear or update the selection when the rows change;
4. let the user choose a row;

5. use the selected row to fill an edit area.

Do not assume that a visible row number is the database ID. Keep the stable ID with the row.

In the designer, set the grid's `KeyColumn` to the unique ID column and `Columns` to the visible columns in display order, for example `Title,Author`. The key column stays hidden, and an empty `Columns` shows every column. Bind a result with `Set mRows = rows.Bind()` and `Set grdBooks.DataSource = mRows`.

A column header shows the column name unless you give it a caption: write `name|Header` in `Columns`, for example `status|Status,title|Title,due_date|Due date`. The header shows the caption, while binding, `KeyColumn`, `FilterColumn` and `SaveAsync` keep using the real column name. Prefer this to renaming columns with an SQL alias such as `SELECT due_date AS "Due date"`: `SaveAsync` writes edited cells back by column name, so an edit in an aliased column cannot be saved.

- `SelectedKey` returns the `KeyColumn` value of the selected row, or `Null` when no row is selected. Assigning a key to `SelectedKey` selects that row. Use the key, not the visible row number, as the row's identity.
- To refresh, run the query again, call `Bind` again, and assign `DataSource` again. `BindingSource.Refresh` only discards unsaved edits; it does not read the database.
- `EmptyText`, `LoadingText`, and `ErrorText` give the empty, loading, and error states. An error stays visible until your code clears `ErrorText`.

If the user edits a grid cell, use the DataGrid events to validate the draft and the binding object's save operation to write it. With `IsReadOnly` set to `False`, edits stay in the `BindingSource` until your code calls `value = Await mRows.SaveAsync(mDb, "Books", "ID")`, which writes all edits, new rows (`AddNew`), and deletions (`Delete`) in one transaction and returns the number of rows written as a `Long`. Assign the result to a `Long`, or call it as a statement. The key column of an existing row cannot be edited. A row can be sorted, filtered, or paged without changing its database identity. Keep `KeyColumn` or an equivalent stable key in the design and show a clear message for `DataError`.

For readability, set `AlternatingRowBackColor` (for example `#F4F7FB`), choose `GridLines`, keep the first visible column in place with `FrozenColumns = 1` (the hidden key column does not count), and turn on `ShowRowNumbers` for long lists.

Handle database errors

Every database failure raises an `McException`. Show `failure.Message`, keep the user's input, and roll back an open transaction.

- **17** — any SQLite failure: a UNIQUE or NOT NULL constraint, a syntax error, a missing table, a locked or busy database, a `?` placeholder, a command used while a transaction holds the database, or an edited migration.
- **5** — an invalid request: empty SQL, a missing, unused, or duplicated parameter, migration numbers that do not increase, a transaction, migration, or save on a read-only database, an unknown column name, or invalid paging.
- **9** — a row or column number outside the result.
- **13** — a BLOB value, an unsupported parameter value, or text that does not fit a typed column. **6** — a typed parameter outside its range.

Export rows to CSV

`Await rows.ExportCsvAsync(path)` writes a `Recordset` to a new UTF-8 CSV file with a header row, CRLF line endings, quotes only where needed, and an empty field for `Null`. The path must be absolute and end in `.csv`. The export is written completely or not at all, and it never replaces a file: an existing file raises error 58, a missing folder error 76, and a folder without write access error 70. Ask for the path with `Dialogs.SaveFile`, or delete the old file with `File.Delete` first when you mean to replace it.

Test database code

Automated tests (Chapter 15) do not load the project's declared migrations: `App.Migrations` is empty during a test run. Build the schema in the test with `Sqlite.CreateMigrationSet()` and the same SQL. Keep each test database under its own folder, such as `AppPaths.DataFile("tests/Books/books.db")`, close it with `CloseAsync`, and then delete the file and its `-wal` and `-shm` files.

What is not supported

- ADO and DAO objects, connection strings, and server databases: McBasic™ works with local SQLite files through `Sqlite.OpenAsync`.
- `EOF`, `MoveNext`, `Fields`, `RecordCount`, and `For Each` over a `Recordset`: loop from 0 to `RowCount - 1` and read values by column name.
- Running SQL directly on the `Database`: create a command with `mDb.CreateCommand(sql)`, then call `ExecuteAsync`, `ScalarAsync`, or `QueryAsync`.
- Positional `?` parameters: use named parameters.
- Reading BLOB columns.
- `Nz`: test values with `IsNull`.

The complete WeatherDesk CRUD cycle

WeatherDesk makes each database operation visible in the user interface:

1. **Create** — search for a city, choose a label, and choose whether to track it. **Add location** writes the location and its first weather snapshot.
2. **Read** — the grid loads from SQLite at startup and after every change. **Refresh list** runs the read query again.
3. **Update** — select a saved row, change its label or **Track this location**, then choose **Update**. The row keeps its ID and coordinates.
4. **Delete** — select a saved row and choose **Delete** twice. The app removes its snapshots first, then removes the location row.

The `tracked` flag is ordinary application data. It gives the project a useful next step: a later version can refresh weather for tracked locations on a timer while leaving untracked locations untouched. The current sample already lets the user store and edit that decision safely.

A safe WeatherDesk workflow

Build and test these cases in order:

- open an empty database;
- search for a known city and review the JSON-backed result;
- add one location;
- select it, change its label, and update it;
- turn tracking off, refresh the list, and confirm the flag remains off;
- delete the selected location and confirm that it disappears;
- close and reopen the app;
- search with an empty query;
- try a search with the network disconnected;
- try an existing file or a locked database;
- close the app while an operation is running.

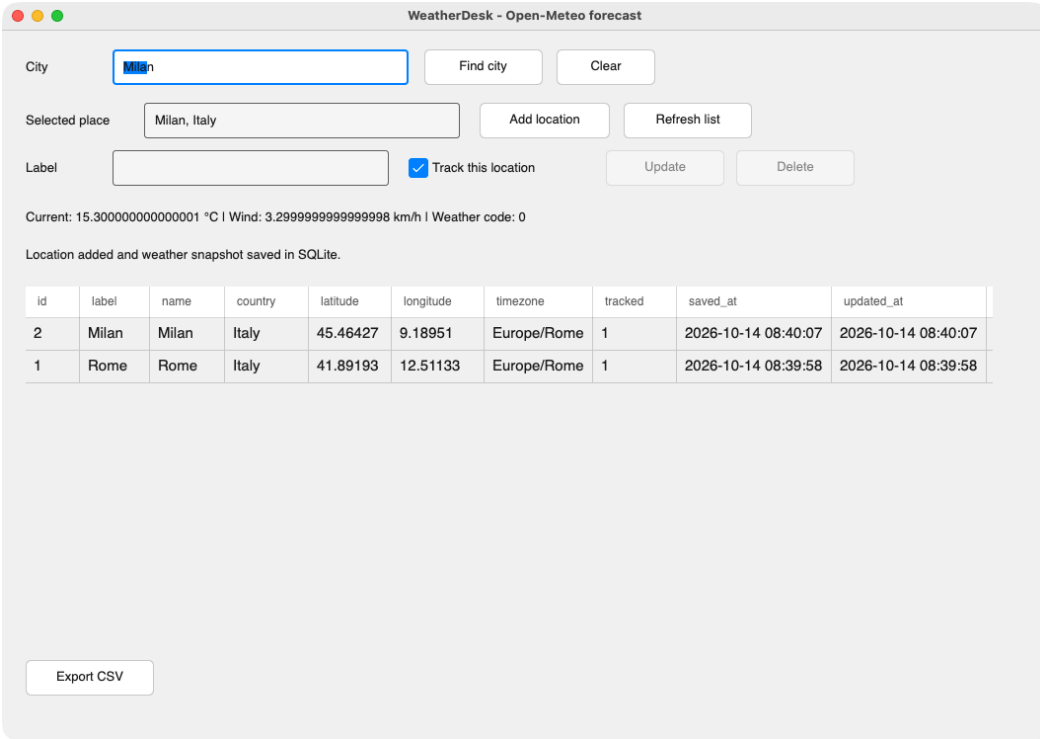


Figure 12.1: WeatherDesk data workflow

Check your work

- User data is stored outside the application bundle.
- Queries use named parameters for user values.
- Changes that belong together run in one transaction, with a rollback on error.
- Every schema change is a new migration; applied migrations are never edited.
- Long operations do not freeze the interface.
- Save, refresh, reopen, and delete have all been tested.
- The app reports errors without losing the user's unsaved input.

Tips and tricks

- **Store the database in a user data path.** Use `AppPaths.DataFile("weatherdesk.db")` and never write data inside the application itself.
- **Always use parameters for user text.** `@name` in the SQL and a value in the command keeps user text from being read as SQL.

- **Disable the button while a query runs.** Re-enable it when the operation ends, so a double click does not run the same query twice.
- **Show the failure message, then keep the user's input.** Every database failure gives you a message to show, and the user does not have to type again.

Chapter 13 — Files, network calls, and reports

Once an app can store data, it may also read a file, download a resource, or create a report. For each operation, decide what the user sees when it starts, while it is running, when it succeeds, and when it is cancelled or fails.

> ***WeatherDesk checkpoint***
> *This is where the same project calls Open-Meteo. Search for a city, use the returned coordinates to request current weather, parse the JSON body, and save the selected location and forecast snapshot in SQLite. The working code belongs in* `samples/WeatherDesk.mcbproj/Forms/frmWeather.bas`*.*

Work with files safely

Choose a user-data path with the McBasic™ application path helpers. Keep the source file and the generated output separate. If an export should not replace an existing file, check for it and tell the user what to do next.

For example, WeatherDesk writes `weatherdesk-locations.csv` and refuses to silently overwrite an earlier export. The user must rename or remove the old file first.

Keep two paths in your head: `App.ResourcePath` is for files shipped with the application, while `AppPaths.DataFile("name")` is for data created or changed by the user. A path that works on the development Mac is not automatically a valid path in a published app.

Read and write files

The `File` service reads and writes whole files. Text is always UTF-8; a file that is not valid UTF-8 raises error 5.

- `File.ReadText(path)` returns the whole file as a `String`.
- `File.WriteText path, text` creates the file, or replaces everything in it.
- `File.AppendText path, text` adds text at the end and creates the file if needed.
- `File.ReadBytes(path)` and `File.WriteBytes path, data` do the same with a `Byte` array.

- `File.Exists(path)` returns `True` when the file exists.
- `File.Delete path` deletes the file; a missing file raises error 53.

A write needs an existing folder (error 76) and is refused inside the application bundle (error 70); other failures raise error 75. `AppPaths.DataFile` creates its folders for you, and there is no `MkDir`.

The `Path` service works on path text only and never touches the disk: `Path.Combine`, `Path.GetFileName`, `Path.GetDirectoryName`, `Path.GetExtension`, `Path.GetFileNameWithoutExtension`, `Path.ChangeExtension`, `Path.GetFullPath`, and `Path.IsAbsolute`. To let the user choose a file, call `Dialogs.OpenFile`, `Dialogs.SaveFile`, or `Dialogs.SelectFolder`; an empty result means that the user chose Cancel.

McBasic™ does not have the VB6 file statements `Open ... For Output`, `Print #`, `Input #`, `Line Input #`, `Close #`, `FreeFile`, `Kill`, `Name ... As`, `FileCopy`, `Dir`, or `MkDir`, and `File` has no `Move`, `Copy`, or `Rename`. Read the whole text and split it with `Split(text, Environment.NewLine)` instead of reading line by line: `Split` returns a `String()` array, so declare `Dim lines() As String` and assign `lines = Split(text, Environment.NewLine)`. Use `File.Delete` instead of `Kill`, and copy a file with `File.WriteBytes target, File.ReadBytes(source)`.

A local variable named `File`, `Path`, `Json`, or `Settings` hides that service inside its procedure. Name such variables `target`, `fileName`, or `text`.

```
Private Sub cmdOpen_Click()
    Dim chosen As String
    chosen = Dialogs.OpenFile("Open notes")
    If Len(chosen) = 0 Then Exit Sub
    Try
        txtNotes.Text = File.ReadText(chosen)
    Catch failure As McException
        lblStatus.Caption = "Error " & CStr(failure.Number) & ": " &
failure.Message
    End Try
End Sub
```

Store per-user data and settings

`App.DataPath` is the application's own folder for user data (`~/Library/Application Support/<bundle identifier>` for a built app). `App.CachePath` is for files that can be recreated, and `App.LogPath` is the file written by `Log.Info`, `Log.Warning`, and `Log.Error`. `AppPaths.DataFile("folder/name.json")` returns a full path inside `App.DataPath` and creates the folders. It accepts only a relative path, without `.` or `..` parts.

For small values, use `Settings`:

```
Private Sub Form_Load()
    txtName.Text = CStr(Settings.Get("customer.lastName", ""))
End Sub

Private Sub cmdSave_Click()
    Settings.Set "customer.lastName", txtName.Text
End Sub
```

`Settings` keeps `String`, `Boolean`, `Byte`, `Integer`, `Long`, `LongLong`, `Single`, `Double`, `Currency`, `Decimal`, and `Date` values, under keys of 1 to 256 characters, in `settings.json` inside `App.DataPath`. `Settings.Get` returns the default value when the key is missing. Every `Settings.Set` rewrites the file atomically, so an interrupted save cannot damage it. To keep a list, store it as JSON text in a `String`.

There is no `GetSetting`, `SaveSetting`, or `Environ`. `Environment.Get(name, defaultValue)` reads only the build values `CI`, `MCBASIC_CONFIGURATION`, `MCBASIC_TARGET`, `MCBASIC_RUNTIME_IDENTIFIER`, and `MCBASIC_BUILD_ID`, and `Environment.NewLine` is the line separator.

Save a document safely

There is no rename, move or copy call, so McBasic™ code cannot replace a file in one atomic step, and `File.WriteText` overwrites its target in place. Choose where to save in this order:

1. small state, such as preferences or a short list saved as JSON text: `Settings.Set`, which the runtime saves atomically;
2. records: SQLite inside a transaction (Chapter 12);
3. a document file: write a pending copy, check it, overwrite the real file, then delete the pending copy.

With the third pattern, an interruption always leaves one complete file: the original while the pending copy is written, or the pending copy while the real file is overwritten. On load, recover a valid pending copy first.

```
Private Const DocumentName As String = "todo/todos.json"

Public Sub SaveDocument(ByVal text As String)
    Dim target As String
    Dim pending As String
    Dim parsed As JsonValue
    target = AppPaths.DataFile(DocumentName)
    pending = target & ".pending"
    File.WriteText pending, text
    parsed = Json.Parse(File.ReadText(pending))
    File.WriteText target, text
    File.Delete pending
End Sub

Public Function LoadDocument() As String
    Dim target As String
    Dim pending As String
    Dim parsed As JsonValue
    target = AppPaths.DataFile(DocumentName)
```

```
    pending = target & ".pending"
    If File.Exists(pending) Then
        Try
            parsed = Json.Parse(File.ReadText(pending))
            File.WriteText target, File.ReadText(pending)
        Catch damaged As McException
            Log.Warning "Discarded an incomplete pending save."
        End Try
        File.Delete pending
    End If
    If File.Exists(target) Then
        LoadDocument = File.ReadText(target)
    Else
        LoadDocument = "[]"
    End If
End Function
```

`Json.Parse` raises error 5 when the pending copy is incomplete, so the real file is overwritten only after the copy is known to be good. To build the JSON text, join it with `&` and escape every string with `Json.Stringify(Json.FromValue(text))`; `Json.FromValue` accepts one `String`, `Boolean`, or number, and there is no serializer for arrays or objects. `Recordset.ExportCsvAsync` already writes safely and refuses to replace an existing file.

Download a file

The HttpDownload sample has a URL, a destination, a Download button, a Cancel button, and a progress bar. Its useful lessons are:

1. disable Download while a transfer is active;
2. show received and total bytes when available;
3. enable Cancel only while there is something to cancel;
4. remove a partial file after an error or cancellation;
5. wait for the operation to finish before closing the form.

Do not test downloads with a file you cannot replace. Use a disposable local file or a test server.

Call a real public API

The main API example is Open-Meteo, also listed by the Free APIs directory. It is a good learning API because the reader can try it without creating an account or copying an API key. It has two calls that naturally form one application:

1. Geocoding turns a city name

into coordinates;

2. Forecast turns coordinates into weather

JSON.

The free endpoint is for non-commercial use, is rate-limited, and has no uptime guarantee. Those limits matter: this is a hands-on learning and personal-use sample, not permission to run an unlimited commercial service. Read the provider's terms before publishing an app.

The first request looks like this:

```
https://geocoding-api.open-
meteo.com/v1/search?name=Singapore&count=5&language=en&format=json
```

In McBasic™, the `WeatherDesk` sample sends it asynchronously:

```
Dim url As String
Dim responseText As String
Dim envelope As JsonValue
Dim document As JsonValue
Dim results As JsonValue

url = "https://geocoding-api.open-meteo.com/v1/search?name="
url = url & Replace(Trim$(txtCity.Text), " ", "+")
url = url & "&count=5&language=en&format=json"
responseText = Await Http.RequestAsync( _
    "GET", url, _
    "{""User-Agent"":""McBasic WeatherDesk""}", _
    "", 10000)
envelope = Json.Parse(responseText)
document = Json.Parse(CStr(Json.Value(Json.Get(envelope, "body"))))

If Not Json.Has(document, "results") Then
    lblStatus.Caption = "The location response is missing results."
    Exit Sub
End If
results = Json.Get(document, "results")
If Json.Count(results) = 0 Then
    lblStatus.Caption = "No city matched the search."
    Exit Sub
End If
```

The forecast request uses values from the first response. This is the useful moment to explain data flow: the second URL cannot be built until the first call has supplied latitude and longitude.

```
Dim forecastUrl As String
Dim current As JsonValue

forecastUrl = "https://api.open-meteo.com/v1/forecast?latitude="
forecastUrl = forecastUrl & CStr(mSelectedLatitude)
forecastUrl = forecastUrl & "&longitude=" & CStr(mSelectedLongitude)
forecastUrl = forecastUrl &
"&current=temperature_2m,wind_speed_10m,weather_code"
forecastUrl = forecastUrl & "&timezone=auto"
responseText = Await Http.RequestAsync("GET", forecastUrl, "", "", 10000)
envelope = Json.Parse(responseText)
document = Json.Parse(CStr(Json.Value(Json.Get(envelope, "body"))))
current = Json.Get(document, "current")
lblCurrent.Caption = "Temperature: " & CStr(Json.Value(Json.Get(current,
"temperature_2m")))
```

`RequestAsync` returns a response envelope. The useful API JSON is inside its `body` field, so the example parses twice: once for the envelope and once for the

provider response. In the full sample, the code checks every required section before reading it and shows a separate message for an empty result, network failure, invalid JSON, and missing fields.

Make the database part visible

Do not save every search automatically. The user first reviews the selected place and current weather, then presses **Add location**. The sample writes the place to `saved_locations`, writes the fetched values to `forecast_snapshots`, refreshes the DataGrid from SQLite, and shows a message that says exactly what happened.

```
Set command = mDb.CreateCommand( _
    "INSERT INTO saved_locations(name, country, latitude, longitude) " & _
    "VALUES(@name, @country, @latitude, @longitude)")
command.AddParameter "@name", mSelectedName
command.AddParameter "@country", mSelectedCountry
command.AddParameter "@latitude", mSelectedLatitude
command.AddParameter "@longitude", mSelectedLongitude
Await command.ExecuteAsync()
Await RefreshLocations()
```

This is the boundary to point out to readers: the API is remote and temporary; the SQLite row is local and durable. Restart the app during the tutorial to prove the difference.

Treat the response as untrusted input

An HTTP success only tells you that a server answered. It does not promise that the answer has the JSON shape you expected. Check the envelope, parse the body, check `Json.Kind`, and use `Json.Has` before reading optional values. Keep the search button disabled while the request is in flight, and give the user a separate message for:

- no network connection or a request timeout;
- a server response that is not valid JSON;
- a valid response with no matching location;
- a forecast response that is missing `current` or a requested value.

Do not put an API key, password, or personal token into the sample. The Open-Meteo free endpoint does not require one for this educational workflow.

Export a PDF report

The ReportDemo sample uses the same rows for preview, PDF export, and printing. That is important: the user should see the same report that the app will save.

Test long text, accented text, page breaks, repeated headings, and the last page total. Decide whether the PDF must contain searchable text before choosing the export path.

The ReportDemo sample uses the same report definition for preview, export, and printing:

```
Private Sub cmdPreview_Click()
    Reports.Preview App.ResourcePath & "/Reports/Customers.report.json"
End Sub

Private Sub cmdExport_Click()
    Reports.ExportPdf _
        App.ResourcePath & "/Reports/Customers.report.json", _
        AppPaths.DataFile("customers.pdf")
    lblStatus.Caption = "PDF saved."
End Sub
```

Use a user-data destination for the generated file. If the report contains private information, say so in the UI and do not place it in a shared project folder by accident.

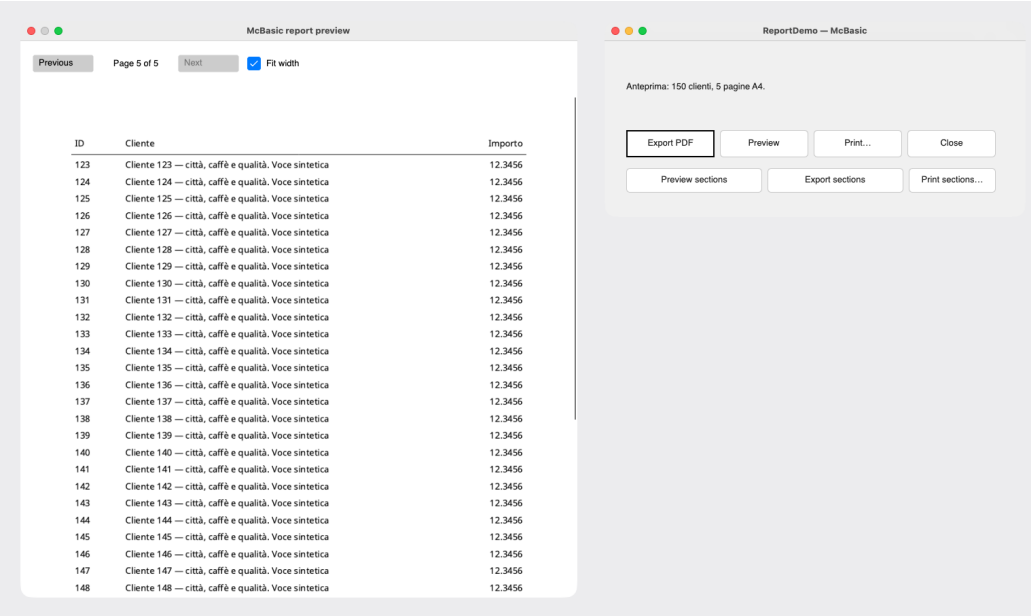


Figure 13.1: Report preview and export

Check your work

- A successful operation creates the expected file.
- A failed or cancelled operation leaves no misleading partial file.
- Existing files are not overwritten by accident.
- User data lives under `App.DataPath`, never inside the application.
- The user sees progress or a clear waiting message.
- The app can stop or close safely after the operation finishes.

Tips and tricks

- **Check whether a file exists before an export.** Tell the user what to do next instead of replacing a file silently.
- **Disable Download while a transfer is active.** Add a **Cancel** button and a progress bar so the user can see and stop the transfer.
- **Treat an answer from the internet as untrusted.** Check that the value you need is really there before you use it.
- **Preview and PDF should use the same rows.** The user then sees on screen what will be saved. Test long text, accented text and the last page.

Chapter 14 — Reuse components and offline packages

Reusable components let you share a control or a library between projects. A package is more than a DLL in a folder: it needs an identity, a version, a target, trusted contents, and a clear public interface.

Two kinds of reusable package

McBasic™ projects can produce two useful kinds of reusable package:

- a **Control Plugin** supplies a reusable visual control and its public

properties/events;

- a **Library Plugin** supplies reusable BASIC-callable procedures and types

without adding a window to the consuming app.

Start with a small package. A reusable component should have:

A reusable control should have:

- a clear name and purpose;
- a stable set of public properties;
- events with simple meanings;
- a checked example project;
- instructions for the project that uses it.

Build the control separately from the app that consumes it. This makes it clear which files belong to the component and which belong to the application.

What an offline package does

An offline package supplies exact local inputs to a project. The lock file identifies the package version and content hash. The local cache supplies the matching archive.

McBasic™ checks that these values agree before Check, Build, Run, or Publish. It does not silently download a different version.

Install once, reference from each project

Use **Settings > Plugins** to install a `.mbpkg` once in the McBasic™ environment. The plugin manager shows the installed identity, version, publisher, licence and capabilities. Installation does not automatically add the plugin to every app.

For each app that needs it, use **Project > Plugin References** and select the installed version. McBasic™ records only the exact reference and hash in the project and its lock file. The package bytes remain in the shared environment store, so using one plugin from ten projects does not create ten copies.

The current Mac implementation exposes this core flow as **Install Component Package** and **Add Component Package**. **Project > Install Component Package…** validates and stores an archive once even when no project is open: with only the Welcome window showing, choose it from the **Project** menu, the file chooser opens as its own window, and an alert confirms the installation. **Add Component Package** lists installed exact versions, previews the selected package, and writes the current project's reference and lock without copying the archive. The complete **Settings > Plugins** inventory with update/removal and the final per-project reference management screen are still narrower than the package format itself. Treat those screens as release-dependent and keep the exact reference in the project under source control.

Trust and development packages

An unsigned local package is a development input, not a release-ready product. The development workflow may require both Developer Mode and an explicit acknowledgement that the local unsigned package is trusted.

Read the warning. Do not enable these choices for a package you have not identified and inspected.

Adding a package to a project does not make it trusted by magic. McBasic™ checks the package identity, version, target, declared files, and content hash. A changed archive must fail the check instead of being silently accepted. An unsigned local package is useful for development, but signing, notarization, and distribution approval are separate release tasks.

Package preview is not package execution

The IDE can read package metadata and show a preview without running package code inside the IDE. This separation protects the editor from a package that is broken or untrusted.

The command-line equivalent for package admission is:

```
mcbasic package verify MyControl.mbpkg \
  --target osx-arm64 \
```

```
  --developer-mode true \
  --acknowledge-unsigned true
```

It checks package structure, declared API version, files, target, capabilities and trust without loading the plugin. A pass means the package is statically admissible. Full SDK certification also requires the recorded runtime, generated-app, cleanup, reproducibility and host-compatibility checks.

A Library Plugin has no application window to run by itself. Test its public procedures through a small consumer project. A Control Plugin can be tested by adding its control to a form and checking the generated app. This distinction keeps “the package was admitted” separate from “the package works for a user.”

Build the WeatherBadge exercise

The WeatherDesk tutorial includes a small visual component named `WeatherBadge`. It displays a place, temperature, and wind speed. Its Refresh button raises a `RefreshRequested` event; it does not know how to call an API or open a database. That boundary is intentional:

- WeatherDesk owns Open-Meteo, JSON parsing, and SQLite.
- WeatherBadge owns its labels, button, public properties, and event.

Create the component in a separate **UserControlLibrary** project, give it the public properties `LocationName`, `TemperatureC`, `WindSpeed`, and `WeatherCode`, and raise `RefreshRequested` when the button is clicked. Check the package, install it once, then add the exact version to WeatherDesk through **Project > Plugin References** (shown as **Add Component Package** in the current Mac workflow). The complete source is in the checked-in [`WeatherBadge` project].

The full step-by-step exercise, including the consuming form code, is in Chapter 18.

A safe package exercise

Use the checked-in `MoneyToolsConsumer` and `CustomerCardConsumer` examples:

1. use the exact lock file;
2. select an absolute local package cache;
3. install or select the exact archive;
4. review the trust settings;
5. choose **Check**;
6. build or run only after Check succeeds;
7. change one package byte and confirm that the next operation rejects it.

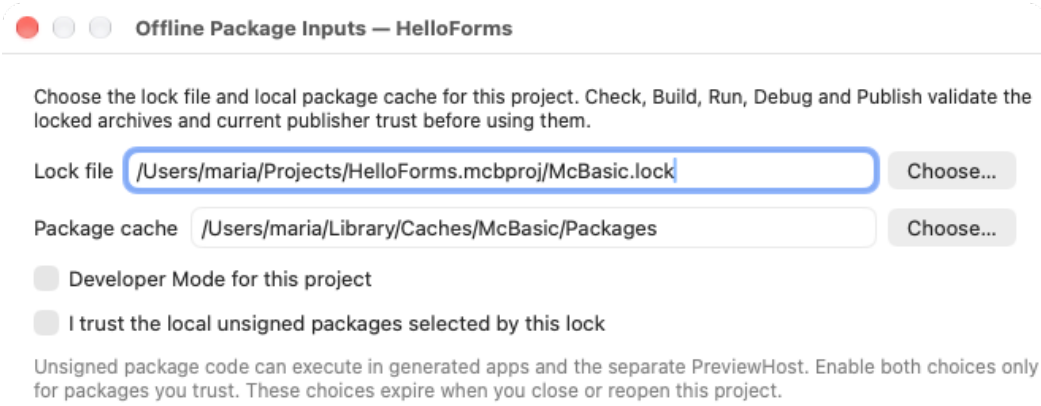


Figure 14.1: Offline package inputs

Check your work

- The package identity and version are clear.
- The lock file and archive match.
- The package is not loaded into the IDE just to preview it.
- Unsigned development trust is separate from release signing.
- A changed or missing package stops the operation instead of replacing an earlier result.

Tips and tricks

- **Install a package once, reference it in each project.** Use **Settings > Plugins** to install a `.mbpkg`. Installing does not add it to every app.
- **A preview is not a run.** McBasic™ reads the package information without running its code, so a broken package cannot harm the editor.
- **Treat an unsigned local package as a development input.** It is fine for your own testing, but it is not ready to give to customers.
- **Keep a component simple.** WeatherBadge only shows values and raises an event. The form does the API call, which keeps the component reusable.

Chapter 15 — Check, test, and debug

A project is not finished just because it opens. Try the actions a user will take, including the mistakes they might make, and check that the app explains what to do next.

> ***WeatherDesk checkpoint***
> *Run the same test sequence against* `samples/WeatherDesk.mcbproj`*: empty city,*
> *successful geocoding and forecast, malformed or incomplete response, network failure, save,*
> *reopen, and export. The catalog must remain unchanged when the remote call fails.*

Use three different checks

Check the project

Choose **Check** after editing. This finds problems in code, forms, names, properties, and event connections. Check also compiles the test modules and shows their errors with the test file and line; Build and Publish never include them.

McBasic™ also checks by itself, in the background, without saving anything: when a project opens, after you save, after the AI assistant applies or undoes a change, and about a second after you stop typing. Problems appear in the Diagnostics list, the status line shows the first one, and the code editor underlines each problem with a dotted line, red for errors and orange for warnings; hold the pointer over it to read "MCBxxxx: message". When the problems are gone the status line says "BASIC check succeeded." The automatic Check waits while your app runs or while another command works, and choosing Check yourself always runs at once. Turn it off with Settings… > Code > Check automatically ("Checks the BASIC code when a project opens, after you save, after an AI change and when you pause typing."); Restore Defaults turns it on again. Underlines appear in the document open in the editor.

Test the running app

Choose **Run** and use the app like a customer. Type real test values, press buttons, use the keyboard, close windows, and reopen saved data.

When you Run or Debug from the IDE, McBasic™ also reports errors that your own code catches with Try … Catch, On Error GoTo or On Error Resume Next. The app

keeps running as you wrote it, and the Output pane shows one line per handled error, for example "Handled runtime error 10 in SudokuGenerator.bas line 42: Array is fixed: ...". At most 20 such lines are shown per run, then "(more handled errors not shown)". A handled error is often the first sign of a bug that the app hides behind a friendly message. Published apps never print these lines. In Fix mode, the lines reach the AI assistant with the Last Run output chip.

When an error is **not** handled by your code, a run from the IDE shows a run-time error dialog in the style of VB6, for example:

```
Run-time error 5: Invalid procedure call or argument
Font must be Family|Size or Family|Size|Regular/Bold|Normal/Italic (6–96
DIP), or empty.
Forms/frmMain.bas, line 23, in Form_Shown
```

Its buttons are **End**, which ends the app; **Go to Line**, which ends the app and opens the code editor at that line; and **Copy**, which copies the full details, including the MCB code, for example to paste them into Fix. The error also appears in the **Output** pane, so Fix receives it with the **Last Run output** chip. A published app shows a calmer dialog instead: "An unexpected error occurred: …" and "The app will close.", with **Copy Details** and **Close**; like VB6, the app ends after the error, and the dialog shows no folders or file paths.

Test a release build

Build or publish the app and test that output separately. A project that works inside the IDE may still have a missing resource, package, or path in the published app.

Use the debugger to answer one question

When a procedure runs but produces the wrong result, set a breakpoint on the line where the value should change and run the smallest action that reproduces the problem. Inspect the current procedure, local variables, form fields, and the call stack. A watch is useful when you want to compare a value before and after one statement.

Keep a debugging session narrow:

1. reproduce the problem once;
2. stop at the first line whose value surprises you;
3. inspect the input and the branch condition;
4. step over one call at a time;
5. remove or disable the breakpoint after the fix;
6. run Check and the same user test again.

The debugger is for finding a cause, not for replacing validation. A value that looks right in one session may still fail for an empty string, a second window, or a published resource path.

Read the first error first

When several messages appear, begin with the first one. Later messages may be consequences of the same problem. Read the file, line, control, or property named by the diagnostic before changing code.

A small debugging loop

1. Reproduce the problem with the smallest input.
2. Write down what you expected.
3. Write down what happened.
4. Add one small diagnostic message if needed.
5. Change one thing.
6. Check and run again.

Do not make five unrelated changes and then guess which one fixed the problem.

Test the unhappy path

For each important action, test at least:

- an empty value;
- a normal value;
- a very long value;
- a value with spaces or accented characters;
- a missing file or database;
- a cancelled operation;
- closing the window at the wrong time.

Write automated tests

Manual tests find what a user sees. Automated tests check the logic behind it every time you choose **Test**. A test module is a `.bas` file declared in `project.mcproj` under `"tests"`:

```
"tests": [
  {"name": "TodoStoreTests", "path": "Tests/TodoStoreTests.bas", "kind":
"unit"}
]
```

The `name` is a BASIC identifier that no form, module, or class of the project uses; `path` is relative to the project; `kind` is `"unit"` or `"ui"`. Test modules count toward the project's limit of 1,000 documents (see "How large a project can be" in Chapter 22). When the AI assistant adds a test, it saves it as `Tests/<name>.bas` and adds this entry for you.

Write a test

The runner calls every procedure whose name starts with `Test_`. Each one must be `Public Sub Test_Name()` or `Public Async Function Test_Name() As Boolean`, without parameters; an `Async` test fails when it returns `False`. Any other `Test_` procedure, or a test module without one, stops the run with MCB5201.

Check results with `Assert`:

- `Assert.AreEqual expected, actual` compares two values of compatible types; with `Variant`, both values must be `Variant`;
- `Assert.IsTrue condition` and `Assert.IsFalse condition`;
- `Assert.IsNull value`;
- `Assert.Fail message`.

There is no `Assert.AreNotEqual`; write `Assert.IsTrue a <> b`. A failed assertion stops that test and reports its message and line. Any other runtime error also fails the test.

```
Option Explicit

Public Sub Test_SaveAndReadBack()
    Dim target As String
    target = AppPaths.DataFile("tests/Todo/roundtrip.json")
    File.WriteText target, "[""milk""]"
    Assert.IsTrue File.Exists(target)
    Assert.AreEqual "[""milk""]", File.ReadText(target)
    File.Delete target
    Assert.IsFalse File.Exists(target)
End Sub

Public Async Function Test_ReadsSecondTitle() As Boolean
    Dim items As JsonValue
    items = Json.Parse("[""milk"",""eggs""]")
    Assert.AreEqual 2, Json.Count(items)
    Test_ReadsSecondTitle = CStr(Json.Value(Json.At(items, 1))) = "eggs"
End Function
```

A test module is a standard module of the project, so it can call the `Public` procedures and functions of the other modules. Test sources never go into Build or Publish output.

Unit and UI tests

A `"ui"` test module needs a desktop project with a startup form; otherwise the Test command reports MCB2301. A project that starts from `Sub Main` has no startup

form, so its tests must be `"unit"` modules. Use `"unit"` modules for the logic in modules. When every test module is `"unit"`, the tests run without any window.

When any test module is `"ui"`, McBasic™ first shows the startup form, so its `Form_Load` has run, and then runs every test of the project, unit modules included. A test can then call the `Public` procedures of a form, for example `frmMain.ResetGame` followed by `Assert.AreEqual 0, frmMain.Moves()`, and read or set its controls through the form, as in `frmMain.lblMoves.Caption`. The startup form is closed after the last test.

A UI test cannot type into a text box or click a button. Keep the app's rules in modules and test them with unit tests; do not add `Public` procedures to a form only so that a test can drive it (see "Keep the work out of the forms" in Chapter 8).

Run the tests

Choose **Test**, next to **Check**, or run `mcbasic test <Project.mcbproj> [--timeout seconds]` in Terminal. The project must pass Check first. Each test is reported as passed or failed, with its duration and the assertion message.

Test runs do not use the application's own data folder: `App.DataPath`, `AppPaths.DataFile`, and `Settings` point to `~/Library/Application Support/McBasicTests`, which the tests of every project share. Keep test files in a subfolder named after the project, and delete them before the test ends.

Save and recovery

Save often. If McBasic™ reports a pending recovery file or an external file change, stop and read the message. Do not overwrite a file just to make the warning disappear.

> **[IMAGE PLACEHOLDER — Figure 15.1: A useful diagnostic]**
> Show: a Check result with one selected error, its source location, and the related code or property visible.
> Purpose: show how to move from a message to the place that needs fixing.
> Labels: error message, source location, selected line, corrected result.
> Crop: result area and a readable code or Inspector detail.
> Alt text: “A McBasic™ diagnostic points to the code or property that needs a fix.”

Code written with the AI assistant

Treat code from the AI assistant like code from a colleague. McBasic™ checks every AI-generated stage and change before it is applied, and each one is a single Undo action, but Check proves only that the project is valid. Run the same normal and unhappy-path tests on generated code as on your own. When a stage cannot pass Check after three attempts, the chat shows its real diagnostics on a failure

card, in the same format as the Results area. The next chapter, Chapter 16, shows how stages are checked, applied, and undone.

Check your work

- Every important workflow has a normal and an error test.
- Logic in modules has `Test_` procedures that pass when you choose **Test**.
- You can reproduce a reported problem.
- Your fix is followed by a new Check and a new run.
- You know whether your evidence came from the IDE, the generated app, or a published app.

Tips and tricks

- **Read the Diagnostics list after a Check.** The list fills when you Check. Since Build 836 the automatic Check also runs on open, on save, after AI changes and after a pause in typing. Turn it off in **Settings… > Code > Check automatically**.
- **Look for the underline in the editor.** Errors are underlined in the code, and holding the pointer over the underline shows the message.
- **A handled error can hide a real bug.** When you Run from the IDE, errors that your code catches appear in the **Output** pane as "Handled runtime error …" with file and line. If your app shows a friendly message but something is wrong, look there first.
- **Pressing Stop on your running app is not an error.** Stopping the app you started is a normal end, so it does not count as a failure.
- **Fix the first error first.** Later messages are often a result of the first one.

Chapter 16 — Work with the AI assistant

McBasic™ includes an AI assistant in a chat panel beside your project. You can ask it questions about McBasic™, ask it to explain or improve your code, and ask it to plan and build a complete small application. Everything it changes is checked by McBasic™ first and shown in the chat as a card you can read, undo, or, if you prefer, approve one change at a time.

The assistant is optional: the IDE works fully offline without it, and nothing is sent anywhere until you choose a provider and press Send.

What the assistant can and cannot do

The assistant is a helper, not a second developer with its own access to your Mac. Keep these boundaries in mind:

The assistant can...	The assistant cannot...
answer questions about McBasic™, its components and its language	see any project document you did not select as context
explain, review, or suggest BASIC code for the documents you share	skip McBasic™'s Check: every generated stage and change is checked before it is applied
propose a replacement for one procedure, or changes to a form	apply a change over an edit you made after it was checked
plan a complete application in stages and build the stages one after another	save the project, run it, or test it for you
look up McBasic™'s own component, form, function, and language reference	read your files, browse the web, or use the network for your app

A good answer is still only a proposal. Every change the assistant applies is one Undo action, and you remain responsible for testing the result with **Check**, **Run**, and your own test cases (see Chapter 15).

The assistant builds menus as Menu Editor items: menu titles, items, separators, shortcuts and pop-up menus (see Chapter 8). It never adds the old Menu component. On a form that still has an old Menu, it tells you to select it in the designer and choose **Convert to Menu Editor…** first.

Choose and set up a provider

McBasic™ does not include an AI model. It connects to one of three kinds of provider, which you choose in **AI Provider Settings**. Open it with **McBasic > AI Provider Profile…** or with the gear button at the top right of the chat panel. Opening the settings sends nothing.

Provider (as shown in the menu)	What it uses	What you need
External Codex (ChatGPT sign-in)	The Codex runtime that ships inside OpenAI's ChatGPT app	the ChatGPT app installed separately, and a ChatGPT account
Responses API (API key)	An HTTPS endpoint that speaks the Responses API, such as OpenAI or OpenRouter	an endpoint address, a model name, and your own API key
LM Studio (local)	A model loaded in LM Studio on this Mac, through LM Studio's local server	LM Studio installed separately with its server started, and a model; an API key only if you turned one on in LM Studio

The settings window is one form with three boxes. **Provider** holds the provider, the model, and the provider's own fields. **AI chat** holds the choices described in "Choose how AI changes are applied" and "Limits". **This project** holds **Save chats in this project** and appears only while a project is open. Labels sit in one column on the left, and a grey line under a control explains it. **Save** (Return) keeps every change in the window; **Cancel** (Esc) keeps none of them.

Save always tells you what happened, in the bar beside the buttons:

- When the settings are saved, a green check mark and a line such as "Saved: Responses API (OpenRouter, https://openrouter.ai/api/v1/responses), model deepseek/deepseek-v4.1-flash. API key stored in Keychain. No network request was made." appear for about a second, with **Save** and **Cancel** dimmed; then the window closes by itself. The same line stays in the status bar of the project window (or in an alert when only the Welcome window is open).
- When something must be fixed, the window stays open with everything you typed. A red warning sign and a message in red beside **Save** name the field first, for example "Responses endpoint: enter an HTTPS address such as https://openrouter.ai/api/v1 …" or "Model: enter the model name …". That field gets a red outline and the cursor; the outline goes away as soon as you edit the field. Fix it and choose **Save** again.

- Spaces and line breaks pasted before or after the model, the endpoint, or the API key are removed when you save; they are never an error.
- If the Keychain cannot store the API key, the message starts with "API key:" and nothing is saved, not even the provider: the previous settings stay as they were.

External Codex with your ChatGPT account

McBasic™ uses the Codex runtime that is installed with the ChatGPT app. There is no API key to manage and nothing to install with Homebrew or npm. McBasic™ does not bundle, copy, or modify Codex, and it does not read Codex's credentials.

1. Install the ChatGPT app from OpenAI, if it is not already on your Mac.
2. Open **AI Provider Settings** and choose **External Codex (ChatGPT sign-in)** in **Provider**.
3. Choose **Autodetect Codex**. McBasic™ looks in your Applications folders, checks the executable with a version request only, and fills in **Codex executable:**. If you prefer, choose **Choose…** and select the executable yourself.
4. Choose **Sign In…**, next to **Provider:**. McBasic™ saves the profile first, then opens the ChatGPT sign-in; complete it.
5. Open the settings again. The **Model:** menu lists the models your Codex offers; choose one and choose **Save**. Use **Refresh** if the list is empty after signing in.

The **McBasic** menu also has **Check Codex Installation**, **Check Codex Status**, **Sign in to Codex with ChatGPT…**, and **Sign Out of Codex**.

A Responses API endpoint (for example OpenRouter)

1. Choose **Responses API (API key)** in **Provider**.
2. In **Responses endpoint**, type the address your provider gives you: its base address ending in `/v1`, for example `https://openrouter.ai/api/v1` or `https://api.openai.com/v1`, or the full address ending in `/v1/responses`. When you save, McBasic™ completes a base address with `/responses` and shows the completed address in the field.
3. In **Model**, type the model name your provider expects.
4. Paste your key into **API key (Keychain):** and choose **Save**.

The key is stored in the macOS Keychain, never in the project. To forget it, select **Remove the saved API key** and save. Saving makes no network request.

When the endpoint is on `openrouter.ai`, two more rows appear under it. Neither row appears for any other endpoint, and neither is a secret: they are saved in the provider settings, not in the Keychain.

- **Providers:** the OpenRouter provider names to prefer, in order, separated by commas and spelled as shown on openrouter.ai, for example `Azure, OpenAI`. Spaces around a name and repeated names are ignored. Leave it empty and OpenRouter chooses the provider.
- **Allow fallbacks** (on by default): when it is off, only the providers listed in **Providers:** are used, and a request fails instead of switching to another provider and price. When it is on, OpenRouter may use another provider if the listed ones cannot answer.

When **Providers:** is not empty, McBasic™ sends these two choices with each request to OpenRouter; with **Providers:** empty it sends neither and OpenRouter chooses. List at most 20 names, each up to 100 characters. It does not download a provider list or show prices; the openrouter.ai website has both.

LM Studio (local)

LM Studio (the LM Studio app (Bionic) or the `lms` command) runs models on your own Mac. McBasic™ talks to the LM Studio server through the same Responses API as the providers above and uses the same chat modes, plans and cards. Whether a model can call McBasic™'s tools, and how its replies stream, depends on the model and on the LM Studio version: a model without tool use can chat, but cannot run plans or propose changes. McBasic™ does not install LM Studio, start its server, or download models.

1. Install LM Studio, download a model in it, and start the LM Studio server: from the LM Studio app, or with `lms server start` in Terminal.
2. In **AI Provider Settings**, choose **LM Studio (local)** in **Provider**.
3. Keep **Server endpoint:** at `http://localhost:1234/v1/responses`, the address LM Studio uses by default, or type the address and port your LM Studio server uses. The base address LM Studio shows, such as `http://localhost:1234/v1`, also works: **Save** and **Load Models** complete it with `/responses`. McBasic™ accepts only an address on this Mac (`localhost` or a loopback address such as `127.0.0.1`), with its port.
4. Choose **Load Models** next to **LM Studio models:**. McBasic™ asks the LM Studio server for its model list and nothing else. The menu lists the chat models you downloaded, each marked "loaded" or "not loaded". A model marked "(tool use not reported)" may not run plans or propose changes; you can still choose it, and the note then says "This model does not report tool use. AI plans and changes may not work with it; chat still does." With an empty **Model:**, McBasic™ picks a loaded model that reports tool use. The model's name goes into **Model:**, which you can also type yourself.

5. Leave **API key (Keychain):** empty unless you turned on authentication in the LM Studio server settings; the grey line under it says "Optional. LM Studio needs a key only if you turned on authentication in its server settings." Choose **Save**. Saving makes no network request.

Load Models is the only thing in the settings that contacts LM Studio, and only when you press it. If it cannot, the note at the bottom of the **Provider** box says why:

- "The LM Studio server is not running on localhost:1234. Start it from the LM Studio app (Bionic) or with `lms server start`, then try again." (with the address you entered).
- "LM Studio refused the API key. Check the key in AI Settings, or clear it if LM Studio does not require one."
- "LM Studio lists no models. Download or load a model in LM Studio, then press Load Models again."

When you send a message, the same first two messages can appear in the chat. When LM Studio refuses a request with its own explanation, the chat shows it after "LM Studio answered:", for example "LM Studio answered: No models loaded. Please load a model in the developer page or use the 'lms load' command." When LM Studio gives no explanation and has no Responses API, the chat says "This LM Studio version does not offer the Responses API. Update LM Studio, then try again." An LM Studio request carries at most 120,000 characters (see "Limits"). Project source still needs your consent for each activity, as with every provider, even though it stays on this Mac. A local model is usually slower than a cloud model, and a small one may follow McBasic™'s instructions less well.

A provider this version no longer offers

If your settings name a provider that this version of McBasic™ no longer offers, McBasic™ treats it as no provider: the chat says "Choose a provider in AI Settings to send." and opens **AI Provider Settings** when you send. Choose one of the providers above and choose **Save**. Chats you saved with the old provider still open.

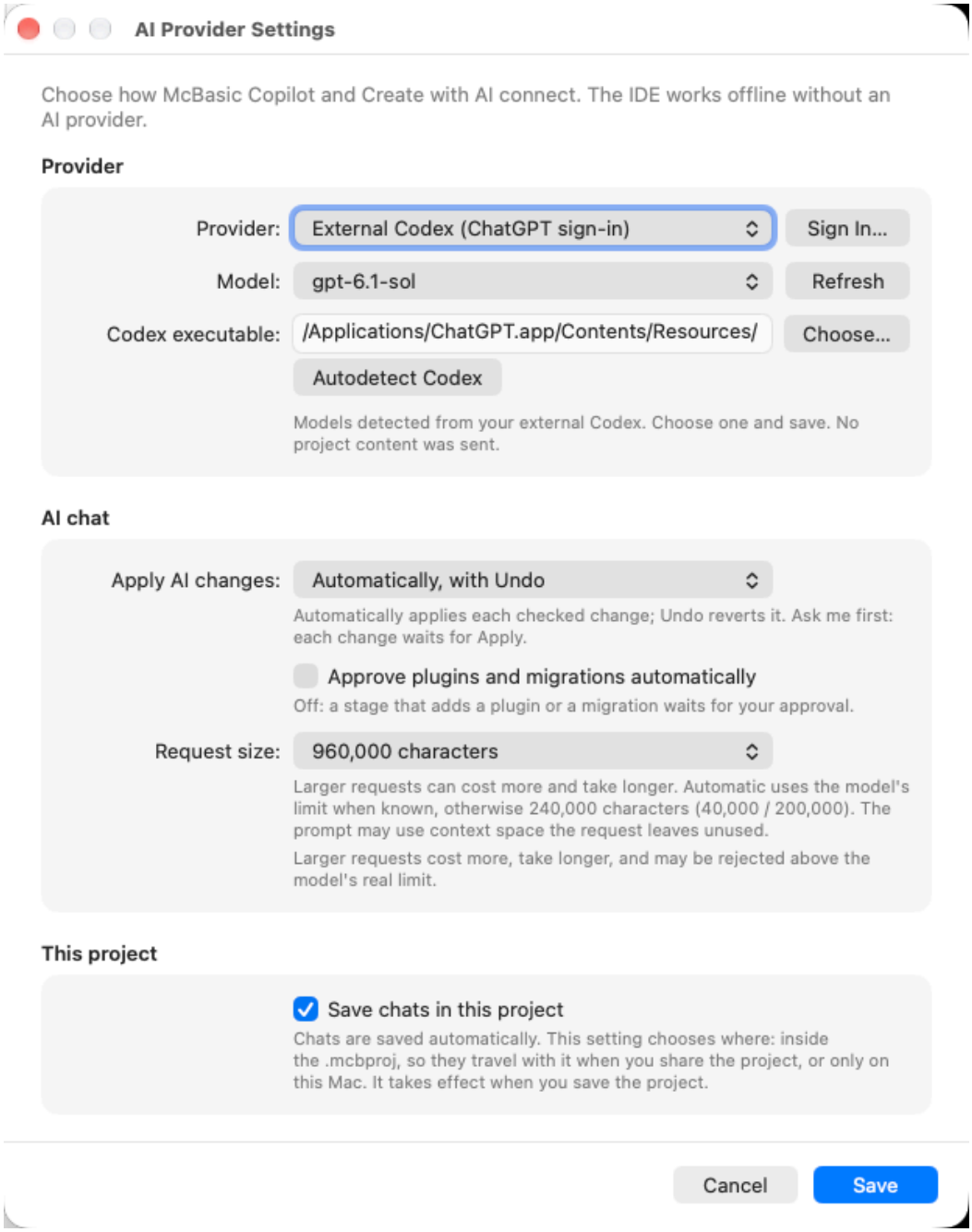


Figure 16.1: AI Provider Settings

A tour of the chat panel

Open the panel with the **AI** button in the toolbar, or with **McBasic > Show AI** (Shift–Command–L). The panel opens on the right side of the project window, with a thin border that separates it from the editor. A project must be open.

Part	What it is for
Chats menu (top left)	A button with a speech-bubble symbol, the current chat title and a chevron; "(3 chats)" follows it when the project has more than one chat. Use it to switch, start, rename, or delete chats.
Show exact content button	Shows or hides the exact project text that Send will transmit.
Gear button (top right)	Opens AI Provider Settings; its tooltip names the provider and model.
Conversation	A white area with rounded corners inside the grey panel (dark in Dark Mode). It holds your messages on the right, the assistant's replies on the left on a tinted surface, and the cards for plans, stages, and changes. Cards have a very light grey fill (a dark grey in Dark Mode) and error cards a light red one, so they stand out from the white area.
Status row	Shows **Connecting…**, **Thinking…**, **Writing…**, or **Retrying…** with the elapsed seconds, then what the assistant is doing when McBasic™ knows it: `Looking up: component DataGrid`, the number of characters received so far (`1,240 characters`), or, after a minute without any sign from the assistant, "Still working; last signal 1 min 20 s ago". Smaller counters at the right end of the same row count the tokens and lookups of the request (see "Live activity"). The row is always one line: when the panel is too narrow, the status ends with "…", and holding the pointer over it shows the full text.
Context chips	The project documents this chat sends; **+ Add** chooses more. After them come the chips of the chat mode, such as **McBasic reference** and **Project map**, and the scope chips that **Add to Chat** creates (see "What the chat sends in each mode").
Composer	The message field, with buttons to attach files, dictate, Send, and Stop. The mode pill, for example "● Build", sits inside the message field on the left and chooses the chat mode

Part	What it is for
	when you click it, and the field's border takes the mode's colour (see "Chat modes").
Usage line	Tokens used by the last reply and by the whole chat, as reported by the provider.

Each message has a small header: `You · 15:42` on your messages and `AI · 15:43 · 12s` on replies, where the last part is how long the reply took. A reply you stopped ends with `· interrupted`. Chats saved by builds before Build 811 show – instead of a time.

Replies are formatted: headings, lists, bold and italic text, inline code, and links (a link opens in your browser only when you click it). A reply longer than 30 lines is shortened, with **Show all** to expand it.

BASIC code in a reply appears in a box coloured like the code editor, with two buttons:

- **Copy** puts the code on the clipboard;
- **Insert into open module** also copies the code and moves the focus to the

editor. In this build it does not insert the code itself: click where the code belongs and paste it with Command–V.

When the assistant consults McBasic™'s reference, a grey line such as `looked up: component TextBox` appears in the conversation. Those lookups are answered on your Mac from McBasic™'s own documentation. Reference lookups are not limited: the assistant may read any McBasic™ reference topic it needs (about 154 topics: the components, the intrinsic functions, forms, and the language). When it consults the same topic again, McBasic™ answers again, but the chat shows only one `looked up:` line per topic in each request.

Only an assistant that keeps requesting lookups without ever finishing is stopped: after 500 lookup calls in one request, the request ends with "MCB4733: The AI kept requesting reference lookups without finishing. Try again or rephrase." Nothing is applied.

Under each reply, a grey line reports the tokens the provider counted, for example:

```
in: 95.3k (cached: 63.7k)  out: 5.2k  = 100.5k
```

`in:` is the input tokens the model read during the request, counted again at each step, so every lookup adds to it; `cached:` is the part of that input the provider served from its cache, which is cheaper and faster; `out:` is the tokens the model wrote; and `=` is the total of input and output. `k` means thousands and `M` millions. When the provider reports a price, the line ends with it, for example `≈ $0.0123`. A

provider that reports nothing shows "usage unavailable". The line at the bottom of the panel, `Chat total — …`, adds up the whole chat.

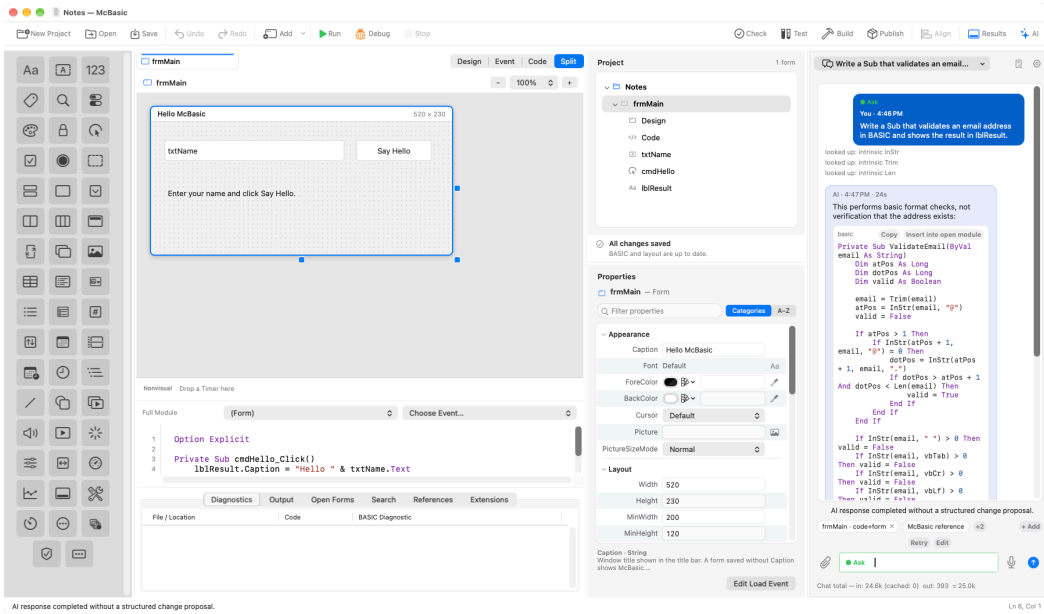


Figure 16.2: The AI chat panel

Conversations

A project can have several chats. Chats are saved automatically after every message; there is nothing to save by hand. When you open a project, its chat shows the most recent conversation again, with its plan card and, if the plan is paused, its **Resume** and **Stop plan** buttons. Nothing is sent until you press a button or send a message. A project without saved chats opens an empty chat.

The **Chats** menu at the top left of the panel shows the current chat's title. Its tooltip reads "Chats in this project: switch, start a new chat, rename or delete". The menu holds, in order:

- **New Chat** starts an empty chat and proposes the open document as its context;
- under **Chats in this project**, one row per chat with its title and when it was last used, for example "today 15:46", "yesterday 09:12", or a date. A check mark shows the open chat. A row can end with "⏸ plan paused", "✓ plan complete", or, for the open chat while a request runs, "● running". With no saved chats, the menu shows "No saved chats yet";
- **Rename Chat…** opens a field in the header with the hint "Return to save, Esc to cancel";
- **Delete Chat…** asks first. Deleting removes the chat from the project's chat storage and cannot be undone;
- at the end, "2 chats could not be read" when some chat files are damaged.

A new chat takes its title from the start of your first message. Chats from earlier builds appear as "Chat of" followed by their date. The menu items are unavailable while a reply or a plan stage is running, and their tooltip says "Wait for the current AI reply or press Stop to change chats".

Each chat keeps its own plan. When you switch away from a chat whose plan is waiting for approval or is paused, its cards come back when you return, with two exceptions, because a checked proposal is never saved:

- A plan stage waiting at **Ready to apply**, for a dependency decision, or after a failed apply is not kept. Its checked proposal is dropped without being applied. When you return, the plan card shows **Paused** with the line "Paused while you were in another chat. Resume to continue.", and **Resume** generates that stage again on the current project.
- A single change waiting at **Ready to apply** (outside a plan) is not kept

at all. When you return, its change card is gone and nothing was applied. Apply or discard it before you switch chats, or ask for it again.

If you delete a chat whose approved plan is paused, the confirmation adds: "Its approved AI plan is paused. Deleting the chat also discards the plan; stages already applied stay in the project."

A chat can hold up to 500 messages. When a chat is full, McBasic™ says "This chat is full. Start a new chat to continue."

Context chips: what the chat may send

The row of chips above the composer is the most important control in the panel. Each chip names one project document and what is sent from it, for example `frmMain · code+form`:

- `code` sends the document's BASIC source;
- `form` sends the form layout (its controls and properties);
- `code+form` sends both.

The chips are your consent for this conversation. Whatever they select is exactly what each request in this chat may transmit, so there is no separate confirmation when you press Send, and none for each stage of a plan you approved.

- A new chat proposes the document that is open in the editor.
- **+ Add** opens a list of the project's documents inside the panel. Select **Code**, **Form**, or both for each document you want to share, then choose **Done**. The list shows every document of the project, up to the project limit of 1,000, but one chat sends at most 64 documents. With more selected, Send stops with "Select at least one BASIC source or layout for each selected document (up to 64)." Remove chips until 64 or fewer are left.

- The **×** at the end of a chip removes it.
- The row shows as many whole chips as fit on one line. When there are more, it ends with a small **+N** button, for example **+3** for three chips that are not shown. Choose it to open a list of everything the chat sends, one line per chip, each with its **×**, and **+ Add** at the bottom. Removing an entry there is the same as its chip's **×**, and the list stays open and updates. Press Esc or click outside the list to close it. The row never scrolls, so the chips stay readable with any scroll-bar setting. With VoiceOver the button reads, for example, "Show all 5 context entries, 3 not shown", and each **×** reads "Stop sending" followed by the chip's name.
- With no document chips, no **Project map** chip and no scope chip, the row says **No project source will be sent**. You can still ask general questions; the assistant then knows nothing about your project. The **Project map** chip is on in every chat until you remove it, so remove it too when you want to send nothing about the project.
- Each chat keeps its own chips while McBasic™ is open. They are never saved in the project, so a project you share never carries your consent with it. The only documents a chat brings back by itself are the ones the stages of a plan you approved on this Mac created (see "Chips during a plan"). Scope chips (◎) are different: they are saved with the chat (see "Add to Chat and the scope").
- After the documents, the row shows the chips that belong to the chat mode: **McBasic reference**, **Project map**, **Decisions (n)** and the history the mode sends. "What the chat sends in each mode" explains them.
- While a reply is running, and for the whole time a plan's stages are running, the chips are frozen: what a request sends was fixed when it started. The **+N** list can still be opened to read, but its **×** and **+ Add** are disabled until the request ends.

Chips during a plan

Every stage of a plan, every repair attempt, and every restart is built from the chat's **current** chips:

- If you change the chips while a plan waits for approval or is paused, the next

stage uses the new selection. McBasic™ takes a fresh grant from the chips; you do not have to confirm it again.

- If you remove every chip, the plan does not generate code it cannot see. It

pauses with "This chat has no context chips, so the AI cannot see the project. Add documents with + Add, then press Resume."

- Only the chips the chat shows count. A chat whose chips belong to another

project window has none for the plan: **Resume** then pauses with the same "no context chips" message (**Paused: no context**). Add the documents with **+ Add** and press **Resume** again.

- Chips are not saved, so a chat you reopen after quitting McBasic™ has none of its earlier chips. Like a new chat, it proposes the document open in the editor as a

visible chip, and **Resume** sends what that chip selects. The documents that the plan's stages created come back as chips too (next point). Add any other document the next stage needs with **+ Add** before you press **Resume**.

- When a stage creates a form, module, class or test, McBasic™ adds it to the chips (code, and form when it has one) so that the later stages can see it. The chat notes it, for example "Added frmDetails to this chat's context: the AI created it." The plan remembers these documents. When you reopen the chat of a paused plan after quitting McBasic™, or after closing and reopening the project, they join the chips again with the same note, and McBasic™ checks them again before every stage. This does not widen your consent: the assistant wrote these files inside the plan you approved, and the chips show them like any other document.
- When a stage creates a file that is not a form or module, such as a database migration (`Database/migrations/0001_tasks.sql`) or a text resource, McBasic™ adds it the same way as a **read-only** chip, for example `Database/migrations/0001_tasks.sql (SQL, read-only)`, with the same note. The later stages can read the file but never change it: a schema change takes a new migration. McBasic™ sends the file as the stage left it, also before you save the project, and after Save it sends the saved file. The file's text counts toward the request budget, and the exact-content preview shows it after the documents. A file you undo or delete leaves the chips by itself before the next request.
- If you remove one of these documents or files with its **×** after the note, it stays out of this plan's context from then on, also after a restart. Add a document back with **+ Add** if you change your mind; a removed file comes back only if a later stage creates it again.
- A created document that does not fit the request budget is left out, and the chat says "Not added to this chat's context because the context budget is full: modTaskStore. Remove a chip or raise the request size in AI Settings, then add it with + Add." A created file that does not fit is left out with "... Remove a chip or raise the request size in AI Settings; McBasic™ adds it before the next stage when it fits."
- A created document is also left out when the chat already sends 64 documents, and the chat says "Not added to this chat's context because a chat sends at most 64 documents: modTaskStore. Remove a chip, then add it with + Add."
- A plan approved on another Mac never brings any document back.

Show exact content

Choose the **Show exact content** button in the header to see, as plain text, everything the next Send will carry from the project: a short summary, the BASIC and layout text of each selected document, the text of each read-only file, and, at the end, McBasic™'s reference index. The summary ends with a line that compares the selected text with the context budget, for example `Total authorized context: 8412 of 200000 characters (budget)`. The preview follows the chips and your edits. Showing it sends nothing.

The preview is built from the same recipe as the request, so it shows exactly what Send carries in the current chat mode:

- its first line names the mode and the parts it sends, for example `Fix: McBasic reference, project map, frmMain · code+form, diagnostics, last 2 exchanges`;
- the "Left out to stay within the budget: …" notes, when the budget left something out;
- a `Project map (names and signatures): n characters` line and the map's text under `--- Project map, names and signatures (no code) ---`. Before McBasic™ has read the map, the line says "Project map: read from the project when you send (names and signatures, no code). Remove its chip to keep it out.";
- with a scope, a `Scope (always sent): n characters` line, a `Total with the scope: …` line, and the scope's text under `--- Scope (the AI may change only these) ---`, with what each item lets the AI change and, after `--- read-only ---`, what it only reads.

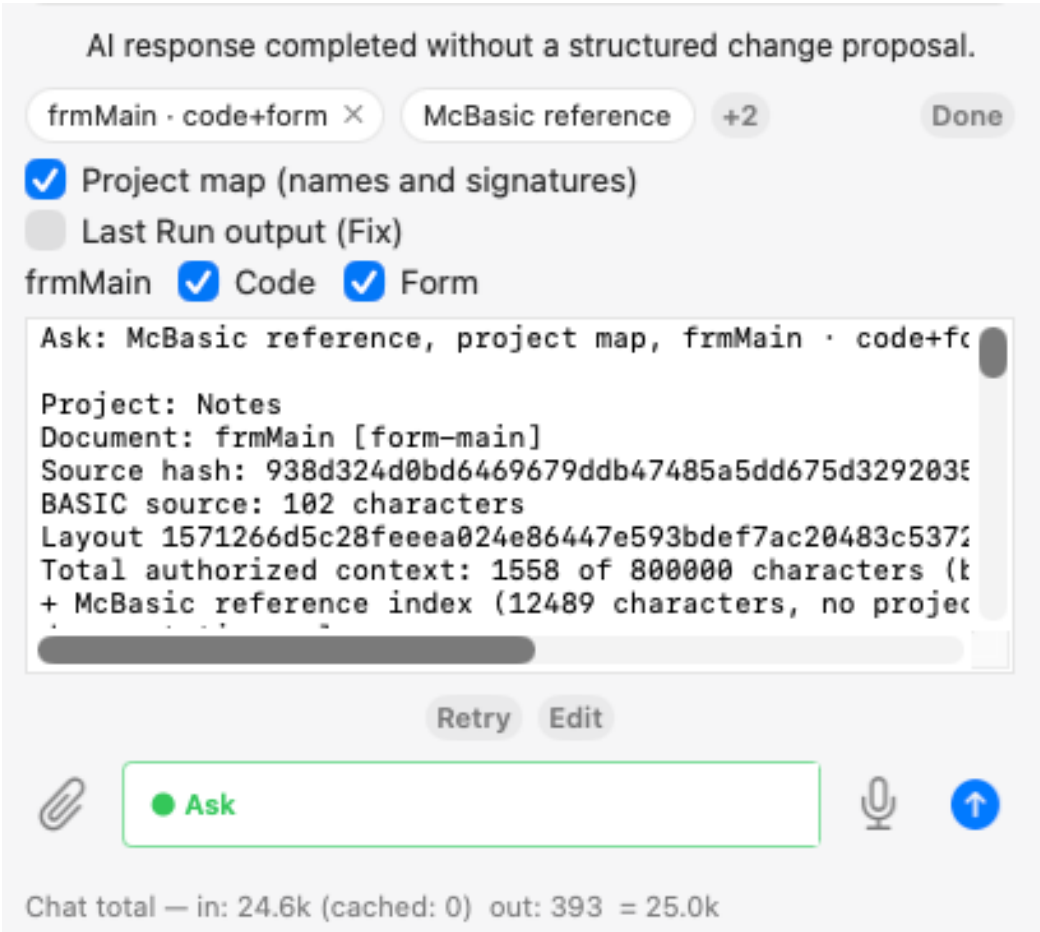


Figure 16.3: Context chips and exact content

Write and send a message

Type in the message field at the bottom of the panel. It starts one line high and grows as you write; past a few lines it stops growing and scrolls, and only then can a scroll bar appear in it.

Action	How
Send the message	Send button, or Command–Return while the message field has focus
Stop a running reply, or pause a running plan	Stop button, or Esc
Edit your last message	Up Arrow in an empty message field, or **Edit**
Send your last message again	**Retry**
Attach a screenshot or text file	Paperclip button (PNG or JPEG images, UTF-8 text files)
Choose the chat mode	The pill in the message field (for example "● Build"), or Command–/ while the message field has focus (see "Chat modes")
Use another mode for one message	Type the command first, for example `/fix the crash`
Dictate	Microphone button (macOS Dictation; review the text before sending)

If no provider is set up yet, Send opens AI Provider Settings first. Attached files are listed above the message field until you send; check the list before you press Send. Their contents are not saved in the chat history.

While a plan's stage is running, the message field is unavailable and only **Stop** works. While a plan waits for approval, or is paused, you can keep writing to the assistant as usual. One exception: if a checked stage is waiting for you (at **Ready to apply**, or for a dependency decision) and you send a new message, the waiting proposal is set aside. Nothing of it is applied, and the plan pauses with "The checked change was set aside by your new message. Resume regenerates this stage on the current project." Press **Resume** to generate the stage again. A single change that waits at **Ready to apply** stays applicable while you ask a follow-up question.

While a plan is paused, a message that is only a button word (retry, riprova, continue, continua, resume, riprendi, stop or ferma, in any letter case) is not sent to the assistant. The chat adds a note that names the buttons of the live card below it, for example "The plan is paused. Use the buttons on the card below: Retry or Stop

plan." McBasic™ never presses a button for you: **Retry** sends a new request and **Stop plan** ends the plan. Any other message is sent as usual.

Retry and Edit

After a reply, **Retry** and **Edit** appear above the composer. They act only on the last exchange:

- **Retry** sends your last message again with the current chips;
- **Edit** puts your last message back in the field so you can change it.

The old answer is dimmed and collapsed as "Previous answer" until the new one arrives, then it is replaced. If you stop a Retry or Edit, or it fails, the previous answer comes back unchanged. If you stop an ordinary message, the partial reply stays, marked `· interrupted`.

Retry and Edit are not offered after a plan or a stage has run in the chat, so the record of what you approved is never erased. Start a new chat to use them again.

Retry and Edit are also refused when the last exchange is a question from the assistant or your answer to one, because the answer is kept as a decision. The chat says "Retry and Edit are not available for a question or its answer, because the answer is kept as a decision. Write a new message instead."

A message that began with a typed command, such as `/fix`, keeps it: Retry sends it again in that mode, and Edit puts the command back in the field with the rest of the text. A message without a command goes in the mode the pill shows when you press Retry or send the edited text.

Chat modes

The chat has six modes. The mode decides what the assistant may do with your project, which tools it gets, and what the chat sends with your message (see "What the chat sends in each mode").

Mode	Command	Colour	May change the project	What it is for
Build	`/build`	blue	yes	Changes the project: change sets, plans and stages. This is the default, and it works as the rest of this chapter describes.
Plan	`/plan`	purple	no	Interviews you, then drafts a

Mode	Command	Colour	May change the project	What it is for
				plan. **Approve and run** switches to Build (see "Plan an app with /plan").
Ask	`/ask`	green	no	Answers questions about McBasic™ or the project. Changes nothing.
Fix	`/fix`	orange	yes, as little as possible	Fixes the diagnostics of the latest Check, Run or Test, or a bug you describe, touching only what is needed.
Review	`/review`	teal	only the findings you tick	Lists findings to tick. **Apply selected** changes only the ticked ones.
Explain	`/explain`	indigo	no	Explains the code or controls in scope or in the chips. Changes nothing.

In Plan, Ask and Explain the assistant has no tool that changes the project. Code it writes in an answer is only text, with the usual **Copy** button. In Review, the change tool is offered only to the request that **Apply selected** sends. In every mode the assistant can look up McBasic™'s reference and ask you a question (see "When the AI asks you a question").

Choose a mode. Click the pill in the message field (for example "● Build"), or press Command–/ while the message field has focus. A list opens with one row per mode: a coloured dot, the name, the one-line description from the table above, and a check mark on the current mode. Use the Up and Down Arrow keys to move, Return to choose, and Esc to close the list, or click a row. The mode you choose stays until you choose another one. The pill's tooltip names the current mode, for example

"Chat mode: Plan • ⌘/". With VoiceOver the pill reads "Chat mode: Plan", and the current row in the list reads, for example, "Plan, selected".

The pill and the border. The current mode is shown as a pill inside the message field, on the left, in the mode's colour, for example "● Plan". The pill is also the control that opens the mode list: the pointer becomes a hand over it, and while a request runs it is dimmed and does nothing. The border of the message field takes the same colour. Build shows its blue pill too.

A command for one message. Type a command as the first word of a message, for example `/fix the total is wrong`. When you type `/` as the first character, the same list opens at the pill as completions: keep typing, or use the arrow keys and Return to complete the command. A recognised command is coloured in the mode's colour, and it applies to that message only: the pill and the mode of the chat do not change. The assistant receives the message without the command. A command it does not recognise, a command later in the text, or `/` alone is sent as plain text.

The mode of each message. Every message you send shows a small tag in the colour of the mode it was sent in, for example "● Fix". VoiceOver reads it as "Sent in Fix mode".

Rules.

- The mode is saved with the chat and comes back when you open the chat again. A new chat starts in Build.
- While a request runs, the pill is unavailable (dimmed), like the chips. A change takes effect from your next message. If you try to change the mode then, the chat says "Wait for the current AI request or press Stop, then change the mode."
- When you press **Approve and run** on a plan in Plan mode, the chat switches to Build by itself and notes "Switched to Build to run the plan".
- Every provider offers every mode.

When the AI asks you a question

In any mode, the assistant can stop and ask you one question when the answer changes what it does: in Build when a request is ambiguous, in Fix when two fixes are possible, and all through the interview in Plan. A question ends the assistant's reply. Your answer starts its next one, so you can answer at once, hours later, or after you quit and reopen McBasic.

The question arrives as a **question card**:

- the question is the card's title;
- for a single choice, each option is a button. When the assistant gave an option a description, it appears under the button's label. The option the assistant

recommends has the accent colour, and VoiceOver reads "Recommended." in its help;

- for several choices, each option is a checkbox, and **Confirm** sends the ones you ticked;
- **Other…** is always there. It opens a field, "Your answer", for your own words, with **Send** (or **Add**, for several choices, which adds your text to the ticked options; then press **Confirm**);
- for a single choice with a recommended option, Return picks that option: on the card, or in an empty message field.

You can also simply type your answer in the message field and send it. While a question waits, a message you send is its answer, in your own words, and goes in the mode the question was asked in.

After you answer, the card shrinks to one line, "Question → answer", for example "Where should tasks be stored? → SQLite database". Your answer is kept as a **decision**: the chat sends it with later requests (see "What the chat sends in each mode"), and in Plan it appears on the **What we decided** card.

A question that waits is saved with the chat. When you open the chat again, also after quitting McBasic™, the card is still there and you can answer it. Older questions that were never answered are shown with their buttons off.

If the assistant also proposed a change in the same reply as its question, the change is not applied. The chat notes "The AI also proposed a change in this reply. It was set aside: answer the question, and the AI proposes it again if it still applies."

Ask questions or ask for changes

In Build, the default mode, you can use the chat in two ways. Ask, Explain and Plan never change the project (see "Chat modes").

Ask a question. "Why does this Click handler not update the label?" or "Which component should I use for a list of dates?" The answer is text, perhaps with code you can copy. Nothing in the project changes.

Ask for a change. "Add a Clear button that empties both text boxes" or "Build a small calculator". The assistant then answers with a structured proposal, which McBasic™ validates and shows as a card in the chat:

- a change to one procedure, to one or more forms, or a small set of files

becomes a **change card** (see "Single changes outside a plan");

- a larger request becomes an application **plan**, built in stages (see

"Plans and stages").

No window opens for any of this. The chat shows every proposal, every stage, and every question McBasic™ needs to ask you as a card, and each card updates in place as the work goes on.

Cards: how the chat shows AI work

A card is a framed box in the conversation. It has a title, a status line, some lines of text, and, when you can act on it, buttons. These are the cards you will meet:

Card	When it appears	Buttons it can show
Plan card	The assistant proposes a plan, and then for the whole plan run	**Approve and run**, **Change…**, **Discard**; later **Resume**, **Retry**, **Stop plan**
Stage card	One for each stage of an approved plan	**Apply**, **Discard** (in Ask me first mode); **Undo**
Dependency card	A stage adds a plugin reference or a database migration file	**Approve**, **Reject**
Failure card	A stage or change did not pass Check after 3 attempts	**Retry stage**, **Skip stage**, **Stop plan** (or **Retry**, **Discard** for a single change)
Incomplete card	The assistant applied part of a stage, and McBasic™ stopped continuing it by slices (or the assistant did not say what is missing)	**Retry stage**, **Continue anyway**, **Stop plan** (or **Retry**, **Continue anyway** for a single change)
Change card	A single change outside a plan	**Apply**, **Discard** (in Ask me first mode); **Undo**
Restart note	You changed the project while a stage was running	none: it informs you
Completion card	Every stage of the plan is settled	**Run**
Error card	A request or a chat action failed	**Retry**, when the request can be sent again
Question card	The assistant asks you something (see "When the AI asks you a question")	one button per option, or checkboxes and **Confirm**; **Other…**; in a Plan interview **Stop asking, draft the plan**
What we decided	A Plan interview has its first decision	none: **Details** ▸ lists the decisions

Card	When it appears	Buttons it can show
Draft plan card	Plan mode proposes a plan ("Draft 1 · …")	**Approve and run**, **Discard plan**
Review findings	Review mode lists what it found	one checkbox per finding; **Apply selected**
Permission card	A change touches something outside the chat's scope (see "Add to Chat and the scope")	**Yes**, **No**, **Yes, and add to scope**

A card's buttons always show their full names. When the chat panel is too narrow for all of them on one line, the buttons continue on a second line.

Only the newest cards act. When a newer plan or change replaces an older one, the older cards keep their text but lose their buttons, so an old card can never change the current work. If an action is no longer possible, the chat says so, for example "This action is no longer available." or "This card belongs to an earlier plan or change. Use the newest card."

While a plan is paused, the card you act on is always the last entry in the chat. That is the red error card when a request failed, or the card that needs your decision: the failure, incomplete, stage, dependency or permission card, or the plan card itself (for example after McBasic™ reopens the chat). A reply, a note or anything else that arrives while the plan is paused goes above it, so you never have to scroll up to find **Retry**, **Resume** or **Stop plan**. The card is moved, not copied, and it stays where it was moved after the plan goes on. There is only one live card: while the red error card offers **Retry**, the plan card keeps only **Stop plan**.

Choose how AI changes are applied

Two settings decide how much the chat asks you. They are in the **AI chat** section of **AI Provider Settings** (the gear in the chat panel). They are stored on this Mac, never in the project, and they apply to every project you open on this Mac.

Setting	Choices	What it does
Apply AI changes:	**Automatically, with Undo** (the default) or **Ask me first**	With **Automatically, with Undo**, every checked stage and change is applied at once, and the card offers **Undo**. With **Ask me first**, each one stops at **Ready to apply** with **Apply** and **Discard**.

Setting	Choices	What it does
Approve plugins and migrations automatically	off (the default) or on	When off, every plugin reference and every migration file a stage adds waits for your **Approve** on a dependency card. When on, they pass, and the card says so.

The same section also holds **Request size:**, which is described in "Limits".

A grey line under each control explains it. Under **Apply AI changes:** it says "Automatically applies each checked change; Undo reverts it. Ask me first: each change waits for Apply.", and under the checkbox it says "Off: a stage that adds a plugin or a migration waits for your approval."

The settings take effect when you confirm the dialog; **Cancel** leaves them as they were. McBasic™ reads them before each plan, each change, and each card button, so a change you make while a plan is paused applies when you press **Resume**. A change you make while a stage is running applies from the next button you press.

Changing the provider during a plan. The provider and model in AI Provider Settings are part of what you approved. If you change them while a plan runs on its own, McBasic™ sends nothing more with the old provider and nothing with the new one: the plan pauses before its next request with, for example, "AI provider changed to Responses API (API key). Resume to continue with it." (the message names the provider kind, also when only the model changed). If no provider is selected any more, the message is "No AI provider is selected. Choose one in AI Settings, then press Resume." Pressing **Resume**, **Retry**, **Retry stage**, or **Approve and run** is your consent to the provider now selected: the stage is generated with it, from the current chips, as a new provider conversation. If you press one of those buttons while no provider is selected, the chat says "Choose an AI provider in AI Settings, then press the button again. Nothing was sent."

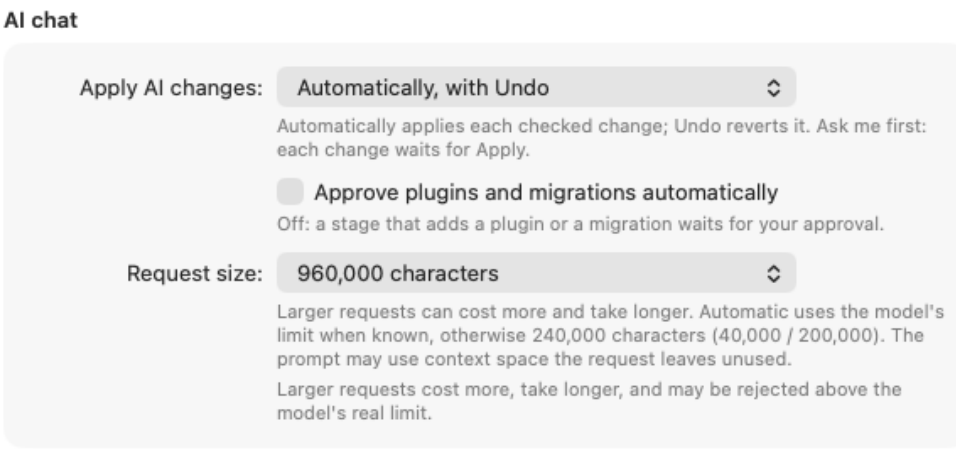


Figure 16.4: The AI chat settings

Plans and stages

For a request that needs several forms, modules, or steps, the assistant first proposes a plan, then builds it stage by stage.

The plan card

The plan arrives as a card, with an accent-coloured border:

- the **title** is the application name, for example `Calculator`;
- the first line is the goal, for example

`Standard application · 1 form · 3 stages` (an MDI application says `MDI application`);

- then one numbered line per stage, with its title and purpose, for example

`1. Layout — Place the number boxes and the operator buttons`;

- **Details** opens the full plan: the application name and kind, its forms

(with the startup form), modules, navigation, data, any trusted plugins and resources, tests, assumptions, and the stages with their deliverables.

The status line reads "Review the plan, then press Approve and run." No files are created while the plan waits. The card has three buttons:

- **Approve and run** starts the plan. Every stage then runs on its own, as

described below.

- **Change…** puts `Change the plan:` in the message field, with the cursor at

the end. Type what you want changed, for example "use a ListBox for the history", and send. The assistant answers with a revised plan, which arrives as a new plan card. The older card stays in the conversation without buttons.

- **Discard** drops the plan. Its card becomes "Plan discarded", and nothing is

created.

You can also keep talking without pressing a button: ask why a stage is needed, or ask for a smaller plan. The plan card keeps waiting, and a revised plan replaces it in the same way as **Change…**.

This is the plan card of Build mode, where a large request can lead straight to a plan. In Plan mode the assistant interviews you first, and its plan card is a numbered draft with **Approve and run** and **Discard plan** (see "Plan an app with /plan").

If the assistant needs more information before it can plan, the plan ends with **Clarification required:** and its questions instead of a plan card. Answer them in the message field and send; the assistant then revises the plan.

The plan decides the windows first. Before it plans the stages, the assistant decides which windows the application needs: the startup form for the main view, modal dialogs for creating and editing items, and separate Settings or About windows when they are needed. Under **Details**, each form has its purpose and the data it receives and returns, for example a task editor dialog that receives a task's title and due date and returns whether the user clicked **OK**. The assistant follows the same rules as Chapter 8: one job per window, dialogs opened with `Await f.ShowDialogAsync(Me)`, the app's logic in modules with unit tests, and the Mac layout guide. If a plan puts a list, an editor and every command on one form, ask for a change with **Change…**, for example "edit the tasks in a dialog".

A project that starts from Sub Main. When **Project > Startup Object** is **Sub Main** (see Chapter 9), the assistant is told that the app starts from `Sub Main` in its module and keeps it. The plan's startup form is then the main window that `Sub Main` shows: it must be a form the project already has, with its current name. A plan that invents a new startup form is refused with MCB4721, and the message says why; ask for a change that marks an existing form as the main window.

Every stage builds something. Each stage must add or change code or form layouts. The assistant cannot run, test or look at your app, so a plan never has a stage whose only purpose is verification, manual testing or review, such as "Verify the complete application". Unit tests are written in the stage that adds the code they test. McBasic™ runs Check and the tests after each stage, and you review the changes and try the app. If the assistant still proposes a stage that only verifies, tests or reviews, the plan is refused with an MCB4720 message that names the stage, for example "AI plan stage 'Verify the Complete Application' only verifies, tests or reviews the app, which the AI cannot do." Nothing is created. Send your request again, for example with "plan it without a verification stage".

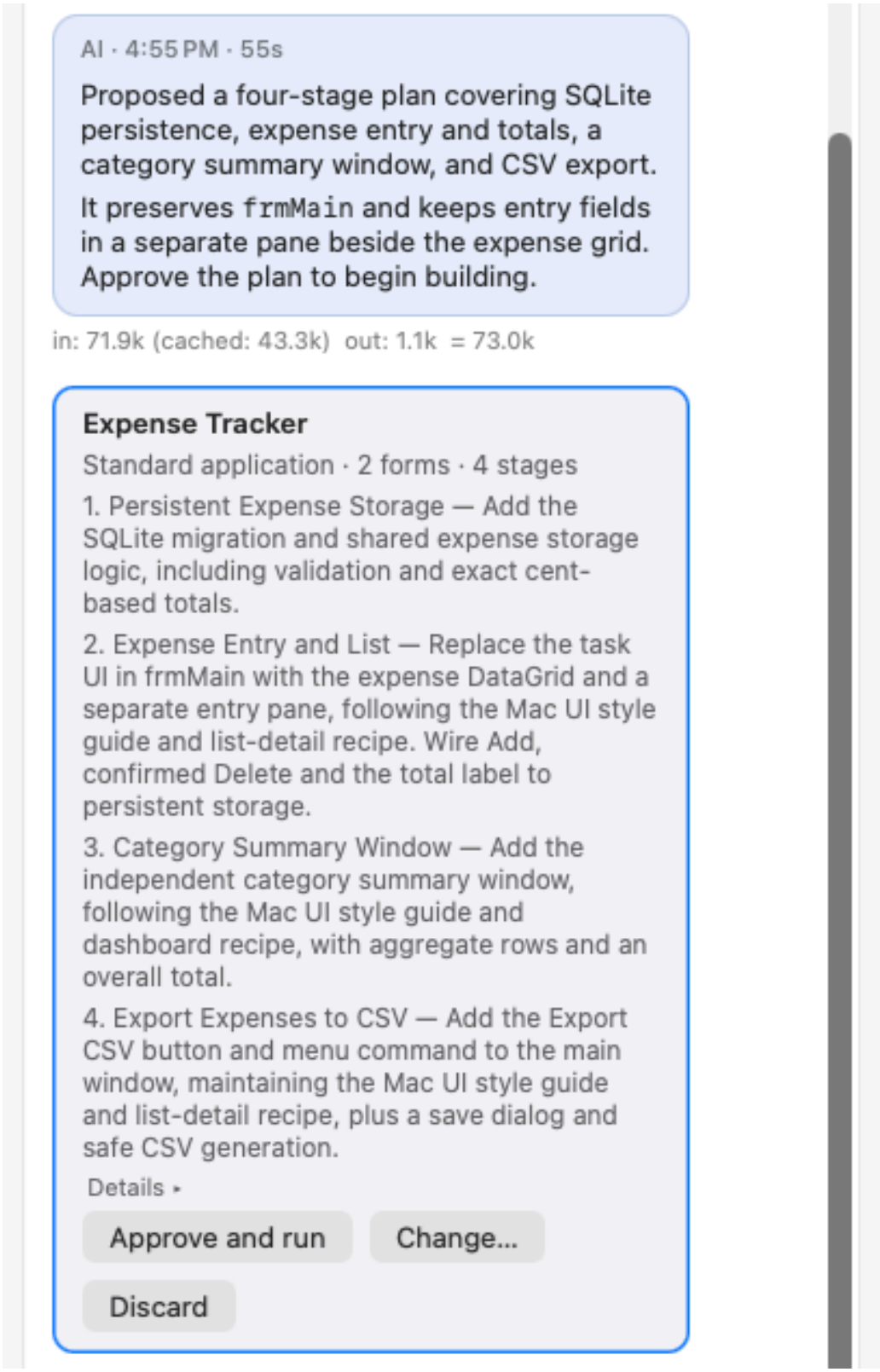


Figure 16.5: The plan card

What happens after Approve and run

When you press **Approve and run**, the plan card compacts to "Plan approved · 3 stages", with the application name and a progress line such as "1 of 3 applied". Below it, one stage card appears for each stage.

Each stage then runs on its own, with no question to you:

1. **Generate.** McBasic™ takes a fresh snapshot of the project as it is now, with the current chips, and asks the assistant for this stage only. The request carries the current stage in full and a one-line summary of the other stages, so it stays small even for a large plan.
2. **Check.** McBasic™ compiles the proposed stage in an isolated copy of your project. Your open project is not touched. It also measures the forms whose layout the stage adds or changes against the Mac layout guide (see "Layout notes" below). With External Codex or a Responses API endpoint, the assistant sees these layout advisories while it is still answering and may send one corrected stage at once; the status line then reads "McBasic™ found layout advisories in the proposal; the AI may correct them now." Layout advisories never make a stage fail. Three exceptions must be fixed like Check errors: a caption clearly clipped by its control (MCB4787), a control outside the form or container that holds it (MCB4771), and a control glued to an edge of the form or container, closer than 8 to it (MCB4799). A problem that was already in your own layout before the stage, on a control the stage did not touch, stays a note.
3. **Repair.** If Check fails, the chat adds a neutral note such as "Refining: McBasic™ found 2 issues; the AI is correcting them (attempt 2 of 3)", with the diagnostics under **Details**, and sends them back to the assistant. A stage has three attempts in all. When a later attempt passes, the chat notes "Corrected on attempt 2".
4. **Apply.** When Check passes, McBasic™ applies the stage to the open project as one Undo action, and the stage card shows **Applied**.
5. **Next stage.** The next stage whose earlier stages are done starts at once.

The chat informs you; it does not ask. A plan stops only for one of the reasons in "When a plan pauses", and every pause tells you why and what to press.

A new form is finished in its own stage. When a stage creates a form, the assistant gives it its complete code, its controls, their event bindings and its properties in that same stage, so the form works as soon as the stage is applied. A module the stage creates gets its complete code in that stage too. The later stages see both (see "Chips during a plan").

While a stage generates or is being checked, the code editor is read-only, as it is during any AI request. The status line shows what is happening, for example "Generating approved AI stage “Layout”. Press Stop to cancel." or "Running local Check for AI stage attempt 1 of 3…".

Stage cards

A stage card's title is the stage number and title, for example `2. Logic`. Its status line follows the stage:

Status	Meaning
Queued	The stage waits for the stages before it.
Generating…	The assistant is writing the stage. The elapsed time follows, for example `Generating… 12 s` or `Generating… 1 min 15 s`.
Checking (attempt 1 of 3)	McBasic™ is checking the proposal. The number is the attempt.
Repairing (attempt 2 of 3)	Check found problems; the assistant is correcting them. **Details** shows the diagnostics.
Checked	Check passed, but the stage is not applied yet: it waits for a dependency decision, or for **Retry** after the change could not be applied.
Ready to apply	Ask me first mode: Check passed, and the stage waits for **Apply** or **Discard**.
Applying…	McBasic™ is applying the stage to the open project.
Applied	The stage is in the project, as one Undo action. The card has a green border and a check mark before the word.
Failed	Check failed three times. The card has a red border and a cross before the word; a failure card follows.
Discarded	You discarded or skipped the stage; nothing of it was applied.
Undone	The stage was applied and then undone.

While a stage works, its card shows a small moving bar and one activity line under the status, updated in place (it never adds lines to the chat):

- "Reading the project…" while McBasic™ reads the current project for the stage;
- "Connecting…" until the assistant answers;
- "Thinking…" while the assistant reports that it is working;

- "Looking up: component DataGrid" while it reads McBasic™ documentation (the `looked up:` lines in the chat stay as before);
- "Writing the change… (12,400 characters)" while text arrives. The number counts only the characters actually received; McBasic™ never estimates progress, so a stage that sends its change in one piece stays at "Thinking…";
- "Writing: frmMain, modTaskStore" when the change arrives, with the forms and modules it writes;
- "Checking…" while McBasic™ checks the proposal;
- "Repairing, attempt 2 of 3: MCB2001 cboPriority.SetFocus" while the assistant corrects what Check found, with the first issue;
- "Layout advisories: 3 to fix" while the assistant may still correct layout notes.

If the assistant sends no sign of activity for a minute, the line becomes, for example, "Still working; last signal 1 min 20 s ago", and the time grows. This is not an error: you can wait, or press **Stop**. McBasic™ stops the request itself after 10 minutes without news.

Under the activity line, a smaller line counts the stage so far, for example `1 min 12 s · in 12.4k · out 740 · 1 lookup`, and **Details ▸** keeps the stage's activity as a timeline (see "Live activity").

An applied stage card shows a line such as **3 files changed**, followed by a small triangle. Click it to open the changes made by the stage; click it again to close them. The section "Read the changes" explains what you see.

Layout notes. When a checked stage still has layout advisories, its card lists them under a line such as `1 layout note from McBasic (advice, not errors; fix them in the designer or ask the AI):`, one line per note, each ending with its code, for example • `frmMain: txtName (Left 12, Top 20, Width 200, Height 32) is 12 from the left edge of the 640 x 400 form; keep 20 between controls and the window edges (move or resize it). (MCB4772).` The stage is applied like any other. Fix a note in the designer, or ask the assistant in a new message. The notes are not stored with the chat, so they do not come back after McBasic™ restarts. The codes are:

Code	What McBasic found
MCB4770	two visible controls in the same container overlap
MCB4771	a control reaches outside the form, Frame, Panel or TabPage that holds it
MCB4772	a control is closer than 20 (but at least 8) to a window edge (docked controls, ToolBar and StatusBar are exempt)

Code	What McBasic found
MCB4773	a label beside a field does not have the field's Top and Height
MCB4774	buttons in one row have a different Top or Height
MCB4775	an **OK**, **Save** or **Done** button is not the default button, or **Cancel** is not the cancel button
MCB4776	one form holds a list and three or more input fields: editing belongs in a dialog
MCB4777	a DataGrid column header shows a raw column name with _: give the column a header in **Columns** (see Chapter 12)
MCB4778	a procedure the AI wrote has more than 15 lines and no comment
MCB4779	a form, module or class the AI added has no header comment
MCB4783	a MsgBox the AI wrote asks a question (Yes/No, OK/Cancel…) without an icon: add vbQuestion, vbExclamation or vbCritical
MCB4787	a caption is clearly clipped by its control (more than a quarter too long); the AI must fix it
MCB4788	too many command buttons in one area (more than 6, or two rows of three or more): move commands to the menu bar (Menu Editor items), a ToolBar or a side panel
MCB4789	buttons in one row have different widths
MCB4790	a command button, single-line field, check box, option button or toggle switch is lower than 28 or narrower than 20: the AI must give it Height 32
MCB4791	labels or fields in one column are almost, but not exactly, aligned
MCB4792	a main window has four or more command buttons and no menu bar (Menu Editor items)
MCB4793	a dialog has no default button or no cancel button
MCB4794	more than one default button

Code	What McBasic found
MCB4795	the default button has a destructive caption such as Delete or Erase
MCB4796	a font smaller than 10 points
MCB4797	text and background colours with too little contrast
MCB4798	a caption may be clipped (up to a quarter too long); a note only
MCB4799	a control touches an edge of the form, or of the Frame, TabPage or bordered or coloured Panel that holds it (closer than 8); the AI must move it inside, 20 from the window edges and 8 from a container's edges (docked controls, ToolBar, StatusBar, Line and Shape are exempt)

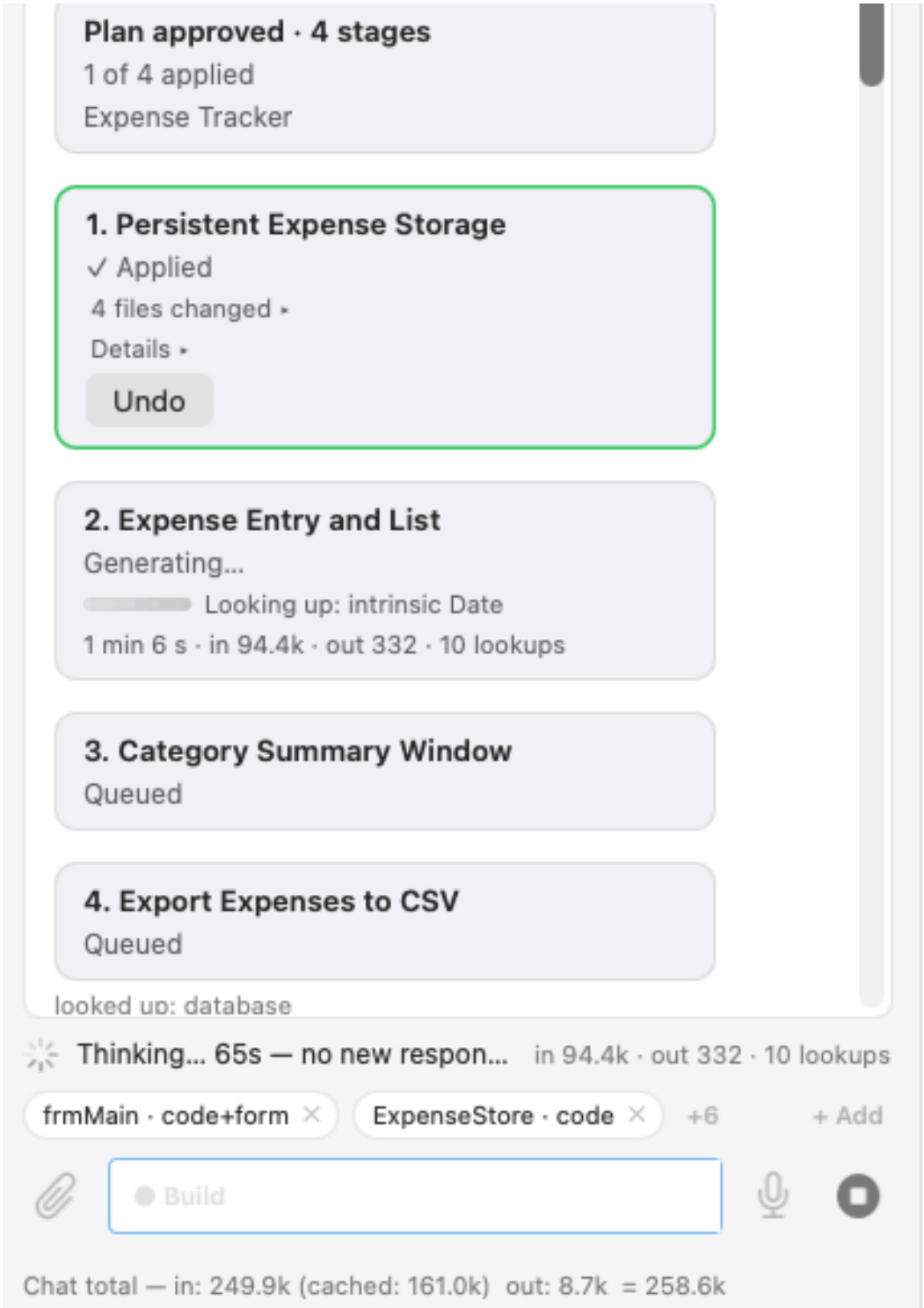


Figure 16.6: Stage cards while a plan runs

Read the changes

The changes of a stage or a change card are shown file by file. Each block starts with the file's path and what happened to it, for example `Forms/frmMain.bas · modified`.

BASIC and other text files appear as a line diff in a monospaced font:

- added lines are green and start with `+`;
- removed lines are red and start with `−`;
- two unchanged lines before and after each change are shown in grey for

context;

- a longer run of unchanged lines is shortened to one line such as

`14 unchanged lines`.

A block with more than 20 lines shows the first 20 and a **Show all** button. Database migration files and other text files are shown in the same way.

Forms are shown as a list of readable operations, not as JSON:

```
Added CommandButton `cmdEquals` · Caption = "=" · Click event
Changed form frmMain · Caption = "Calculator"
Changed CommandButton `cmdEq` · renamed from `cmdEquals` · Caption =
"Equals"
Removed Label `lblOld`
```

A new form starts with `Added form frmDetails`, and a removed one shows `Removed form frmDetails`. When a control changes more than six properties, the line names six and ends with, for example, `3 more`. **Details** under the list opens the raw layout difference, as green and red JSON lines, for when you need the exact values. If McBasic™ cannot read a layout, the line says `Changed form frmMain (the layout could not be summarized; see Details)`. A property that was removed appears as, for example, `ToolTipText removed`. When the layout changed in a way the list does not describe, such as the order of the controls, the line says `Changed form frmMain (no property or control change; see Details)`.

If a text file changed only in its whitespace or line endings, its block says "No visible line changes (whitespace or line endings)."

Ask me first

If you set **Apply AI changes:** to **Ask me first**, each stage stops after a successful Check:

- the stage card shows **Ready to apply**, with its changes already open;
- the plan card shows **Waiting for Apply**;
- **Apply** applies the stage as one Undo action, and the plan continues with

the next stage;

- **Discard** drops the stage. Nothing of it is applied, the card shows

Discarded, and the plan continues with the next stage.

A discarded stage counts as settled, so the stages that depend on it still run, on a project that does not contain its work. If a later stage needs what you discarded, expect its Check to fail. To end the plan instead, choose **Stop plan** on the plan card: the waiting stage is not applied, and no further stage runs.

If you change the project while a stage waits at **Ready to apply**, McBasic™ does not apply the old proposal when you press **Apply**: it was checked against a version of the project that no longer exists. Your edit stays, a restart note appears, and the stage is generated again on the current version.

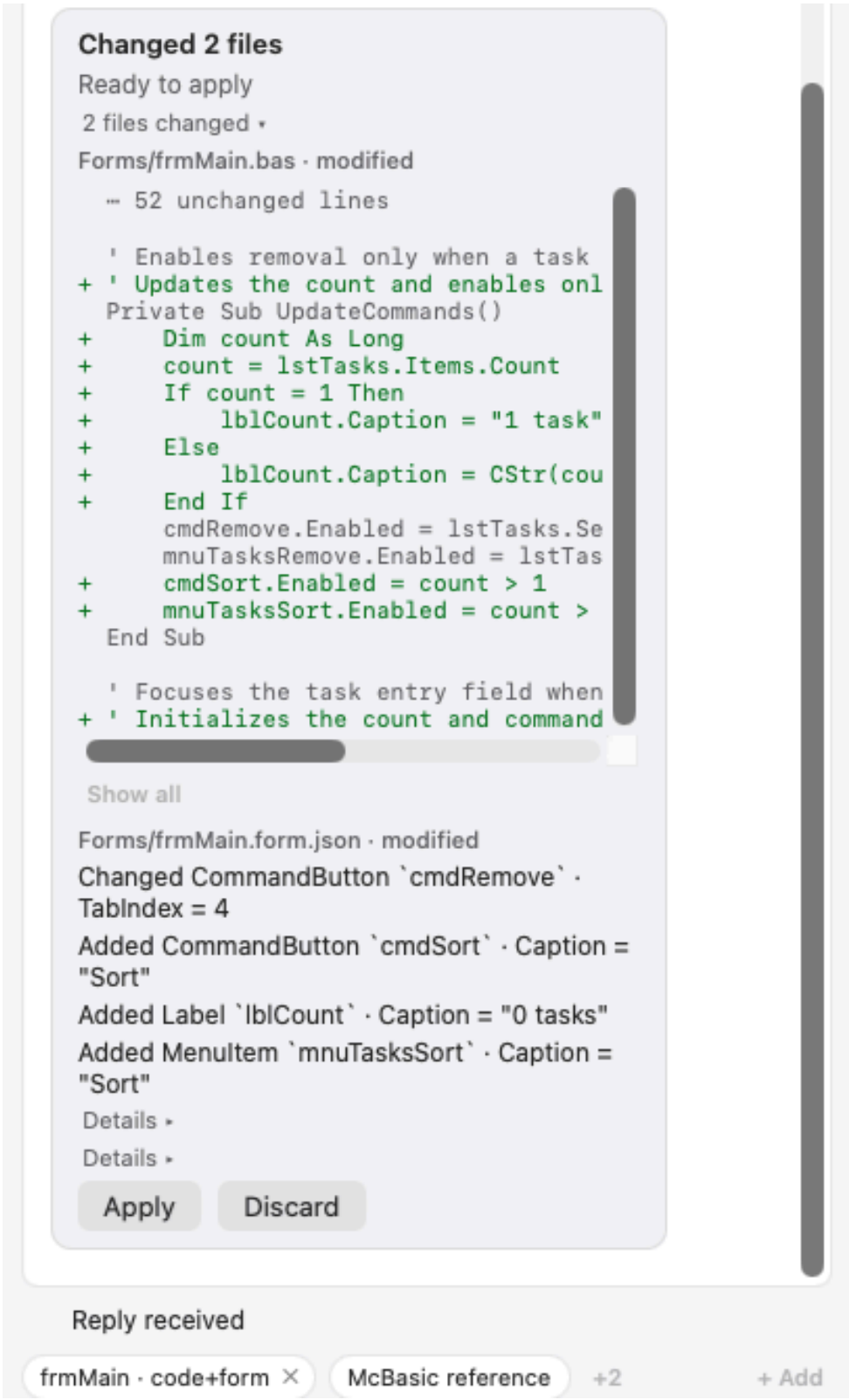


Figure 16.7: A stage waiting in Ask me first mode

Plugins and migrations: dependency cards

A stage can add a reference to a trusted plugin, or a database migration file. McBasic™ never applies those without telling you. After the stage passes Check, a dependency card appears, titled for example "Stage 2 adds a plugin reference", "Stage 2 adds a migration file", or "Stage 2 adds a plugin reference and a migration file". Its lines give the exact details:

- for a plugin, one block per package with `Package:`, `Version:` and

`SHA-256:` lines, followed by "These exact packages passed the candidate's offline lock/cache/trust Check. Approval adds project references only; it does not install a package.";

- for a migration, one block per file that starts with `Migration 1:` and the

file's path in the project, then its SQL, followed by "Approval adds these SQL files to the project. It does not execute a migration or access a database."

- **Approve** accepts them. The card notes "Approved." and the stage is

applied (or, in Ask me first mode, waits for **Apply**).

- **Reject** applies nothing. The card notes "Rejected. The stage was not

applied." and the plan pauses with "Dependency rejected. The stage was not applied." Press **Resume** on the plan card to generate the stage again, for example after you asked for a version without the plugin, or **Stop plan**.

While the card waits, the plan card shows **Waiting for dependency approval**.

If **Approve plugins and migrations automatically** is on, the stage is applied without stopping, and the dependency card has no buttons. It notes "Approved automatically (AI Settings)." so that you can still see what was added.

When Check fails three times

The repair notes are neutral, not errors: most stages pass on the second or third attempt. When all three attempts fail:

- the stage card shows **Failed**, with a red border;
- the plan card shows **Paused: Check failed**;
- a red failure card appears, titled for example "Stage 3 did not pass Check

after 3 attempts". Its **Details** are open and list McBasic™'s real diagnostics, one per line, for example `MCB1001 [Main.bas:2]: Expected a parameter name.`

Nothing of the failed stage was applied. The failure card offers three buttons:

- **Retry stage** generates the stage again from the current project, with

three new attempts. Change the chips or the project first if the diagnostics point to something missing.

- **Skip stage** gives the stage up and continues with the next one. The

skipped stage counts as settled, so the stages that depend on it still run, on a project that does not contain its work. The skipped stage card shows **Discarded**.

- **Stop plan** ends the plan. Stages already applied stay applied.

A failure card also appears, titled for example "Stage 3 changed no file", when the assistant's change set passes Check but leaves every file as it was. It offers only **Retry stage** and **Stop plan**: the stage cannot be skipped, because the stages after it need its work. Its **Details** say "The AI's change set changed no file, so nothing was applied and the stage is not done. Retry stage to ask again." Nothing is applied, no Undo step is added, and the next stage does not start, so a later stage never runs without the work this one should have done. Press **Retry stage** to ask for the stage again.

Builds 819 and 820 could record such a stage as applied. When McBasic™ reopens a paused plan that holds one, the plan waits on the earliest of them with the same failure card, titled for example "Stage 4 changed no file", and the line "Stage 4 was recorded as applied but changed no files; Retry stage to run it again." It is the last entry in the chat. The stages after it keep the files they applied: a stage that was complete is not generated again over its own files, and a stage that was continuing by slices continues with its slice once **Retry stage** has done the stage it was waiting for.

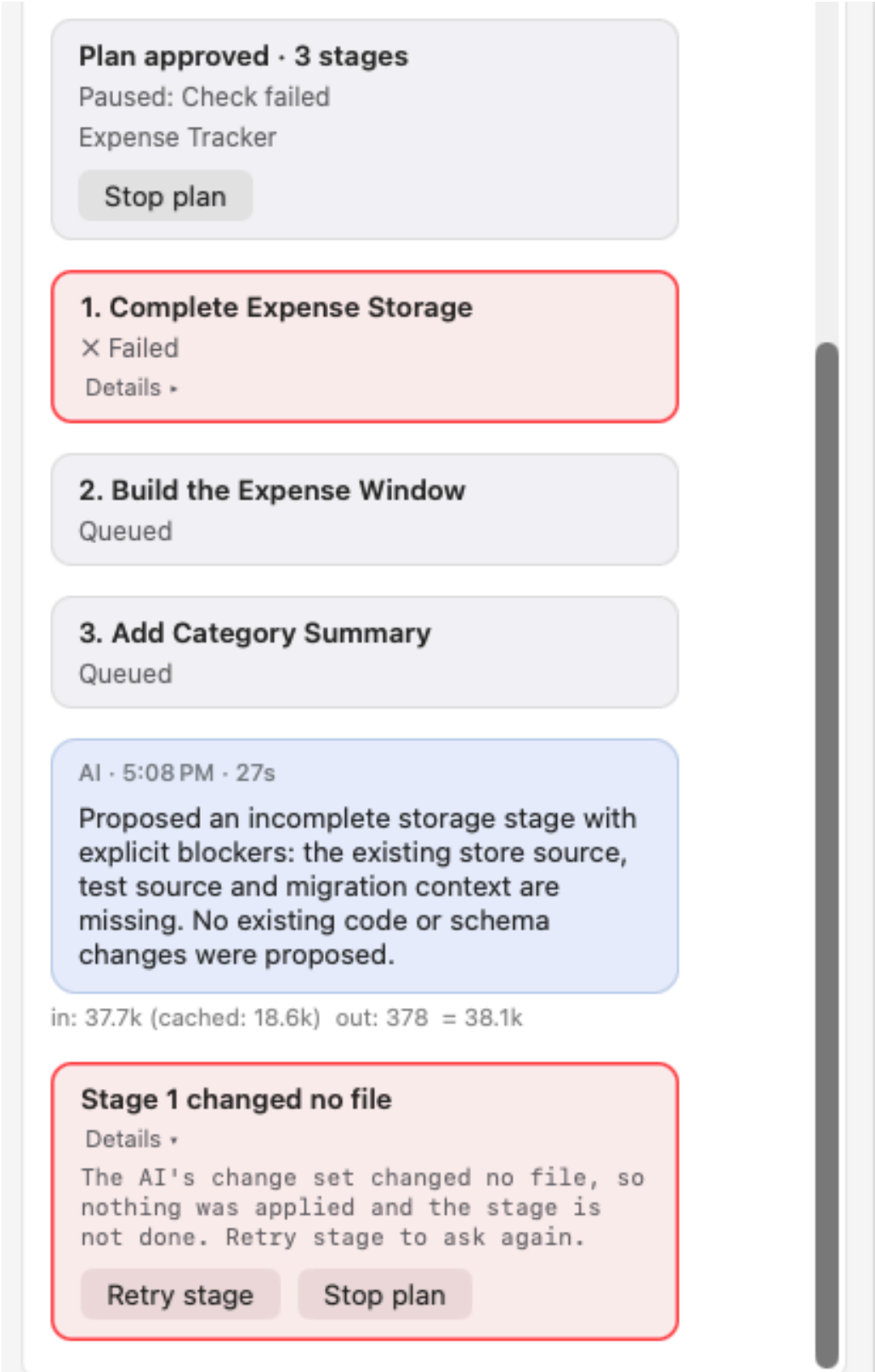


Figure 16.8: A failure card

When the assistant says a stage is incomplete

Sometimes the assistant can do only part of a stage, for example because the next step needs a document that is not in the chips. It then says so in its proposal, with a short list of what is missing. McBasic™ never guesses this from the text of the reply.

"Incomplete" means only that code, layout or a file the stage asks for is missing from the proposal. Running Check and the tests, reviewing the changes, approving a migration or a plugin, and trying the app are done by McBasic™ and by you after the proposal, so they never make a stage incomplete. The assistant is told so, and is told that verification it cannot run is not missing work: a stage that asks to verify a behavior is met by the code and the unit tests that cover it.

When that proposal passes Check, McBasic™ applies the part that is ready, as one Undo action, like any stage. What happens next depends on the list.

The stage continues by itself, one missing piece per slice

A large missing piece, such as a whole new window, does not fit in the same reply as the rest of the stage. So when the assistant lists what is missing, the plan does not pause: McBasic™ asks again for the same stage at once, on the current project, and this time for **only the first piece on the list**. This request is a *slice*. The assistant is told that the other pieces come in later slices, that work the plan gives to another stage is never missing, and that it may say the stage is complete only when nothing of it remains.

While a slice runs:

- the chat adds a line such as "Stage 4 · continuing: A Library window lists the saved puzzles (slice 1 of 3)";
- the stage card shows two lines, for example "Stage 4 · slice 1 of 3" and "Stage 4 · continuing: A Library window lists the saved puzzles";
- **Details** ▸ on the stage card records each slice, for example "0:42 Slice 1 of 3: A Library window lists the saved puzzles".

A slice is an ordinary stage request: it uses the stage's chips and request budget, it is checked and repaired in the same way, and it waits for **Apply** in **Ask me first** mode. A dependency card, a permission card, an over-budget or a failed request pause it as they pause any stage, and **Retry** or **Resume** then runs the same slice again. Each slice that is applied is its own Undo action.

If the slice finishes the stage, the stage card shows ✓ **Applied** and the plan goes on. If the slice applies its piece and the list still has pieces, the next slice asks for the new first piece. A stage runs at most **3 slices** after its first attempt.

When slicing stops

McBasic™ stops continuing the stage, and the plan waits for you, when:

- the 3 slices have run ("Stopped after 3 slices.");
- a slice makes no progress: the assistant replies without a change, or its change changes no file and lists the same pieces again ("No progress on: A Library window lists the saved puzzles."). Nothing is applied for that slice;
- a slice does not pass Check after 3 attempts ("Slice 2 did not pass Check after 3 attempts."); **Details** ▸ on the card shows the last Check report;
- you press **Stop** ("Stopped during slice 2.");
- you keep changing the project while a slice runs ("The project kept changing during slice 2.");
- you press **Discard** on a slice in **Ask me first** mode ("Slice 2 was discarded.");
- you undo a slice ("Slice 2 was undone.").

The stage keeps everything already applied, and:

- the stage card shows ⚠ **Applied, incomplete**, with a red border;
- the plan card shows **Paused: the stage is incomplete**;
- a red card appears, titled for example "Stage 2 is incomplete". Its first line says why slicing stopped, as listed above. Then it lists what is still missing, one point per line, for example "• The task editor needs the storage module's applied source."

If the assistant marks a stage incomplete without saying what is missing, the card appears at once, with "The AI marked this stage incomplete without saying what is missing." and no slice runs. A single change outside a plan never continues by slices: its card appears at once.

The plan waits for you. The card offers three buttons:

- **Retry stage** asks the assistant for the whole stage again, on the current project. The part already applied stays, and so do the documents it created, which are now in the chips. Add a chip first if the card says that something is missing. If that answer is incomplete again, slicing starts afresh, with up to 3 new slices.
- **Continue anyway** keeps the stage as it is and runs the next one. The stage stays incomplete.
- **Stop plan** ends the plan. Stages already applied stay applied.

The applied part is an ordinary change: **Undo** on the stage card, or **Edit > Undo**, removes the last one. Undoing the stage's first attempt pauses the plan with **Paused after Undo**. Undoing a slice removes only that slice: the stage goes back to the list that slice was asked about, and the incomplete card says "Slice 1 was undone." The pause, its list and the reason slicing stopped are saved with the chat: after you quit and reopen McBasic™, the same card is back with the same buttons. A plan you quit while a slice was running comes back paused, with **Resume**; **Resume** runs that same slice again, still counted as the same slice.

A plan with an incomplete stage is never reported complete. If you press **Continue anyway** on its last stage, or the last stages finish after an incomplete one, the plan ends. The chat then says "AI application plan ended with an incomplete stage: Stage 2 “Main window and task editor” is not complete. Ask the AI to finish it, or finish it yourself." With more than one, it names them all, for example "AI application plan ended with incomplete stages: Stage 2 “Data” and Stage 4 “Windows” are not complete. Ask the AI to finish them, or finish them yourself." The plan card says, for example, "Plan not complete · 2 of 3 applied · 1 incomplete".

When you change the project during a plan

You can edit the project while a plan waits for approval, while it is paused, and between your own sessions. McBasic™ never overwrites your edits. Every generation, including **Resume** and **Retry**, starts from a fresh snapshot of the project as it is at that moment, so the assistant always builds on your latest version.

A checked proposal is applied only to the exact version of the project it was generated and checked against. If the project changed in between, for example because a file changed on disk, or because you edited the code while a stage waited at **Ready to apply**, McBasic™ drops the proposal and restarts the stage on the current version. The chat says so in a restart note, for example:

> `You changed the project during stage 3: restarting stage 3 on the current version.`

The restarted stage begins again at attempt 1. The restart note is information only; **Stop** stays available.

A stage restarts at most twice in a row on its own. If the project changes a third time before the stage can be applied, the plan pauses with "The project kept changing while this stage ran. Resume regenerates it on the current version." Press **Resume** when you have finished editing. The count starts again after a stage is applied and after any card button you press.

An edit you make while a plan is paused is not a restart: it is simply what the next stage will see when you press **Resume**.

Stop a plan

While a stage runs, press **Stop** in the composer, or Esc:

- the running stage is cancelled without touching the project;
- stages already applied stay applied;
- the plan pauses with "Stopped. Stages already applied stay applied; the

running stage changed nothing." and the plan card shows **Resume** and **Stop plan**.

Resume generates the stopped stage again, from the current project. **Stop plan** ends the plan for good: the plan card becomes, for example, "Plan stopped · 2 of 3 applied", and no further stage runs. The plan card offers **Stop plan** in every pause,

whatever the reason: also while a stage waits for **Apply** or for a dependency decision, and after a failed Check (where the failure card has its own **Stop plan** too). A checked stage that was waiting is not applied.

When a plan pauses

A plan pauses only for a reason you can act on. The plan card's status line names the reason, a line under it explains it, and the buttons are on the card that needs your decision. **Stop plan** is on the plan card in every pause:

Status on the plan card	Why	Where to act
Waiting for Apply	Ask me first mode: a checked stage waits for you.	Stage card: **Apply** or **Discard**; or plan card: **Stop plan**.
Waiting for dependency approval	The stage adds a plugin reference or a migration file.	Dependency card: **Approve** or **Reject**; or plan card: **Stop plan**.
Paused: Check failed	Three Check attempts failed, or the checked change changed no file.	Failure card: **Retry stage**, **Skip stage** (not when the stage changed no file), or **Stop plan** (also on the plan card).
Paused: the stage is incomplete	The assistant applied part of the stage, and McBasic™ stopped continuing it by slices (the card's first line says why), or the assistant did not say what is missing.	Incomplete card: **Retry stage**, **Continue anyway**, or **Stop plan** (also on the plan card).
Paused	You pressed Stop; you rejected a dependency; the project kept changing; McBasic™ was closed during the plan; the provider in AI Settings changed or none is selected; a new message set aside a waiting checked stage; or the remaining stages depend on stages that cannot run.	Plan card: **Resume** or **Stop plan**.
Paused after Undo	You undid the last applied stage.	Plan card: **Resume** or **Stop plan**.
Paused: no context	The chat has no chips.	Add chips with **+ Add**, then plan card: **Resume**; or **Stop plan**.

Status on the plan card	Why	Where to act
Paused: the AI request failed	The provider failed, timed out, or replied without a usable change.	Error card at the bottom of the chat: **Retry** or **Stop plan** (after McBasic™ reopens the chat, on the plan card).
Paused: the request is over budget	The stage request is larger than the request budget, even with the context space it leaves unused. Nothing was sent.	Follow the fix in the message, then the error card at the bottom of the chat: **Retry**; or **Stop plan** (after McBasic™ reopens the chat, on the plan card).
Paused: the change could not be applied	Applying the checked stage failed, for example because a previous save needs recovery.	Plan card: **Retry** or **Stop plan**.
Waiting for your permission	The stage changes something outside the chat's scope (see "Add to Chat and the scope").	Permission card: **Yes**, **No**, or **Yes, and add to scope**; or plan card: **Stop plan**.
Paused: outside the scope	You answered **No**, and the assistant cannot finish the stage without that change.	Error card at the bottom of the chat: **Retry** or **Stop plan** (after McBasic™ reopens the chat, on the plan card).

The line under **Paused** is one of these:

- "Stopped. Stages already applied stay applied; the running stage changed nothing."
- "Dependency rejected. The stage was not applied."
- "The project kept changing while this stage ran. Resume regenerates it on the current version."
- "Paused when McBasic™ closed. Resume to continue."
- "Paused while you were in another chat. Resume to continue."
- "Paused while this chat was closed. Resume to continue." (you closed the project and opened it again without quitting McBasic™)
- "AI provider changed to External Codex (ChatGPT sign-in). Resume to continue with it." (with the name of the provider now selected)
- "No AI provider is selected. Choose one in AI Settings, then press Resume."
- "The checked change was set aside by your new message. Resume regenerates

this stage on the current project."

- "Remaining stages depend on stages that cannot run." In that case **Resume**

runs the waiting stage anyway; **Stop plan** ends the plan.

- "The change waiting for your permission was set aside by your new message. Resume regenerates this stage on the current project."
- "Every item of this chat's scope left the project, so the plan paused: without a scope the AI may change anything. Press Resume to continue without a scope, or add items to the chat first."

A stage that was **Waiting for your permission** when McBasic™ closed comes back as **Paused** with "Paused when McBasic™ closed. Resume to continue.": **Resume** generates the stage again, and the card asks again if it still needs to.

The line under **Paused: outside the scope** names what you did not allow:

- "The AI cannot finish this stage without changing what you did not allow: SaveTask in modTaskStore. Add it to the scope and press Retry, or stop the plan."
- when the refused change is a new document, a plugin or a migration, which cannot be a scope item: "The AI cannot finish this stage without changing what you did not allow: … Press Retry and the AI asks your permission again, or stop the plan."

The line under **Paused: the AI request failed** gives the provider's message, or one of these:

- "The AI replied without a change for this stage. Nothing was applied."
- "The AI answered for another stage. Nothing was applied."
- "Context changed during this stage. Nothing more was sent and the project was

not changed."

- "The AI sent nothing for 10 minutes, so McBasic™ stopped this request. Nothing was changed. Retry, or split the stage if it keeps happening."
- "This AI request ran for 60 minutes, so McBasic™ stopped it. Nothing was changed. Retry, or split the stage."

For **Paused: the change could not be applied**, **Retry** applies the same checked stage again when the project has not changed since; otherwise it generates the stage again on the current version. The project is never left half-applied.

Resume and **Retry** both continue from the current project and the current chips, so you can fix the cause first: add a chip, shorten a document, or finish an edit.

While a plan is running or paused, the chat does not start a second one. A new plan proposal is refused with "A plan is running. Press Stop before asking for another plan." or "Resume or stop the paused plan before starting a new one." While a single change is being applied, a plan proposal is refused with "A change is being applied. Wait for it to finish, or press Stop, before asking for a plan."

A button that no longer applies to the plan's current state does nothing, and the chat says "This action is no longer available." While an AI request runs, only **Stop plan** and **Undo** act; any other card button says "Wait for the current AI request or press Stop."

When the plan is complete

When every stage is applied, discarded, or skipped, the chat shows the completion card with a check mark and a **Run** button. When every stage was applied, it says "All 3 stages applied" (or "The only stage is applied"). If some stages were discarded or skipped, it counts only the applied ones, for example "Plan complete · 2 of 3 applied". The status line says "AI application plan completed. Save Project to keep the generated application."

A plan with an incomplete stage never shows the completion card or this line: it ends as described in "When the assistant says a stage is incomplete".

Plan an app with /plan

Plan mode is the deliberate way to start an application: the assistant asks before it plans, and nothing changes in the project until you approve a draft. In Build, a large request can still lead straight to a plan card, as described in "Plans and stages".

1. **Start.** Choose **Plan** with the pill in the message field and describe your idea, or send `/plan` followed by the idea. If you send `/plan` alone, the assistant first asks what you want to build, with a few typical kinds of app as options. In a project that already has forms and modules, the assistant starts from the project map (see "What the chat sends in each mode").
2. **The interview.** The assistant asks one question at a time, each on a question card: what the app is for and who uses it; its data (what is stored and which fields); its windows (the main window, dialogs, Settings, About); its rules and validation; its tests; and its look. It stops when it knows enough, normally within about 8 questions. Answer with the buttons, with **Other…**, or in the message field.
3. **Stop asking, draft the plan.** While the interview runs in Plan mode, this button is on the question card and above the message field. Its tooltip reads "End the interview: the AI drafts the plan from what you decided so far". It sends "Stop asking. Draft the plan now from what we decided." and the assistant drafts the plan from the answers so far. Whatever you were typing stays in the message field. If you switch the chat to another mode, the interview pauses and the button hides; switch back to Plan to continue it.
4. **What we decided.** After your first answer, the chat shows a card titled "What we decided · 1 decision". It is updated in place after each answer. Open **Details** ▸ to read the decisions, one per line, for example "• Where should tasks be stored? → SQLite database".

5. **The draft.** The assistant proposes the plan as a card titled "Draft 1 · " followed by the app's name, with the windows (their purpose and the data they receive and return) and the stages under **Details**, and two buttons: **Approve and run** and **Discard plan**.
6. **Correct the draft in words.** Type what you want changed, for example "merge stages 2 and 3", and send it in Plan. The assistant proposes the whole plan again as "Draft 2 · …", and the older draft loses its buttons. The draft under review goes with each Plan request, so the assistant revises that draft and not an older one.
7. **Approve and run.** The chat records the decision "Approved plan: …", switches to Build with the note "Switched to Build to run the plan", and the stages run on their own, as described in "What happens after Approve and run".

Discard plan ends the interview: the card becomes "Plan discarded", and the **What we decided** card stays, read-only. Your decisions are kept and go with later requests. The next Plan message starts a new interview, with drafts numbered from 1 again.

In Plan mode the assistant has no tool that changes the project. The interview, the decisions and the drafts are saved with the chat, so you can quit McBasic™ in the middle of an interview and continue later.

Finish an imported project with AI

After you import a Visual Basic 6 project (Chapter 19), the migration report lists the work that is left. Tick entries in the report window and press **Finish with AI…**. The chat opens in Plan mode, the declared files of the ticked entries appear as scope chips, and the text of the entries fills the message box. Nothing is sent until you press Send, and the first send asks for your consent as for any request. The assistant then receives the text of the ticked entries and the files of those chips, nothing else: the `Import/` folder itself is never sent beyond the entries' text. From then on the chat behaves as for any plan: request budgets, staged plan, approval and per-stage Undo. One request takes at most 40 entries that touch at most 32 files.

Single changes outside a plan

A smaller request, such as a change to one procedure or one form, does not need a plan. The assistant's reply then contains a single change, which goes through the same steps as a stage, without a plan card:

• a change to one BASIC procedure has already been validated by McBasic™'s compiler, so it is applied (or waits for **Apply**) at once;

• a change to forms is validated in the same way;

• a change that touches several files is checked like a stage, with up to three

attempts.

The change card's title says what changed, for example:

Kind of change	Example title
One BASIC procedure	"Changed procedure `cmdOK_Click` in frmMain"
A BASIC document, when the procedure has no name McBasic™ recognises	"Changed frmMain"
One form	"Changed form frmMain"
Several forms	"Changed 2 forms: frmMain, frmDetails"
A set of files	the assistant's own summary, or "Changed 3 files"

The card shows the changes, as described in "Read the changes", and follows **Apply AI changes:**:

- with **Automatically, with Undo**, the change is applied, the card shows

Applied, and **Undo** is on the card;

- with **Ask me first**, the card shows **Ready to apply** with **Apply** and

Discard.

If a multi-file change fails Check three times, its failure card is titled "The change did not pass Check after 3 attempts", with **Retry** and **Discard**. A single change pauses for the same reasons as a stage, but it has no plan card: the change card itself shows the reason and its buttons, **Resume** and **Discard** (for example after Stop or a provider change) or **Retry** and **Discard** (after a failed request, an over-budget request, or a change that could not be applied). A change that adds a plugin reference or a migration file shows a dependency card titled "This change adds …". If the project changes before the change can be applied, McBasic™ sends your original request again on the current version and notes "The project changed before this change was applied: regenerating it on the current version."

While a plan waits for approval, is running, or is paused, a single change is not applied. The chat says why:

- "Approve or discard the proposed plan before applying another change."
- "Wait for the plan to finish, or press Stop, before applying another change."
- "The approved plan is paused, so this change was not applied. First press …, then send the change again.", naming the buttons of the card that waits. For example, on an incomplete stage: "First press Continue anyway or Stop plan, then send the change again." After Stop it names **Resume** or **Stop plan**; after a failed request,

Retry or **Stop plan**. This happens also when the assistant gives the change the paused stage's own identity.

A change you ask for in the chat is always a single change, never part of the plan: the assistant is told to give it its own identity and never to ask you for the plan's or the stage's IDs.

The menu commands **Edit > Review BASIC Procedure Patch…** and **Edit > Review Structured Form Patch…** are not part of the AI chat. They keep their own review windows for patches you paste yourself (see Chapter 4).

In a chat with a scope (see "Add to Chat and the scope"), McBasic™ applies only a change it can check against the scope. If the assistant replies with a direct edit of a procedure or a form instead, nothing is applied, and the chat says "This chat has a scope, so McBasic™ applies only a change set it can check against it. The AI's direct edit was not applied."

What the chat sends in each mode

Each request is built from a few parts. Each part appears as a chip in the chips row, so you can see what goes with your message before you press Send.

Chip	What it sends	Can you remove it?
McBasic reference	McBasic™'s reference index. The assistant looks up the details it needs. It holds no project data.	No
Project map	The names in your project: each form with its controls (names and types) and event handlers, the Public procedures of modules and classes (names and signatures) with the first sentence of their header comments, the tests, the migration and resource file names, and the project settings. No procedure bodies and no code. When the request is near its size budget, McBasic™ sends it in three steps: with comments, then signatures only ("Project map (signatures)"), then names only ("Project map (names)").	Yes, with its **×**, or with the **Project map (names and signatures)** checkbox at the top of the **+ Add** list

Chip	What it sends	Can you remove it?
A document chip, for example `frmMain · code+form`	The document's full source or layout, as in "Context chips: what the chat may send".	Yes
A scope chip, for example ◎ `cmdSave (frmMain)`	What the AI may change, and what it reads to do it (see "Add to Chat and the scope").	Yes
Decisions (n)	Your answers to questions, the approved plan, and what you allowed or added to the scope, word for word.	No
Diagnostics (n)	The diagnostics of the latest Check, Run or Test (Fix only).	No
Last Run output	Fix only: the last 2,000 characters your app printed in its latest Run and how it ended (normal exit, exit code, or Stop), marked for the AI as program output, not instructions. A new Run or any edit replaces it. A message McBasic™ sends again by itself (after a card answer or after the Check before Fix) never carries output newer than your last Send.	Yes, with its **×**. It then stays out of this chat, also for later Runs, until you turn on **Last Run output (Fix)** in **+ Add**
Last 6 exchanges, **Last 2 exchanges** or **Interview**	The most recent part of this chat, word for word. Older messages stay visible in the chat but are not sent. In a Plan interview, the whole interview.	No

The **Project map** chip is on in every new chat. Like the document chips, your choice to remove it lasts while McBasic™ is open, and the map is read again from the project before each request. Removing it is the way to send nothing about the project at all.

Each mode sends a different recipe:

Mode	What a request sends
Build, one message	reference, project map, decisions, your chips and scope, the last 6 exchanges
Build, a plan stage	reference, project map, decisions (with the approved plan), the stage's instructions and a summary of the other stages, the chips and scope; no chat history
Plan	reference, project map, the draft you are reviewing, decisions, your chips and scope if any, and the whole interview
Ask	reference, project map, decisions, your chips and scope if any, the last 6 exchanges
Explain	reference, project map, the scope or the chips it explains, the last 6 exchanges
Fix	reference, project map, the diagnostics and the documents they name, or, with no diagnostics, your bug report and the documents it names; the last Run output; the scope if any; the last 2 exchanges
Review	reference, project map, the target: the scope, or the chips, or the open form when there is neither; no history
Apply selected (Review)	reference, project map, the target, and the findings you ticked
A repair after a failed Check	only the errors, the failed proposal and the documents involved, with the scope

Every request carries the chat's scope when it has one, whatever the mode. In Ask, Plan and Explain the AI only reads it: those modes cannot change the project.

Fix works from diagnostics or from your report. When the latest Check, Run or Test reported errors, Fix works on them: when you choose Fix, or type /fix, McBasic™ adds the documents they name to the chips, with a note such as "Added for Fix from the latest diagnostics: frmMain · code+form. Remove a chip to keep it out." The Diagnostics (n) chip, and the exact-content preview if it is open, follow every new Check. Stopping your app with Stop is not an error and is never sent as one. If the diagnostics are missing or older than your last edit, Send first runs a Check ("Checking before Fix…"), even when the automatic Check is off; when the Check adds chips you have not seen, the chat says "Check finished. Press Send to send your message." and the next Send sends; if the Check cannot start, it says "Check could not start; press Send again." and sends nothing.

When there are no diagnostics (a bug in how the app behaves, while Check passes), describe it like a tester: what you did, what you expected, what happened instead. On the first Send, McBasic™ adds as chips the documents your report names and the forms and modules their code uses (at most 8), with the note "Added for Fix from your report: …" ending "Review the chips, then Send; remove a chip to keep it out." Nothing is sent yet: check the chips and press Send again. A document counts as named when you write its exact name ("frmNewGame", "SudokuGenerator"), or its name as separate words without its prefix ("New Game" for frmNewGame); a name shorter than four letters only counts when written exactly. With no named document, the startup form is added. With a scope, the scope is the target and nothing is added. Retry behaves the same way: if it adds chips, it says "Review the chips, then Retry" and sends on the next Retry.

Documents earlier stages created. In an approved plan, each stage receives the forms and modules that earlier stages of the same plan created, as chips when they fit the budget. When the Project map chip is on, the stage also receives the map entries and hashes of created documents it cannot send whole, and the chat says so, for example "Sent the project map entries and hashes of frmSolver and SudokuGenerator, which earlier stages created." When the Project map chip is off, nothing about those documents is sent beyond the chips you can see.

How the assistant designs windows. Before it lays out a form, the assistant reads McBasic™'s Mac style guide and the recipe for the kind of app it is building (list and detail, data entry, document, settings, utility, dashboard, game, dialog, search or wizard). The layout checks in the table below then measure what it built.

When a fix needs another file. If the errors McBasic™ is about to send, or the errors a repair must correct, are in a file the chat does not send or its scope does not cover, McBasic™ asks first instead of letting the AI try blind. A card such as "To fix these errors the AI needs to read and change Tests/SudokuGeneratorTests.bas (MCB2102 at line 71). Allow?" names exactly the files it would send: for an error in a form's layout only the `.form.json`, for a whole form both files. **Allow** adds exactly those (and to the scope, when the chat has one) and continues; **Deny** says file by file what happens, for example "Details.bas stays read-only (already sent); Helper.bas will not be sent; their errors will remain." A failed Check that only leads to this card does not use up one of the three attempts. The AI can also ask for a file it needs; the card then shows its reason, and **Allow** sends your message again with the file. If you edited your message or changed the mode while the card waited, nothing is sent and the chat says "Your message changed while the card waited; press Send to send it." A file you denied is not offered again in that chat.

Review and Apply selected. Review asks the assistant for findings: bugs, layout problems against the Mac guidelines, and improvements. If the chat has no chips and no scope when you choose Review, McBasic™ adds the open form as the target, with the note "Added for Review: frmMain · code+form (the open form). Remove the chip to keep it out." The reply ends with a **Review findings** card, for

example "Review findings · 3 findings", with one checkbox per finding, such as "[bug] cmdSave_Click saves an empty title". Tick the findings you want and press **Apply selected**: only those go to the assistant, as a change request, and the result comes back as a change card like any other. When the reply does not list its findings in the expected form, it stays ordinary text.

When a request does not fit the budget, McBasic™ leaves parts out, in this order, and adds a note for each:

1. the oldest exchanges: "Left out to stay within the budget: the oldest 2 exchanges of this chat.";
2. the project map's details: "Left out to stay within the budget: the project map's details (form and module names kept).";
3. the chips that are not the work target, the most recently added first, for example "Left out to stay within the budget: frmAbout · code+form.";
4. the rest of the map: "Left out to stay within the budget: the whole project map, form and module names included."

Your message, the decisions, and the work target are never left out: the scope, a stage's documents, the documents Fix's diagnostics name, and what Review or Explain works on. If they alone do not fit, nothing is sent, and the message explains what to shorten (see "Over budget").

The first line of **Show exact content** names the mode and these parts in order, for example `Ask: McBasic reference, project map, decisions, frmMain · code+form, last 6 exchanges`, so you can check the recipe before you send.

Add to Chat and the scope

Add to Chat tells the assistant exactly what it may change. The items you add form the chat's **scope**. While the scope is not empty, every change the assistant proposes, in Build, in Fix, in Review's **Apply selected**, and in every stage of a plan, is checked against it by McBasic™'s toolchain before it is applied.

You add items from the right-click menus. A Control-click opens the same menus:

Where you right-click	Item	What joins the scope
A control in the form designer (Chapter 5)	**Add to Chat**	that control, or every selected control when you click inside the selection
A member of a control array in the form designer	**Add Array to Chat**	every member of the array and its shared event procedures; a member's own **Add to Chat** shows those procedures read-only

Where you right-click	Item	What joins the scope
The empty part of a form in the designer	**Add Form to Chat**	the whole form
A form, module, class or test in the project tree (Chapter 4)	**Add to Chat**	the whole document
Inside a procedure in the code editor (Chapter 6)	**Add Procedure to Chat**	that procedure

These items are unavailable while the assistant works, and while a Check, Build or Run is running.

Adding opens the chat panel. Each item becomes a scope chip with a target symbol, for example ◎ `cmdSave (frmMain)`, ◎ `txtCell(5) (frmMain)` for one member of a control array, ◎ `txtCell array (frmMain)` for the whole array, ◎ `frmMain` or ◎ `SaveTask (modTaskStore)`, and the chat notes "Added to this chat's scope: cmdSave (frmMain). The AI may change only what is in scope; anything else needs your permission." Items may come from several forms. Adding an item that is already in the scope does nothing. A chat holds at most 32 scope items; beyond that the chat says "A chat holds at most 32 scope items. Remove one, then add it again."

Scope item	The AI may change	The AI only reads
Control	the control's properties and layout, and its event handlers: the procedures named `cmdSave_Click` and so on, and those the layout binds to its events	the form-level variables and form procedures those handlers use, and the signatures of the module procedures they call. A handler that another control or the form also uses is only read.
Control array	every member's properties and layout, and the array's shared event procedures, such as `txtCell_Change(Index As Integer)`	what its handlers use, as for a control. A new member, added with another Index, is outside the scope, so McBasic™ asks with a permission card.
Form	the whole form: its layout and its code	—
Module, class or test	the whole document	—
Procedure	that procedure's body and signature (a property's **Get**, **Let**	the declarations it uses, as for a control

Scope item	The AI may change	The AI only reads
	or **Set** part is a separate item, but the AI cannot yet replace a property part on its own: it has to rewrite the whole document, so McBasic™ asks your permission for that document with a permission card)	

Chips follow the project. A scope chip follows the control it names, so renaming the control renames the chip. A control array's chip follows the array's name: when you rename the array, its chip leaves the scope, and you add the array again with **Add Array to Chat**. When an item leaves the project, for example because you deleted the control, McBasic™ stops before the next Send and says "Scope changed: cmdSave (frmMain) left this chat's scope. Review the chips, then Send." Check the chips and press Send again. During a plan, an item that left is removed with the note "Removed from this chat's scope because it is no longer in the project: …", and a plan whose whole scope left the project pauses (see "When a plan pauses").

Remove an item with the **×** on its chip. With VoiceOver the chip reads, for example, "Scope: cmdSave (frmMain)", and its **×** reads "Remove from scope: cmdSave (frmMain)". The scope is saved with the chat, and it stays until you remove its chips. Each change of the scope is kept as a decision: "Added to scope: …", "Removed from scope: …", or "Removed from scope (no longer in the project): …".

The permission card

When a proposed change touches something outside the scope, McBasic™ does not apply it. It asks you with a permission card:

> *The AI wants to change also: SaveTask in modTaskStore. Allow?*
> *Reason given by the AI: the Save button must store the task.*

What counts as outside the scope: a change to another control, another procedure or another document, a new control on a form whose scope is only some of its controls, or a new form, module, plugin reference or migration.

Button	What happens
Yes	The change is allowed for this one Check. If the Check fails and the assistant repairs the change, the card asks again.
No	The change is not applied. McBasic™ asks the assistant to "Proceed without them, or

Button	What happens
	explain in one short paragraph why you cannot."
Yes, and add to scope	The change is allowed, and what it touches joins the scope as chips, with the usual "Added to this chat's scope: …" note. For a property, each of its parts (for example **Get** and **Let**) gets a chip.

A new form, module, plugin reference or migration, and a procedure that does not exist yet, cannot be a scope item. **Yes, and add to scope** then allows them once only, with the note "Allowed once only, because it cannot be added to the scope: …".

Your answer is kept as a decision: "Allowed once: …", "Not allowed: …", or "Allowed and added to scope: …" (followed by ". Allowed once: …" for what could not join the scope).

If the assistant cannot do the work without the change you refused:

- a plan stage pauses with **Paused: outside the scope** and **Retry** and **Stop plan** (see "When a plan pauses");
- a single change ends with the note "The AI could not make this change without changing what you did not allow: SaveTask in modTaskStore. Nothing was applied."

If you send a new message while a permission card waits, the change it asks about is set aside (see "When a plan pauses").

Shared projects. The scope, the permissions and the decisions about them belong to this Mac. When you open a chat that McBasic™ did not write here, for example one that came inside a project package from someone else, it has no scope and no permissions. Your answers to questions and the approved plan stay readable as decisions.

Live activity

Every running request shows what is happening, on the status row above the message field and, during a plan, on the stage card. The line changes in place, and on the status row it never wraps: in a narrow panel the counters keep their place and the status ends with "…"; its tooltip shows the full text.

- "Reading the project…", "Connecting…", "Thinking…";
- "Looking up: database" while the assistant reads McBasic™'s reference;
- "Writing: frmTaskEditor, modTaskStore" with the forms and modules a change writes;

- "Checking…";
- "Repairing, attempt 2 of 3: MCB2001 cboPriority.SetFocus";
- "Layout advisories: 3 to fix".

Counters. A small line counts the work: on a stage card, for example `1 min 12 s · in 12.4k · out 740 · 1 lookup`, and on the status row, which already shows the elapsed time, `in 12.4k · out 740 · 1 lookup`. `in` and `out` are the tokens the provider reported so far, and they show – until it reports any. The lookups count the different reference topics read. A stage card counts the whole stage, including its Checks and repair attempts. While McBasic™ itself reads the project or runs Check, the status row shows no counters, because no provider is involved.

Silence. After a minute without any sign from the assistant, the line says, for example, "Still working; last signal 1 min 20 s ago". The request is still running: wait, or press **Stop**. McBasic™ stops a request itself after 10 minutes without any sign.

Details ▸. A stage card, and a change card, keep the activity as a timeline. Open **Details ▸** to read it, one line per step with the time since the stage began, for example "0:03 Looking up: forms" and "1:15 Checking…", above the Check report a repair works on. The timeline is saved with the chat, so it is still there after you quit and reopen McBasic. The counters are not saved.

The activity comes from what McBasic™ and the provider already exchange: no extra request is sent to the provider to show it.

Undo, Redo, and saving

Every applied stage, every applied slice of a stage and every applied change is exactly one Undo action, including every file it created or changed. A file a stage adds outside the code and layouts, such as a migration `.sql` file, is removed by Undo only before you save; after **Save**, Undo leaves that file on disk.

Undo on a card. The card of the most recent AI change has an **Undo** button. It does the same as **Edit > Undo**, once. The button is enabled only while that change is still the last thing that happened to the project:

- if you edit the project after the AI change, or if a Redo is pending, the

button is disabled and the card says "Use Edit → Undo to step back.";

- older cards have no Undo button and show the same sentence.

Use **Edit > Undo** to step back through older changes in order.

If you press a card's **Undo** when it cannot act, nothing is undone, McBasic™ beeps, and the chat says why:

Message	Why	What to do
"Use Edit → Undo to step back."	The AI change is no longer the most recent change.	Use **Edit > Undo**.
"This chat's project is no longer the open project."	The card belongs to a project that is not the one open in this window.	Open that project, or use **Edit > Undo** in its window.
"Wait for the current build or run to finish, then press Undo again."	A build, Check, or run is in progress.	Wait, then press **Undo** again.
"Finish or cancel the property you are editing, then press Undo again."	A property in the designer is still being edited and cannot be committed.	Press Return or Esc in the property, then press **Undo** again.
"Your property change was saved. Use Edit → Undo to step back."	A property you were editing was committed first; it is now the most recent change.	Use **Edit > Undo**: the first step takes back the property, the next one the AI change.

Undo during a plan. If you undo the last applied stage, with the card's **Undo** or with **Edit > Undo**, while the next stage is running, McBasic™ cancels the running stage without touching the project, marks the undone stage **Undone**, and pauses the plan with "Stage 2 was undone. Resume regenerates it on the current project." The plan card shows **Paused after Undo**, with **Resume** and **Stop plan**. **Resume** generates the undone stage again. The same happens if you undo the last stage after the plan completed. After **Stop plan**, an undo only marks the stage **Undone**.

Undoing one of your own edits is never treated as undoing the AI change. The chat recognises an undo of the AI change only while that change is the most recent one. If you edit the project after the AI change, then use **Edit > Undo** first on your edit and then on the AI change, the code goes back, but the card still shows **Applied** and the plan does not pause. Check the code before you continue in that case.

Redo. Edit > Redo puts an undone AI change back like any other edit. The stage card still shows **Undone**, and the plan stays paused. If you then press **Resume**, the assistant generates that stage again, on a project that already contains the redone code. If you want to keep the redone version as it is, press **Stop plan** instead.

Saving. Applying a change edits the open project, not the files on disk. Choose **Save** to keep it. Until you save, closing the project without saving discards the AI's changes like any other edit. McBasic™ does not save for you during a plan.

Resume a plan later

A plan's progress is saved with its chat: the plan, the state of each stage, and why it paused. A plan is never saved as running.

- **After you quit.** If McBasic™ closes while a plan runs or is paused, the plan is paused the next time you open the chat. Its plan card shows **Paused** and "Paused when McBasic™ closed. Resume to continue.", with **Resume** and **Stop plan**. Nothing runs or is applied until you press **Resume**.
- **After you switch chats.** If you left a chat while its plan waited at **Ready to apply**, for a dependency decision, or after a failed apply, its plan card says "Paused while you were in another chat. Resume to continue." when you come back. If you closed the project and opened it again without quitting McBasic™, it says "Paused while this chat was closed. Resume to continue." A plan already paused when McBasic™ closed keeps its "Paused when McBasic™ closed" line until you press **Resume**. Other pauses keep their own reason, as described below.
- **A stage waiting in Ask me first mode** is not kept across a restart: after

Resume, McBasic™ generates it again on the current project.

- **Edit > Show Paused AI Plan** opens the chat panel on the newest chat of the

project whose approved plan is paused, and the status line says "Press Resume on the plan card to continue the approved AI plan." If the project has no paused plan, the chat shows an error card: "This project has no paused AI plan to resume."

- **The pause message is kept.** The plan card shows the same reason and

message after you reopen the chat. A very long message, such as a provider error, is kept up to its first 4,000 characters and ends with "…". A stage waiting at **Ready to apply** or for a dependency decision comes back as **Paused**, because its checked proposal is not saved.

- **Check the chips before you press Resume.** Chips are not saved. A reopened chat proposes the document open in the editor, as a new chat does. It also adds the forms, modules, classes and tests that the plan's applied stages created, with the note "Added … to this chat's context: the AI created them.", except those you removed. **Resume** sends what the chips select. Add any other document with **+ Add**, and remove any chip you do not want to send, before you press **Resume**. If you remove every chip, the plan pauses with **Paused: no context**.
- **An incomplete stage stays paused.** After a restart, its card is back with why slicing stopped, the list of what is missing and **Retry stage**, **Continue anyway** and **Stop plan**. A slice that was running comes back as **Paused** with "Paused when McBasic™ closed. Resume to continue.", and **Resume** runs that slice again.
- **Resume uses the current provider.** Pressing **Resume** is your consent to

the provider and model now selected in AI Provider Settings. If they differ from the ones the plan started with, the stage is generated with the new one, as a new provider conversation. If no AI provider is configured at all, nothing is sent and the chat says "Choose an AI provider in AI Settings, then press the button again. Nothing was sent."; AI Provider Settings does not open by itself. If a provider is configured but its settings are incomplete (for example no model is chosen),

Resume opens AI Provider Settings first. When you complete them there, nothing is sent yet: the chat says "Provider settings updated. Press Resume again." Press **Resume** a second time to send with that provider. If you close the settings without completing them, the chat says "Choose a provider in AI Settings, then try again."

- **Chats from another Mac.** A plan approved on another Mac is never treated

as approved on yours. In a chat that arrived inside a project from someone else, that plan stays as text in the conversation, with no plan card and no **Resume**. A plan that was only proposed there, and not yet approved, keeps its plan card, and you can approve it on your Mac with **Approve and run**.

Errors in the chat

Every error of the AI chat appears in the chat, never in a separate window. Errors of a request (the provider cannot be reached, times out, or returns something McBasic™ cannot use) and errors of the chat's own menu commands, such as **Edit > Open Latest AI Activity** when there is nothing to open ("This project has no saved AI activity yet."), are shown as a red error card: a readable title, **Details** (collapsed, with the technical message), and **Retry** when the request can be sent again. A card's Retry is disabled once you have sent a newer message. If the chat panel was closed, McBasic™ opens it first.

During a plan, a failed request pauses the plan (see "When a plan pauses"). The plan card shows **Retry** and **Stop plan**. The chat also adds a red error card at the bottom, where you are looking, with the same two buttons, and scrolls to it. For a single change, the buttons are **Retry** and **Discard**. Once the plan runs again or ends, that card's buttons are disabled. Nothing is ever half-applied.

The only windows that can still appear around the chat are the system file picker for attachments, **AI Provider Settings**, and the confirmation before you delete a chat.

When a form change names something McBasic™ does not know, the diagnostic says exactly what is wrong, for example:

```
MCB4732 [frmMain/cmdEquals] property 'Text' is not a CommandButton
property
```

The part in brackets is the form and the member; the rest names the property, value, or event at fault. When a close match exists, the message ends with a suggestion such as "(did you mean ...?)".

McBasic™ also tells the assistant exactly which IDs to use in a change, for example `module:todostore` for a new module named TodoStore. The ID of a new module, test or resource follows from its name, so when the assistant sends a wrong one McBasic™ corrects it, together with every other change aimed at the same wrong ID, and the stage goes on. The kind of change always decides what is added: a new module stays a module even if its ID looks like a test's.

The ID is refused only when McBasic™ cannot tell what the assistant meant, for example when two new documents of one stage share the same wrong ID. The MCB4732 message then names the operation and the expected value, so the next repair attempt can fix it:

```
MCB4732 [op-add-helpers addModule] targetId must be 'module:modhelpers'
for module 'modHelpers' (got 'test:modTests'); addModule always adds a
module (addTest adds a test), and every operation of this change set on
the new module uses 'module:modhelpers'.
```

Run the result

When a plan is complete, choose **Run** on the completion card. It uses the same Run command as the toolbar and saves the project first. If a build, run, or debug session is already active, the chat asks you to wait for it and press Run again. Library plugins have no window to run, so their card has no Run button; use **Test** or the package build instead.

Then test the application as you would test your own code: valid and invalid input, every button, closing and reopening. A generated app has passed Check, not your tests.

Where chats are stored

Chats are saved automatically, after every message and every change of a plan; you never save a chat yourself. A chat is saved with its messages, times, titles, plan, stage progress and pause state, and with its chat mode, its decisions, a question that waits for your answer, its Plan interview and drafts, its scope and permissions, and the activity timelines of its stages. It is never saved with your chips' consent or with attachment contents.

- **Inside the project.** In a `.mcbproj` package, chats can be saved in the

`AI/Activities` folder inside the package. They then travel with the project: anyone you send the project to can read them.

- **On this Mac.** Otherwise, chats stay in McBasic™'s folder in your user

Application Support folder.

Choose per project in AI Provider Settings, in the **This project** section: **Save chats in this project**. Its help reads "Chats are saved automatically. This setting chooses where: inside the .mcbproj, so they travel with it when you share the project, or only on this Mac. It takes effect when you save the project." Only the location waits for the save: when you save the project, McBasic™ moves the existing chats. **Settings… > AI > Save AI chats in new projects** sets the default for projects you create later; it is on unless you turn it off. Legacy folder projects always keep their chats on this Mac.

Chats that arrive inside a project from someone else are readable, but McBasic™ never treats a plan approved on another Mac as approved on yours. A small local record lets McBasic™ recognise the chats it wrote itself.

A chat record can hold up to 2 MiB. Replies are always kept in full; if a chat grows beyond that size, the chat says "This chat is too large to save (over 2 MiB). The reply stays here; start a new chat to keep saving." The reply stays visible in the chat.

Privacy

AI features send requests only after an action of yours:

- when you press **Send**, **Retry**, or **Edit** and send;
- after you press **Approve and run** on a plan card: each stage, each repair

attempt, and each restart of that plan then sends its request automatically, with the chat's current chips, until the plan completes, pauses, or you stop it;

- when you press **Resume**, **Retry**, or **Retry stage** on a card;
- when you press a card button that continues an approved plan or a change: **Apply**, **Approve** on a dependency card, **Skip stage**, or **Discard** can start the next stage of the approved plan, or generate a stage again on the current project after it changed, and **Apply** on a single change can send its original request again on the current version;
- when you answer a question card: an option button, **Confirm**, **Send** under **Other…**, or Return on the recommended option;
- when you press **Stop asking, draft the plan**, or **Apply selected** on a Review findings card;
- when you press **No** on a permission card: McBasic™ sends your refusal to the assistant, which then works without the change. **Yes** and **Yes, and add to scope** send nothing themselves: McBasic™ checks the change on your Mac, and an approved plan then continues as before.

McBasic™ has no servers of its own that receive your code, and it contains no analytics or telemetry. Each request goes only to the provider you selected.

A request contains:

- your message, or for a stage the plan's instructions for that stage;
- the recent part of this chat that the chat mode sends (for example the last 6 exchanges, or a Plan interview), and the decisions;
- the BASIC source and form layouts selected by the chips;

- the project map, unless you removed its chip: the names of the project's forms, controls, handlers, tests and files, and the signatures of its `Public` procedures, with no code;
- for a scope item, what it lets the AI change and the declarations it reads;
- in Fix, the diagnostics of the latest Check, Run or Test;
- any attachments you added to that message;
- McBasic™'s reference index: instructions and reference text, with no project

data. Lookup answers are McBasic™ documentation only.

A request never contains the code of a document that is not in the chips or the scope, your other projects, files outside the project, your Keychain items, or your other chats. The forms, modules, classes and tests that the stages of a plan you approved created join the chips by themselves, each with a note in the chat, and you can remove them like any chip. With no document chips, no scope and no **Project map** chip, no project data is sent at all. A plan with no document chips pauses instead of generating a stage.

The provider's own privacy policy and terms apply to what it receives, and McBasic™ cannot control how a provider stores or uses it. API keys stay in the Keychain and are sent only to the endpoint you configured for them.

Before you share a project, remember that chats saved in the project go with it. Turn **Save chats in this project** off and save if you do not want to share them.

Limits

The request budget

Each request has a budget, counted in characters: one part for the prompt (your message, or a stage's instructions, plus the conversation) and one part for the project context the chips select. The prompt may also use the context space the request leaves unused: a stage that carries a long approved plan fits beside a small project. The context never takes more than its own part, and the whole request never takes more than the total. McBasic™ and its toolchain check the same numbers with the same rule, so a request the IDE accepts is never rejected by the toolchain for its size.

- **When McBasic knows the model's context window**, the budget is derived from

it, minus a reserve for the reply.

- **Otherwise**, McBasic™ uses the default: **240,000** characters in all, **40,000** for the prompt and **200,000** for the context.

In this build, no provider reports its context window to McBasic™, so the defaults apply to every provider unless you change **Request size:**.

LM Studio (local) requests carry at most **120,000** characters, **20,000** for the prompt and **100,000** for the context, even when **Request size:** is larger; a smaller **Request size:** still applies. That is about 40,000 tokens. A local model reads only as much as the context length it was loaded with, so load the model in LM Studio with a context length of at least 40,000 tokens, or lower **Request size:** to fit the length you chose.

Request size

Request size: in the **AI chat** section of AI Provider Settings sets the total for one request. McBasic™ splits it one part prompt to five parts context, and the prompt may use whatever part of the context the request leaves unused:

Request size	Prompt	Context
Automatic	the model's window when known; otherwise 40,000	otherwise 200,000
60,000 characters	10,000	50,000
120,000 characters	20,000	100,000
240,000 characters	40,000	200,000
480,000 characters	80,000	400,000
960,000 characters	160,000	800,000

Choose a smaller size to keep requests short, cheap, and quick. Choose a larger size when the project you share, or the plan a stage carries, does not fit. Above 240,000 characters the section warns: "Larger requests cost more, take longer, and may be rejected above the model's real limit." When McBasic™ knows the model's window, that window caps every choice. The grey line under **Request size:** says "Larger requests can cost more and take longer. Automatic uses the model's limit when known, otherwise 240,000 characters (40,000 / 200,000). The prompt may use context space the request leaves unused."

The budget counts the text McBasic™ selects: your message and the conversation for the prompt, and the BASIC source and layout text of the chips for the context. The request that actually travels to the provider also carries McBasic™'s framing and the structure around each document, so it is somewhat larger than these numbers.

Over budget

McBasic™ never truncates a request to make it fit, and it never sends a request over the budget. It says by how much the request is over, and what to do:

```
The stage request is 238,500 characters, 1,500 over the 237,000-character
budget; nothing was truncated or sent. Raise Request size in AI Settings
(for example to 480,000), or shorten the plan, then Retry.
```

For a prompt, the budget in the message is the prompt part plus the context space the request leaves unused: in this example, 40,000 plus the 197,000 characters a 3,000-character context leaves free. Chips and the context cannot make the prompt smaller, so the advice is a larger **Request size:** (the next size up from the current one) or a shorter plan. At **960,000 characters** the advice is "Request size in AI Settings is already at its largest: shorten the plan, then Retry."

The first words name the part that is too large, and the last sentence is the advice for it:

The message starts with	It ends with
"The request is …" (the first message of a chat)	"Shorten it before sending."
"The request is …" (the final check of any request just before it is sent)	"Raise Request size in AI Settings (for example to 480,000), or shorten the request."
"The conversation is …" (a message in an ongoing chat, with the conversation so far)	"Start a new chat with a shorter summary."
"The clarification is …" (your answer to the plan's questions)	"Shorten your reply or start a new chat with a summary."
"The stage request is …"	"Raise Request size in AI Settings (for example to 480,000), or shorten the plan, then Retry."
"The repair request is …"	"Retry the stage or split it."
"The selected context is …" (the chips alone)	"Remove a chip or lower the context."

Before any of these, McBasic™ first leaves out what the chat mode allows, as described in "What the chat sends in each mode", and notes each part it left out. When what remains still does not fit, the message starts with "The request is …" (your message, the decisions and the history) or "The context is …" (the documents, the scope and the map), and its advice depends on the mode, for example:

Mode	Advice for "The request is …"	Advice for "The context is …"
Build, Plan, Ask	"Shorten your message; the decisions and the current work are always sent."	"Remove a chip or lower the context; the scope and read-only files are always sent."
Fix	"Shorten your message; it and the diagnostics are always sent."	"Remove a chip; the documents the diagnostics name are always sent."

Mode	Advice for "The request is …"	Advice for "The context is …"
Explain, Review	"Shorten your message; it is always sent in full."	"Remove a chip; the target documents are always sent in full."
Apply selected	"Tick fewer findings; the ticked findings are always sent."	"Remove a chip; the target documents are always sent in full."
A plan stage	"Raise Request size in AI Settings (for example to 480,000), or shorten the plan, then Retry."	"Remove a chip, lower the context, or split the stage."

If the toolchain refuses a request for its size, the message carries the code `MCB4702`. When the context is too large it ends with "Remove a chip, lower the context, or split the stage."; when the prompt is too large it ends with the stage advice above, "Raise Request size in AI Settings (for example to 480,000), or shorten the plan, then Retry." Nothing was sent.

During a plan, this pauses the plan with **Paused: the request is over budget**, and the message is the line under it. Follow the advice, then press **Retry** on the plan card. A **Retry** that fails the same check again does not add another card: the chat keeps one card for it, moved to the bottom with fresh **Retry** and **Stop plan** buttons. There is no button that splits a stage: to split one, press **Stop plan** and ask the assistant for the remaining work in smaller steps.

A message you type is limited by the prompt part of the budget: "Enter an AI request up to 40,000 characters." for the first message of a chat, and a little less in an ongoing chat, which also carries the conversation, for example "Enter a chat message up to 38,000 characters." Shorten the message or split the request.

Other limits

Limit	Value
Documents in a project that uses the AI chat	1,000 (the project limit)
Documents one chat sends	64
Scope items in one chat	32
Options on a question card	2 to 6, plus **Other…**
Decisions kept in one chat	128
Plan drafts kept in one chat	8

Limit	Value
Check attempts for one stage or change	3
Automatic restarts of one stage in a row	2
Lines shown in one diff block before **Show all**	20
Messages in one chat	500
Size of one saved chat	2 MiB
AI request without any activity	10 minutes
One AI request	60 minutes

The reference index is sent in addition to the request budget. Replies are stored in full, however long they are.

Lookups, time, and cost

Every reference lookup makes a request longer, and how much that costs depends on the provider:

- **Responses API (for example OpenRouter).** Each lookup round sends the whole request again: your message, the context, and every earlier lookup and its answer. A stage that does many lookups therefore uses more input tokens and costs more. The `cached:` part of the token line is usually cheaper.
- **External Codex.** McBasic™ answers each lookup inside the same Codex turn and does not send the request again. Codex itself still reads the conversation again at each step it takes, so lookups add input tokens there too, mostly from its cache. The token line shows the total Codex reports for the request.

A complex stage can take several minutes; in one real run, a stage of a small To-Do application took about 3 minutes. While the assistant works, the stage card shows the elapsed time, for example `Generating… 2 min 40 s`, and the activity line under it (see "Stage cards"). Press **Stop** or Esc at any time: the plan pauses safely, and nothing of the running stage is applied.

McBasic™ stops an AI request that sends nothing for 10 minutes, and any AI request after 60 minutes. Everything the AI sends counts as activity: reply text, lookups, retry notices, and the signals External Codex sends while it is still thinking. A stage that keeps working is therefore never stopped at 10 minutes. When McBasic™ stops a request, nothing is changed: during a plan, the plan pauses with **Paused: the AI request failed**.

Troubleshooting

Codex is not found, or does not start

The chat shows "Codex could not start. Nothing was sent and your draft is preserved." Open AI Provider Settings with the gear, choose **Autodetect Codex**, choose **Save**, and send again. Make sure the ChatGPT app is installed in your Applications folder and opens normally. Never save a McBasic™ project inside the ChatGPT app or any other application: it damages that app's signature and macOS will then refuse to start Codex.

Codex is not signed in, or the model list is empty

The Model menu says "No models available — sign in, then Refresh". Choose **Sign In…** in AI Provider Settings, or **McBasic > Sign in to Codex with ChatGPT…**, then reopen the settings and choose **Refresh**. **Check Codex Status** reports whether McBasic™'s Codex is signed in.

The plan stopped and shows a status

Read the plan card's status line and the line under it, then use the table in "When a plan pauses". In most cases you fix the cause (add a chip, finish an edit, shorten the context) and press **Resume** or **Retry** on the plan card.

"This chat has no context chips…"

The plan reached a stage while the chat had no chips. Choose **+ Add**, select the documents the stage needs, and press **Resume** on the plan card.

"AI provider changed to …"

You changed the provider or the model in AI Provider Settings while the plan ran, so McBasic™ paused it before sending anything to the new provider. If the new choice is the one you want, press **Resume**. Otherwise set the provider back first, then press **Resume**.

"Provider settings updated. Press Resume again."

Resume opened AI Provider Settings because the provider's settings were incomplete, and you completed them there. Nothing was sent. Press **Resume** again to send with that provider.

"The checked change was set aside…"

You sent a message while a checked stage waited at **Ready to apply** or for a dependency decision. The proposal was dropped without being applied. Press **Resume** to generate the stage again on the current project.

A stage keeps restarting

The project changed while the stage ran, for example because another app wrote to a project file. After two restarts in a row the plan pauses with "The project kept changing while this stage ran…". Finish the other change, then press **Resume**.

"Project documents changed…"

A document was added, removed, or renamed after the chips were set. McBasic™ updated the chips: "Project documents changed. Review the updated context beside your message, then Send." Review the chips and press Send again.

MCB4732 after a stage

The generated layout or code does not fit McBasic. The bracketed part names the form and member, or the operation, such as `[op-3 addModule]`; the message names the value McBasic™ expected. If all three attempts fail, the failure card offers **Retry stage**; before you press it, include the affected form as a chip, or ask in the chat for a smaller step. **Skip stage** continues without the stage.

MCB4733: lookups without finishing

"MCB4733: The AI kept requesting reference lookups without finishing. Try again or rephrase." The assistant consulted McBasic™'s reference 500 times in one request without producing an answer. Nothing was applied. Send the message again, or rephrase it as a smaller step; during a plan, press **Retry** on the plan card.

"The AI sent nothing for 10 minutes…"

The provider sent no text and no sign of activity for 10 minutes, so McBasic™ stopped the request. Nothing was changed. Press **Retry** on the error card at the bottom of the chat, or on the plan card. If it happens again on the same stage, the stage is probably too large for one reply: press **Stop plan** and ask again for a plan with smaller stages, or remove chips the stage does not need.

"This AI request ran for 60 minutes…"

One request ran for an hour, so McBasic™ stopped it. Nothing was changed. Press **Retry**. If it happens again, split the stage.

The card's Undo is disabled

Something changed after the AI change, or a Redo is pending. The card says "Use Edit → Undo to step back." Use **Edit > Undo** as many times as needed. If the button is enabled but the chat answers with another message, see the table of Undo messages in "Undo, Redo, and saving".

A request is over budget

The message says by how much, and what to do. When the context is too large, remove chips the request does not need or share only the forms involved. When the prompt is too large (a stage carries the whole approved plan), choose a larger **Request size:** in AI Settings, for example 480,000 characters, or ask for a shorter plan. Then press **Retry**.

"Send a message to reconnect this chat, then try again."

The chat was shown from its saved record and has no provider connection yet. Send any message in the chat, then press the card button again. If the chat says "Choose a provider in AI Settings, then try again.", set up a provider first.

"Fix works on the diagnostics of the latest Check, Run or Test. Run Check first."

The chat is in Fix mode, or the message starts with `/fix`, and there are no diagnostics to fix. Choose **Check** (or run the app or its tests), then send again. To ask something else, choose another mode.

"Scope changed: … left this chat's scope. Review the chips, then Send."

An item of the scope is no longer in the project, for example because you deleted or renamed a procedure. McBasic™ removed its chip and sent nothing. Check the scope chips, add an item again with **Add to Chat** if you need it, and press Send again.

"Stop asking ends a Plan interview. Switch the chat to Plan, then press it again."

Stop asking, draft the plan works only while the chat, or the message you are typing, is in Plan mode. Choose **Plan** with the pill in the message field, then press the button again.

The mode cannot be changed

While a request runs, the pill is unavailable, and the chat says "Wait for the current AI request or press Stop, then change the mode." Wait for the reply, or press **Stop**.

Walkthrough: build a small calculator with the assistant

This exercise uses External Codex, but any configured provider works. The answers you receive will differ from the example; follow the steps, not the exact text. It assumes **Apply AI changes:** is **Automatically, with Undo**, the default.

This is the request you will send:

```
Build a simple calculator on frmMain. Two text boxes for the numbers,
four buttons for + - * /, and a label for the result. Show a friendly
message instead of an error when the input is not a number or when
dividing by zero. Use clear control names such as txtFirst and cmdAdd.
```

Follow these steps:

1. Choose **File > New Project…**, choose a Desktop application, and name it `Calculator`. (**Create with AI…** in the Welcome window creates a blank project and opens the chat in one step; if the open project has unsaved changes, McBasic™ first asks whether to save them.)
2. Press Shift–Command–L to open the chat. The chip `frmMain · code+form`

is proposed.

3. Type the calculator request shown above the steps.
4. Press Command–Return. Watch the status row and the `looked up:` lines.
5. If the assistant asks clarification questions, answer them and send.
6. Read the plan card: the goal line and each numbered stage. Open

Details if you want the full plan. If something is wrong, choose **Change…**, type the change after `Change the plan:` , and send.

7. Choose **Approve and run** on the plan card. The editor is read-only while

each stage is generated and checked; leave the project alone until the plan completes.

8. Watch the stage cards go from **Queued** to **Generating…**, **Checking**

(attempt 1 of 3), and **Applied**. If a "Refining" note appears, let the repair attempt run.

9. When the last stage is applied, open **N files changed** on a stage card and

read the BASIC lines and the form operations.

10. Choose **Run** on the completion card. Try 6 and 3 with every button, then

an empty box, a word, and division by zero.

11. Choose **Save**. Then choose **Undo** on the last stage card and notice that

the whole stage disappears and the plan pauses with **Paused after Undo**. Choose **Edit > Redo** to bring the code back, then **Stop plan** on the plan card, and save again.

If the plan pauses before it completes, read the plan card's status and act as described in "When a plan pauses".

Check your work

- The calculator runs and handles invalid input without an error dialog.

- You can name the documents that were sent, from the chips.
- No window opened while the plan ran; every step appeared as a card.
- One Undo removed one whole stage, and the plan paused.
- The project is saved, and its chat is in the chat menu.

Tips for good requests

- **Say what the user sees and does.** "A button that clears both boxes and puts the cursor in the first one" is better than "add clear logic".
- **Name things.** Give the control names you want (`txtFirst`, `cmdClear`). The assistant follows them, and your code stays readable.
- **Share only what matters.** Add the forms the change touches. Fewer chips mean shorter requests and less private text sent.
- **Ask for one feature at a time.** Small requests check on the first attempt more often and are easier to review and undo.
- **State the unhappy path.** Mention empty input, wrong types, and cancelled actions, as in the calculator prompt.
- **Ask for explanations.** "Explain this handler line by line" is a quick way to learn from generated code before you keep it.
- **Review what was applied.** Open the changes on each card. If you do not understand a change, undo it and ask why. Choose **Ask me first** if you prefer to read every change before it is applied.

Check your work

- You have a provider configured and know where its settings are.
- You can tell from the chips exactly what a request will transmit.
- You can choose a chat mode, and use a typed command for one message.
- You can answer a question card, and stop a Plan interview to get a draft.
- You can add a control to the chat's scope and decide on a permission card.
- You can open the exact-content preview and find its budget line and the reference index line.
- You know which setting decides whether AI changes are applied automatically.
- You can name the card that holds the buttons for each pause.

- You know that each applied stage is one Undo action and must be saved.
- You know how to continue a paused plan after you quit McBasic.
- You know where this project's chats are stored and whether they travel with it.

Tips and tricks

- **Know what the card buttons do. Retry** repeats the step, **Discard** drops it, and **Stop plan** ends the whole plan. Stages already applied stay applied.
- **Control heights.** The AI is told to use Height 32 for buttons, fields, check boxes and option buttons, and never below 28. If MCB4790 appears in the notes, set the height to 32 in the designer or ask the assistant to fix it.
- **Saving is disabled while an AI plan is working.** Wait for the plan to pause or finish, or use **Stop plan**, and then save.
- **Use Fix with Check errors or with a bug report.** Write the report as steps, expected result and actual result, for example `/fix the total is wrong`. Put only one bug in each Fix request.
- **Review the chips after your first Fix Send.** On the first **Send** of a report, Fix adds the documents you named as chips. Check them, then press Send again to send them.
- **Remove the Last Run output chip if you do not need it.** The chip carries your app's last Output lines. Click its **×** if they are not relevant; it stays out until you turn on **Last Run output (Fix)** in **+ Add**.
- **Let McBasic ask for the files a fix needs.** When the errors are in a file the chat does not have, a card such as "To fix these errors the AI needs to read and change Tests/SudokuGeneratorTests.bas (MCB2102 at line 71). Allow?" lists exactly the files it would send. **Allow** sends them; **Deny** says file by file what stays read-only and what is not sent.
- **Check LM Studio with Load Models before you save.** If you changed the LM Studio server port, type the same port in **Server endpoint:** and press **Load Models**: a filled menu proves the connection, and "loaded" shows which model answers at once. Prefer a model without "(tool use not reported)" for plans.
- **Paste the address your provider documents.** OpenRouter's documentation gives `https://openrouter.ai/api/v1`; paste it as it is into **Responses endpoint**. **Save** completes it to `https://openrouter.ai/api/v1/responses`. If **Save** shows a red message instead, read the field name at its start: that is the field to fix.
- **Pin a provider to keep the price stable.** OpenRouter can serve one model through several providers at different prices. Type the provider you want in **Providers:** (for example `Azure`) and turn **Allow fallbacks** off: a request then fails instead of

silently moving to a provider with another price. The **Activity** page on openrouter.ai shows which provider answered.

- **Ask for a menu bar in one sentence.** For example: "a File menu with New (Ctrl+N) and Open…, a View menu with Show Toolbar, and a right-click menu on the list".
- **Ask for a grid in one sentence.** For example: "a control array of 81 TextBox (txtCell) with one Change handler".
- **Finish an imported project one area at a time.** Tick the report entries of one area, such as data access, press **Finish with AI…**, and run that plan before you tick the next area (Chapter 19).

Chapter 17 — Build and publish an app

Building creates an application from the project files. Publishing prepares a copy that can be moved or delivered. Treat the two steps as related but not identical.

Before you build

Make sure that:

- the project is saved;
- Check succeeds;
- all forms and source files are included;
- resources are present;
- package inputs are exact and trusted;
- the target Mac architecture is the one you intend to test.

Test the output, not only the source

After a build, open the resulting app and repeat the important user actions. Check that it can find its resources, create its user-data path, open its database, and close without leaving a process behind.

Temporary generated files and application bundles are build products. The `.mcbproj` project (its `project.mcproj`, `.bas`, form layout, resources, and package declarations) is the input you should keep under version control. Generated files never live inside the project: McBasic™ builds in `~/Library/Caches/McBasic/Builds` and cleans that folder automatically, so a prepared build older than a day may simply be rebuilt.

Development output and customer output

A Debug app used on the development Mac is not automatically ready to distribute. A customer-ready output needs its own checks for signing, notarization, architecture, included assets, and installation on a clean Mac.

Do not describe an unsigned development build as a finished release.

Publish a local Mac app from the IDE

The current delivery command is **Project > Publish Local App** or the **Publish** command in the top strip. It first saves the project, validates the source and form layout, and prepares a self-contained unsigned `.app`. The standard Save panel then lets you choose the application name and destination. If you type a name without `.app`, McBasic™ adds the suffix.

The Save panel starts in the folder that contains the project, but you can choose another writable folder. The output must be outside the project and outside any application: McBasic™ refuses to write a published app inside a `.mcbproj`. If an app with that name already exists, McBasic™ asks before replacing it. The **Copyright** from **Project > Project Settings…** is written into the app's `Info.plist`, so Finder shows it in Get Info. The user-facing success message stays short: `Published <Name>.app.` Use **Project > Show Published App** to reveal the delivered copy.

The app gets the icon chosen with **Icon:** in **Project > Project Settings…** (see Chapter 2, "Choose the app icon"); without one it gets the McBasic™ icon. From a square PNG, McBasic™ makes the app's `AppIcon.icns` (sizes 16 to 1024 pixels) at every build, so Run, Debug and Publish all show your icon in the Dock and in Finder. An ICNS file is copied as it is. An ICO file has no Mac form: the Mac app keeps the McBasic™ icon and Publish shows the warning MCB2407. Changing the icon, or the image file, always makes the next build fresh. The user-facing success message stays short: `Published <Name>.app.` Use **Project > Show Published App** to reveal the delivered copy.

The Save panel also has a **Platform:** pop-up. **macOS** is the Mac app described here. **Windows · x64** and **Windows · ARM64** publish a single `MyApp.exe` for Windows from the same project, without a Windows PC. **Linux · x64** and **Linux · ARM64** publish a single executable `MyApp` for Linux in the same way. Chapter 20 describes the Windows steps and Chapter 21 the Linux steps: the first launch and how each service works there.

Building a plugin package works the same way. **Build** asks for the folder that receives the `.mbpkg`, and if a package with the same name is already there, McBasic™ offers **Move to Trash and Build** rather than deleting it.

The Free Edition launch splash

Every app you build with McBasic™ Free Edition opens with a launch splash. It is a small card in the middle of the screen, over your app's first window. It shows:

- on the left, a blue panel with the McBasic™ logo;
- **Made with McBasic™** and a **Free Edition** badge;

- your app's name, its version and build, and the **Copyright** from **Project > Project Settings…** when you set one;
- the line "Create your own apps for Mac, Windows and Linux." and a **mcbasic.com** link button;
- a thin blue bar along the bottom edge that empties over the five seconds.

The splash stays for five seconds and then fades out. It never stops your app. Your startup form, or your `Sub Main`, starts at the same time and shows behind the splash, and the keyboard stays with your form, so the person can type at once. A click on the card does not close it early. A click on **mcbasic.com** opens the website in the default browser and leaves the splash where it is. If the app quits before the five seconds are over, the splash closes with it.

The splash never covers your app's own dialogs. When the app opens a dialog while the splash is up (a `MsgBox`, an `InputBox`, an Open, Save, Color, Font or Print dialog, a form shown with `vbModal`, or the run-time error dialog), the splash steps aside until the dialog closes, then comes back for the rest of its five seconds. A login form that `Sub Main` shows with `vbModal` at start therefore appears alone, and the splash follows it.

The splash looks the same in every project, with the **macOS** or the **Classic** appearance. On the Mac it follows the light or dark mode of the Mac. It has no title bar, never takes the keyboard, does not appear in the **Window** menu, hides while another app is in front, and its accessible name for VoiceOver is "Made with McBasic, Free Edition"; VoiceOver can press its **mcbasic.com** link. Command-W and the keyboard cannot close it early. The app collects nothing and sends nothing because of the splash.

When it appears:

- in every app made by **Project > Publish Local App** or `mcbasic publish`, for the Mac, for Windows and for Linux, whatever the startup object (a form, an `MDIForm` or `Sub Main`);
- also when you choose **Run** or **Debug** in the IDE, and with `mcbasic run` and `mcbasic build`, because each of them builds a complete app that would work on its own if copied out of the build cache. Your form is usable behind the splash at once, so testing is not held up;
- never in a console app, because it has no window.

You cannot turn the splash off in the Free Edition: McBasic™ adds it to every app it builds, and no BASIC code or project setting removes it. The Pro edition, when it is available, will publish apps without the splash.

Apple may reject an app with a splash like this from the Mac App Store, because it looks like a demo. Plan to submit a Mac App Store app with the Pro edition.

Share a project

To give a project to someone else, zip the `.mcbproj` or send that single item. It carries no build output, because builds live in the build cache. If the project saves its AI chats in the project, they are in its `AI/Activities` folder and travel with it. To keep them private, open **AI Provider Settings**, clear **Save chats in this project**, and save the project before you share it: McBasic™ then moves the chats out of the package to this Mac. See "Project files and folder layout" and Chapter 16, "Work with the AI assistant".

This is a local publication workflow. It does not, by itself, sign, notarize, submit, or certify an application for distribution. Those steps belong to the release process for the target Mac and should be recorded separately.

A simple publish checklist

1. Copy the project or use a clean checkout.
2. Save and choose **Check**.
3. Choose **Run** and exercise the main workflow.
4. Choose **Project > Publish Local App**.
5. Pick a clear `.app` name and a destination you can write to.
6. Launch that published copy outside the IDE.
7. Test first launch, normal use, data storage, resources, and quit.
8. Inspect the output bundle and included licences.
9. Apply the public signing and notarization process if the release requires it.
10. Test the signed result on a clean Mac and record the exact version.

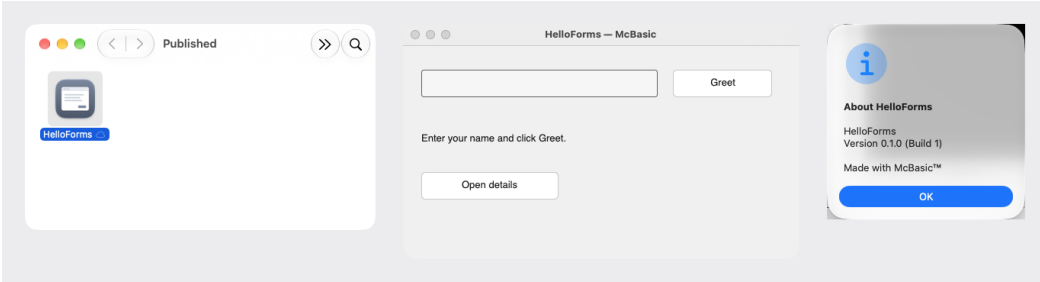


Figure 17.1: Published app result

Check your work

- The app runs without the IDE when the release process requires that.
- User data is stored in a writable user location.

- Resources and packages are present and match the project.
- Signing and notarization status is recorded honestly.
- The clean-Mac test has been performed, not assumed.

Tips and tricks

- **One PNG for both systems.** A square 1024 × 1024 PNG chosen with **Icon:** in **Project > Project Settings…** gives the Mac `.app` and the Windows `.exe` their icon. If Finder still shows the old icon after you publish over an app, move the old copy to the Trash first: Finder caches icons by location.
- **Save and Check before you publish.** Then choose **Run** and use the main workflow once.
- **Use Project > Publish Local App for your own Mac.** Use **Project > Show Published App** to reveal the delivered copy.
- **Publish for Linux from the same project too.** Choose **Linux · x64** or **Linux · ARM64** in the same pop-up; Chapter 21 lists what works on Linux and what waits for a later version.
- **Publish for Windows from the same project.** Choose **Windows · x64** in the **Platform:** pop-up of **Project > Publish Local App**; Chapter 20 lists what to check first.
- **Keep working while the splash is up.** In **Run** and **Debug** the splash shows too, but your form takes clicks and typing at once; if a breakpoint stops the app in its first seconds, the splash hides while the IDE is in front.
- **Test the published app separately.** A project that works in the IDE may still miss a resource or a path in the published copy.
- **Zip the .mcbproj to share a project.** It carries no build output. If chats are saved in the project, clear **Save chats in this project** first when you want to keep them private.

Chapter 18 — Capstone: build WeatherDesk

WeatherDesk is the project that ties the language chapters together. It is a small desktop weather tool with two real boundaries:

1. an external API supplies location and forecast data;
2. a local SQLite database stores the places and weather snapshots the user

chooses to keep.

The project uses Open-Meteo because a reader can try its public endpoints without creating an account or copying an API key. The Free APIs directory is useful for discovering alternatives, but always read the selected provider's own documentation and terms before shipping an application.

Use the checked-in `WeatherDesk` project as the guide. The complete source is in `samples/WeatherDesk.mcbproj`.

The user story

The user wants to:

1. type a city name;
2. use the geocoding API to find matching places;
3. use the selected coordinates to request current weather;
4. see a clear result or a useful failure message;
5. add the place and the fetched values to SQLite;
6. read saved places after closing and reopening the app;
7. update a label or tracking choice;
8. delete a selected place and its forecast snapshots;
9. export the saved locations to CSV.

This is deliberately more than “call an API and print the answer”. The API is the remote input; SQLite gives the app memory, history, and a reason to exist after the network call finishes.

The two APIs

WeatherDesk uses these Open-Meteo endpoints:

```
https://geocoding-api.open-meteo.com/v1/search
https://api.open-meteo.com/v1/forecast
```

The first accepts a city name and returns coordinates. The second accepts latitude and longitude and returns weather JSON. Neither endpoint needs an API key for the free, non-commercial service. The provider publishes limits and terms, including a daily request limit and attribution requirements. A book sample is educational; a commercial application should be checked against the provider's current terms and may need a paid plan.

The database schema

The migration in `samples/WeatherDesk.mcbproj/Database/migrations/0001_initial.sql` creates two tables:

Table	Why it exists
`saved_locations`	The places the user chose to keep, including a label, coordinates, and the `tracked` flag.
`forecast_snapshots`	The current temperature, wind, and weather code fetched for a saved place.

The database lives in `AppPaths.DataFile("weatherdesk.db")`, not in the project. The API never receives this database. On startup the app opens SQLite, applies the migration, queries `saved_locations`, and binds the rows to the DataGrid. That makes the database visible in the first minute of use.

Build order

Build the project in this order. Keep the same `WeatherDesk` project throughout:

1. design the city search, result, status, and saved-location grid;
2. add `Option Explicit`, form variables, and the Clear action;
3. open SQLite and apply the migration;
4. query `saved_locations` and show the empty state;
5. call the geocoding endpoint with the city text;
6. parse the response and keep the first location's name, country, latitude, longitude, and timezone;

7. call the forecast endpoint with those coordinates;
8. show the current values and handle missing JSON fields;
9. add the chosen location and a forecast snapshot with parameters;
10. select a saved row and update its label or tracking flag;
11. delete the selected row and its snapshots after a second confirmation click;
12. reopen the app to prove that the saved row comes from SQLite;
13. export saved locations to CSV.

Run Check after each step. Do not wait until both network and database code have been added before testing.

The API request in BASIC

The geocoding request is small and easy to inspect:

```
Dim url As String
Dim envelopeText As String
Dim bodyText As String
Dim envelope As JsonValue
Dim document As JsonValue

url = "https://geocoding-api.open-meteo.com/v1/search?name="
url = url & Replace(Trim$(txtCity.Text), " ", "+")
url = url & "&count=5&language=en&format=json"
envelopeText = Await Http.RequestAsync( _
    "GET", url, _
    "{""User-Agent"":""McBasic WeatherDesk""}", _
    "", 10000)
envelope = Json.Parse(envelopeText)
bodyText = CStr(Json.Value(Json.Get(envelope, "body")))
document = Json.Parse(bodyText)
```

The response is an HTTP envelope whose `body` contains the provider's JSON text, so the sample parses twice. It then checks that `results` exists and reads the first result only after checking that the array is not empty.

The forecast request uses the coordinates returned by the first call:

```
url = "https://api.open-meteo.com/v1/forecast?latitude="
url = url & CStr(mSelectedLatitude)
url = url & "&longitude=" & CStr(mSelectedLongitude)
url = url & "&current=temperature_2m,wind_speed_10m,weather_code"
url = url & "&timezone=auto"
```

Keep the two calls separate while learning. It makes it obvious that the second request depends on data from the first response.

The database write in BASIC

When the user presses **Add location**, the sample writes the selected place and then its forecast snapshot. It does not save every search result automatically. That

distinction teaches an important application rule: remote data becomes user data only after the user chooses to keep it.

```
Set command = mDb.CreateCommand( _
    "INSERT INTO saved_locations(" & _
    "label, name, country, latitude, longitude, timezone, tracked) " & _
    "VALUES(@label, @name, @country, @latitude, @longitude, @timezone,
@tracked)")
command.AddParameter "@label", Trim$(txtLabel.Text)
command.AddParameter "@name", mSelectedName
command.AddParameter "@country", mSelectedCountry
command.AddParameter "@latitude", mSelectedLatitude
command.AddParameter "@longitude", mSelectedLongitude
command.AddParameter "@timezone", mSelectedTimezone
command.AddParameter "@tracked", 1
Await command.ExecuteAsync()
```

The sample uses parameters instead of joining user text into SQL. After the write, it queries the table again and refreshes the DataGrid. Closing and reopening the app is part of the tutorial, not an optional extra test.

The complete CRUD cycle

The same project then turns the saved-location grid into a small, real data editor. The stable database ID is kept with the selected row; the visible row number is only a position in the current grid.

Create

Find city creates a candidate in memory. **Add location** creates the SQLite row and adds its first record to `forecast_snapshots`. The label and tracking choice are stored with the location. A repeated add is rejected by the `UNIQUE(name, country)` rule instead of silently replacing the existing row.

Read

At startup and after every change, `RefreshLocations` reads the saved rows:

```
Set command = mDb.CreateCommand( _
    "SELECT id, label, name, country, latitude, longitude, " & _
    "timezone, tracked, saved_at, updated_at " & _
    "FROM saved_locations ORDER BY label, name, country")
Set rows = Await command.QueryAsync()
Set mRows = rows.Bind()
Set grdLocations.DataSource = mRows
```

The grid is not the database. It is a view of the latest query result. That is why **Refresh list** runs the query again and why the selected record is tracked by `id`.

Update

Select a row, edit its label or tracking choice, and press **Update**. The coordinates and provider name stay unchanged because the update names only the fields the user is allowed to edit:

```
Set command = mDb.CreateCommand( _
    "UPDATE saved_locations SET label=@label, tracked=@tracked, " & _
    "updated_at=CURRENT_TIMESTAMP WHERE id=@id")
command.AddParameter "@label", Trim$(txtLabel.Text)
command.AddParameter "@tracked", 1
command.AddParameter "@id", mSelectedId
Await command.ExecuteAsync()
```

If the check box is clear, pass `0` instead of `1`. Refresh the grid after the operation so the new `updated_at` value and the new label are visible.

Delete

Delete the dependent snapshots first, then the location. The sample requires a second click so a single accidental click only arms the operation:

```
Set command = mDb.CreateCommand( _
    "DELETE FROM forecast_snapshots WHERE location_id=@id")
command.AddParameter "@id", mSelectedId
Await command.ExecuteAsync()

Set command = mDb.CreateCommand( _
    "DELETE FROM saved_locations WHERE id=@id")
command.AddParameter "@id", mSelectedId
Await command.ExecuteAsync()
```

After deletion, clear the editor and read the grid again. The user should see the row disappear, and a restart should not bring it back.

Create a simple WeatherBadge component

The capstone also teaches how to extract a small reusable visual component. The goal is not to hide the application logic. WeatherDesk continues to own the Open-Meteo calls, JSON parsing, and SQLite. WeatherBadge only displays values and asks the host application to refresh them.

Step 1: create the component project

Create a new **UserControlLibrary** project and name it `WeatherBadge`. Add a UserControl named `WeatherBadge`. Keep it in its own project so it can be checked and packaged independently from WeatherDesk.

Add two labels and a button to the control:

- `lblPlace` — the selected location;
- `lblTemperature` — the temperature in degrees Celsius;
- `lblWind` — the wind speed;
- `cmdRefresh` — the button that asks the host to fetch fresh data.



Figure 18.2: WeatherBadge in the visual designer

Step 2: expose a small public contract

Add public properties and one event to `WeatherBadge.bas`:

```
Option Explicit

Public Event RefreshRequested()

Private mLocationName As String
Private mTemperatureC As Double
Private mWindSpeed As Double
Private mWeatherCode As Long

Public Property Get LocationName() As String
    LocationName = mLocationName
End Property

Public Property Let LocationName(ByVal value As String)
    mLocationName = value
    UpdateDisplay
End Property

Public Property Let TemperatureC(ByVal value As Double)
    mTemperatureC = value
    UpdateDisplay
End Property

Public Property Let WindSpeed(ByVal value As Double)
    mWindSpeed = value
    UpdateDisplay
End Property

Public Property Let WeatherCode(ByVal value As Long)
    mWeatherCode = value
    UpdateDisplay
End Property

Private Sub cmdRefresh_Click()
    RaiseEvent RefreshRequested
End Sub
```

The `UpdateDisplay` procedure assigns the public values to the labels. Keep that procedure inside the component. The host should not reach into `lblTemperature` or `lblWind` directly.

Step 3: check and package the component

Run **Check** on the `WeatherBadge` project. Build the component package and install it in the McBasic™ environment. The checked-in version is available at `samples/WeatherBadge.mcbproj`.

In a copy of WeatherDesk, choose **Project > Plugin References** and add the exact installed `WeatherBadge` package. In the current Mac workflow this action is named **Add Component Package**. Check the consuming project again before placing the control on the form.

Step 4: place the control and pass JSON results to it

Add `WeatherBadge` to the WeatherDesk form. The existing `LoadCurrentWeather` procedure already parses the forecast JSON into `mTemperature`, `mWindSpeed`, and `mWeatherCode`. Add a small presentation procedure:

```
Private Sub ShowWeatherBadge()
    weatherBadge1.LocationName = mSelectedName
    weatherBadge1.TemperatureC = mTemperature
    weatherBadge1.WindSpeed = mWindSpeed
    weatherBadge1.WeatherCode = mWeatherCode
End Sub
```

Call `ShowWeatherBadge` after a successful forecast request. Then handle the component event. The event calls the host's existing API procedure, so the component remains reusable:

```
Private Async Sub weatherBadge1_RefreshRequested()
    If Len(mSelectedName) = 0 Then
        lblStatus.Caption = "Find a city before refreshing the badge."
        Exit Sub
    End If

    Await LoadCurrentWeather()
    ShowWeatherBadge
    lblStatus.Caption = "WeatherBadge refreshed from the JSON response."
End Sub
```

The complete reusable component is `samples/WeatherBadge.mcbproj`. Use the same pattern for other controls: give the component a small public contract, keep data access in the host application, and use an event when the user asks the host to do work.

Step 5: test the component boundary

Run these tests:

Test	Expected result
Assign a location and three weather values	The badge updates its labels.
Press Refresh with a selected place	The host calls the API, parses JSON, and updates the badge.

Test	Expected result
Press Refresh without a place	The host shows a useful message and makes no API call.
Remove the package from the consuming project	Check identifies the missing component instead of opening a misleading partial app.
Change the package archive	The package identity or hash check rejects the changed input.

Repository checkpoints

The complete build is kept in one folder rather than split into unrelated demos:

Checkpoint	What to add	Proof that it works
1. Form	Search fields, result label, buttons, grid	The form opens and Clear works.
2. SQLite	Migration, open, query, bind	Saved locations survive restart.
3. Geocoding API	City request and JSON parsing	“Singapore” returns a place and coordinates.
4. Forecast API	Coordinate request and current values	Temperature and wind appear with the selected place.
5. CRUD	Add, read, update, delete, and tracking flag	A selected location can be changed and removed safely.
6. Component	WeatherBadge properties and event	A reusable control displays the parsed weather values.
7. Export	Parameterized CSV export	Saved locations can be exported.

The final source is `samples/WeatherDesk.mcbproj/Forms/frmWeather.bas`.

What to test

Test	Expected result
Start with no database	SQLite creates the local database and the grid shows its empty state.

Test	Expected result
Search with an empty query	The app asks for a city and does not call the API.
Search for `Singapore`	A location and current weather appear.
Search for nonsense text	The app reports no matching location.
Turn off the network	The app reports an API error and does not change saved rows.
Return incomplete JSON	The app reports a response problem instead of saving partial data.
Add a location	The location and forecast snapshot are stored in SQLite.
Update a location	The new label and tracking choice survive a refresh and restart.
Delete a location	The location and its snapshots are removed after confirmation.
Close and reopen	The saved location is loaded from SQLite.
Export twice	The app explains what happens when the destination already exists.
Repeat the same add	The unique location rule prevents an accidental duplicate.

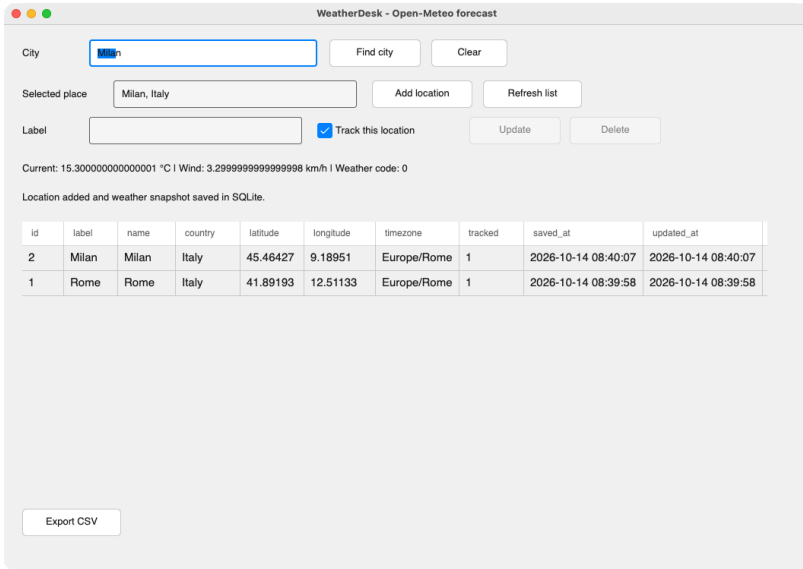


Figure 18.1: WeatherDesk capstone

Final review

Before calling the project complete, ask a person who did not build it to do the main tasks. Watch whether they understand the difference between **Find city** (remote data), **Add location** (Create), and **Update/Delete** (local database operations). That distinction is the heart of this capstone.

Tips and tricks

- **Build the sample in the order the chapter gives.** Each step is checked before the next one starts, so a problem is easy to place.
- **Show the result of each database operation.** Seeing the new row in the grid proves the write worked.
- **Keep the API call and the database write in separate procedures.** Then you can test each one alone.
- **Test the unhappy paths too.** Try a city that does not exist and a missing network.

Chapter 19 — Import a Visual Basic 6 project

McBasic™ can read a Visual Basic 5 or 6 project and write a new McBasic™ project from it. The new project passes Check, keeps the windows and the structure of the original code, and comes with a migration report that says, file by file and line by line, what was converted, what was approximated and what is left for you. This chapter shows the whole path: preparing the files, the **File ▸ Import Visual Basic 6 Project…** command, the report window, and the hand-over to the AI assistant for the rest.

What the importer does

The importer reads a `.vbp` project (or a `.vbg` project group) and the files it lists: forms, modules and classes, with their `.frx` resource files. From them it writes a new `.mcbproj` package.

- **Your Visual Basic files are only read.** The importer never changes the folder you chose, and the new project is always a separate package.
- **The new project passes Check.** Code that McBasic™ does not accept is not dropped. The importer comments it out, keeps the original text beside it, and adds an entry to the report. Every such block is marked as manual work, so a weak import is visible, never hidden.
- **Nothing is sent anywhere.** The import runs on your Mac and does not use an AI provider. The AI assistant helps only when you ask it to, after the import (see "Finish with AI").
- **Controls keep their stacking order.** A control that was in front in Visual Basic 6 is in front in McBasic™ too. Labels, images, shapes and lines stay behind buttons, text boxes and picture boxes, as they did in Visual Basic 6, so text drawn over a background picture stays visible.
- **The new project starts with the Classic look.** The importer sets **Appearance** to **Classic** in Project Settings, so the controls are drawn like a Windows app of the Visual Basic 6 era: grey faces, 3D bevels and sunken text boxes. To use the macOS look, choose **Project > Project Settings…**, set **Appearance:** to **macOS** and press **OK**. Only the look changes. A project that you imported before Build 868 is macOS until you choose **Classic** there, or import the original again to get Classic from the start.

- **System colors keep their names.** A Windows system color in a form (the grey of a button face, the blue of a selection) is imported as the name, for example `ButtonFace`, and no longer as a fixed Windows 10 color. The report has no entry for it, so the result is **Converted**. In the code, a system color number such as `&H8000000F&` assigned to a color property becomes the constant `vbButtonFace`. The colors then follow the project's **Appearance**.
- **A tab control keeps its background.** An `SSTab` with no `BackColor` gets the Visual Basic 6 default button grey (#F0F0F0), so it covers what lies beneath it, as it did in Visual Basic 6.
- **An SSTab keeps its rows of tabs.** Visual Basic 6 shows `Tabs` divided by `TabsPerRow` rows of tabs (three per row by default) and places the controls below them. The importer copies `TabsPerRow` to the TabControl, so the Classic look draws the same rows, and takes the height of all the rows off the controls' `Top`.
- **A DTPicker keeps its format.** A date picker becomes `Mode` Date with `Format` Short or Long; a time picker (format dtpTime) becomes `Mode` Time with the spin buttons of the Classic look, on a 12-hour clock when your Mac uses one; a custom format becomes `Format` Custom with its `CustomFormat`. The date picker's `UpDown` setting is not imported (the report says so).

Every report entry has one of three statuses:

Status	Meaning
Converted	The result behaves like the original.
Approximated	The result compiles and runs, but differs from the original. The entry says how.
Manual	The code or control was stubbed, commented out or replaced by a placeholder. You or the assistant finish it.

Windows-only features are reported, never converted:

- Calls to Windows system libraries (`Declare` of DLL functions, `AddressOf`, window handles and subclassing) become stubs that raise error 5, "not available on macOS". The original line stays beside the stub as a comment.
- ActiveX and COM objects (`CreateObject`, `GetObject`, ADO and DAO data access) are flagged. Use SQLite instead (Chapter 12).
- Controls with no McBasic™ equivalent become a placeholder panel with the original name and position, and an entry in the report.
- OLE, DDE, `SendKeys`, Windows registry calls and Windows Help are flagged or dropped, each with an entry.

- Projects that are not applications (ActiveX DLL and ActiveX Control projects), data report designers and property pages are refused or skipped, and the report says which.

McBasic™ is a trademark of Aniello Castellano. Visual Basic is a trademark of Microsoft Corporation. McBasic™ is an independent product and is not affiliated with, sponsored by or endorsed by Microsoft.

What the importer converts in code

Most Visual Basic 6 code needs only a different spelling in McBasic™, and the importer rewrites it for you. Each kind of rewrite gets one report entry per file, `MCB5643`, that ends with what to do: "nothing to do, keep them as they are", "check …", or "make the change by hand: …".

- **Optional parameters.** A typed `Optional` parameter without a default gets the default Visual Basic 6 used (0, `""`, `False` or `Empty`), written out.
- **Conditional compilation.** `#If`, `#ElseIf`, `#Else`, `#End If` and `#Const` are evaluated as Visual Basic 6 does on Windows. The branches not taken stay in the code as comments (`MCB5644`).
- **Type characters.** `Dim n%` becomes `Dim n As Integer`, and the suffixes `% & ! # @ $` are dropped from names and from functions such as `Left$`.
- **Enums.** A `Public Enum` of a class or a form moves to a standard module named `Vb6Enums`, and references such as `clsShape.skCircle` lose the class name.
- **Events.** Handlers get the McBasic™ signature and event names (`DblClick` becomes `DoubleClick`). When the signatures differ too much, the original code moves to a Sub named `<handler>_Vb6`, and an adapter with the McBasic™ signature calls it. Change the code in the `_Vb6` Sub, not in the adapter. `KeyPress` for Enter, Backspace and Esc is routed from `KeyDown`, because McBasic™ sends those keys only to `KeyDown`.
- **Default properties.** A control used as a value gets its default property written out: `Text1` becomes `Text1.Text`, `Label1` becomes `Label1.Caption`.
- **Lists.** `AddItem`, `ListCount`, `List(i)` and `RemoveItem` of a ListBox or ComboBox become `Items` members, with the index moved from 0-based to 1-based.
- **Modal dialogs.** `frmX.Show vbModal` becomes `Await frmX.ShowDialogAsync()` inside an `Async Sub`. Where that is not possible, a manual marker says where to move the call.
- `End`**,** `Screen` **and class events.** McBasic™ now has the `End` statement, the `Screen` object (for example `Screen.MousePointer`) and the class events `Class_Initialize` and `Class_Terminate` (Chapter 9 and Appendix — McBasic™ language quick reference), so code that uses them is kept as it is.

`Class_Terminate` runs when McBasic™ reclaims the object, not at the moment the last reference goes; its report entry says what to check.

- **User-defined types.** `Type … End Type` stays as written: McBasic™ now has user-defined types, with the copy-on-assignment behaviour of Visual Basic 6 (Appendix — McBasic™ language quick reference). `With` blocks on a Type variable or array element stay as they are.
- **Variables without a declaration.** In files without `Option Explicit`, each variable the code uses without declaring it gets `Dim … As Variant` at the top of its procedure (or the type of its type character or `DefInt`-style range). Check the report entry for misspelt names: a typing error in Visual Basic 6 also became a new variable there. In a form, a bare form property becomes `Me.Caption`, `Me.Left` and so on. A bare `Left`, `Top`, `Width` or `Height` that the code reads, such as `If Width < 3000 Then`, is reported as `MCB5640`, because the value is now in points, not twips. A Visual Basic 6 form property or method that McBasic™ forms do not have (`ScaleWidth`, `KeyPreview`, `AutoRedraw`, `CurrentX`, `hWnd`, …) never becomes a variable: its line keeps its quarantine, with a marker that says why.
- **ByRef arguments.** McBasic™ passes only a variable to a `ByRef` parameter. A parameter the procedure never changes becomes `ByVal`. Otherwise an expression or a property goes through a variable named `importTemp1` (`importTemp2`, …) on the line before the call, as Visual Basic 6 passed it: a copy that the procedure may change without changing the original. In a procedure with `On Error Resume Next` no temporary is used: if the line that fills it failed, it would be skipped and the call would still run with an old value, where Visual Basic 6 skipped the whole statement.
- **Text used as a number.** Text compared with or added to a number, or assigned to a numeric variable, gets `CDbl`, `CLng` or the conversion of the target type. McBasic™ reads numbers without the Windows locale, so check text that holds a decimal comma.
- **Enums used as numbers.** An Enum whose values the code mixes with numbers becomes Long constants (`Public Const lvLow As Long = 1`), as Visual Basic 6 treated it.
- **Files.** `Open`, `Print #`, `Line Input #`, `Input #`, `Write #`, `EOF`, `FreeFile`, `Close`, `Kill`, `FileLen` and `Dir` become calls into a module named `Vb6FileIO`, which the importer adds to the project. Files are written as UTF-8, and `App.Path & "\name"` becomes a file in the app's data folder, `Path.Combine(App.DataPath, "name")`. An `Open` of a relative file name, such as `Open "scores.txt" For Output As #1`, is reported: Visual Basic 6 looked for the file in the current folder, usually the program's folder, but the current folder of a Mac app is the root of the disk. Build the path with `Path.Combine(App.DataPath, "scores.txt")`.

- **App, Clipboard and a few Windows functions.** `App.Title`, `App.EXEName`, `App.PrevInstance` and the version members (`Major`, `Minor`, `Revision`, `ProductName`, `CompanyName`, `FileDescription`) exist in McBasic™, and `App.LegalCopyright` returns the copyright. The importer reads the `.vbp` keys `VersionLegalCopyright` and `VersionCompanyName` into the project's **Copyright** and **Company** (see **Project > Project Settings…**). It drops a value that is empty or only a placeholder (`none`, `company`, `n/a` or `-`, in any letter case), so an imported project does not show a meaningless credit; a long value is shortened to 200 characters. `Clipboard.SetText` and `Clipboard.GetText` lose the text format argument. `ShellExecute` with the verb `"open"` and a written `http`, `https` or `mailto` link, `MessageBox`, `GetTickCount` and `SetWindowPos` to keep a window on top are rewritten; `InitCommonControls` and `App.HelpFile` are removed and kept as a comment. Any other `ShellExecute` (a document, a program, a link in a variable, another verb) gets a marker. `GetTickCount` counts milliseconds since midnight, in steps that grow to about 8 ms late in the day.
- **The app icon.** When the `.vbp` names an `IconForm` and that form has an `Icon` in its `.frx`, the importer stores it as the resource `Resources/AppIcon.ico` and makes it the project's app icon (**Icon:** in **Project > Project Settings…**; the `icon` key of `project.mcproj`). Section "2. Project" of `Import/Report.md` lists it as "App icon". The Windows `.exe` uses it as it is. Visual Basic 6 icons are ICO files, which the Mac cannot use, so the Mac app keeps the McBasic™ icon and publishing for the Mac shows the warning MCB2407 until you choose a square PNG. A form without an `Icon`, or an icon that is a cursor or not a valid ICO file, leaves the McBasic™ icon.
- **Registry settings.** `GetSetting` and `SaveSetting` become `Settings.Get` and `Settings.Set`. The Windows registry ignored the case of section and key names, and McBasic™ settings keys do not, so the importer writes every key in lower case (`"setup.dblocation"`); a computed name goes through `LCase`.
- **Numeric functions on text.** `Int`, `Fix`, `Sgn`, `Round` and the other numeric functions get `CDbl` around a text argument, such as `Int(CDbl(txtScore.Text))`, because Visual Basic 6 converted the text silently.
- **Sounds.** A Windows Media Player control that is invisible or only plays sounds becomes an `AudioPlayer`. A sound file that the form or the code names (a written path, or `App.Path & "\roll.wav"`) and that is in the Visual Basic folder is copied into the project's Resources and becomes the player's `Source`. Its play, pause and stop calls, position, autostart and volume become the `AudioPlayer` members.
- **Timers, pictures and frames.** A Timer without an `Interval` imports with `Interval` 0, which McBasic™ also treats as a timer that never fires. An Image's `BorderStyle` becomes `None` or `Solid`, and a PictureBox that holds other controls becomes a Panel that keeps its picture as `Panel.Picture`.

- `Form_Terminate`**.** It becomes `Form_Closed`, which runs after the form is unloaded. The entry is **Approximated**: check code that relied on the exact moment Visual Basic 6 ran it.
- **Windows API declarations no longer used.** When every call of a `Declare` was rewritten into a McBasic™ service (for example `ShellExecute` or `GetTickCount`), its stub is removed and the `Declare` stays only as a comment; the entry is **Converted**.

Before you import

1. **Copy the whole Visual Basic folder to your Mac.** Include every form, module and class the `.vbp` lists, and include the `.frx` files that sit next to the forms: they hold the pictures and list items. A project that references a file in a sibling folder needs that folder too, so copy the folder one level above the project when the `.vbp` lists paths that start with `..\`.
2. **Save VB3 and VB4 forms as text.** The importer reads text forms from Visual Basic 5 and 6. A binary form from an older version must be opened in Visual Basic 6 and saved again as a text form. The importer refuses a binary form and names the file.
3. **Use an application project.** Projects of type ActiveX DLL and ActiveX Control are refused. Import an Exe project instead.
4. **Keep the original.** The importer never writes in the folder you chose, but keep your own copy for comparison.

Import step by step

1. Choose **File ▸ Import Visual Basic 6 Project…**. If a build, a run, a debug session or an AI request is still running, the status line reads "Finish or stop the running operation first." and nothing opens.
2. In the open panel, titled **Import Visual Basic 6 Project**, choose the folder that holds the `.vbp` file (or the `.vbg` group file). The panel's message reads: "Choose the folder that holds the Visual Basic 6 project (its .vbp or .vbg file). McBasic only reads the project and the files it lists." Press **Choose**. You choose the folder, not the file. The importer never reads above the folder that holds the `.vbp`: a file the project lists from a surrounding folder is reported as missing (`MCB5627`). Copy such shared files next to the `.vbp` before you import.
3. A sheet reads **Reading “<name>”…** while McBasic™ examines the project. **Cancel** stops it.
4. If the folder or the group holds more than one project, an alert titled **Choose a Project to Import** says "“<name>” holds more than one project." Pick one project file name in the pop-up menu and press **Continue**, or press **Cancel**.
5. The save panel is titled **Save Imported McBasic Project**, and its message reads "Choose where to save the new project. The Visual Basic 6 files are only read,

never changed." The panel proposes your McBasic Projects folder and the name of the Visual Basic project, with the `.mcbproj` ending. When that name is taken, the proposal ends with `(imported 2)`, then `(imported 3)`, and so on. The importer never replaces a project. If you choose an existing name, the panel says "A project with this name already exists. The importer never replaces a project; choose another name." If you choose a place inside the Visual Basic folder you picked, the panel says "The new project cannot be saved inside the Visual Basic 6 folder you chose, because that folder is only read. Choose a folder outside it."

6. Below the name, the panel shows an accessory:

- **Code page:** a pop-up menu. It starts with **Automatic (<name>)**, the code page McBasic™ detected, and then lists the supported code pages as `<number> — <name>`. Leave it on Automatic unless the captions come out wrong (see Tips and tricks).
- When files that the project lists are missing, a line says "<n> file(s) listed in the project are missing. The migration report lists them first."
- The trademark disclaimer.

7. Press **Import**. A sheet reads **Importing “<name>”…** and shows the latest progress line:

- Reading the Visual Basic 6 project…
- Converting forms and code…
- Writing the new project…
- Checking the imported code (round <n>)…
- Writing the migration report…
- Finishing…

The "Checking" line can repeat: each round comments out whatever Check still rejects, until Check passes.

8. **Cancel** on this sheet stops the import. McBasic™ then says "Import cancelled. Nothing was written." and leaves no folder behind. If the import had already finished when Cancel took effect, you read "The import finished before Cancel took effect, so the project was created." and the project exists.

McBasic™ also stops an import that takes too long, and leaves no project behind. The alert **The Project Was Not Imported** then shows one of these messages:

- "The importer showed no progress for 5 minutes, so McBasic stopped it. Import the project again." (no new progress line for 5 minutes)
- "The import ran for 60 minutes, so McBasic stopped it. Import the project again." (the whole import took longer than 60 minutes)

A timeout shows its own alert and is never reported as "cancelled".

Cancel, a timeout and a quit all stop the importer the same way: McBasic™ asks it to end, and forces it to end after 5 seconds if it does not. If you quit McBasic™ while an import runs, it does not ask. It stops the import, waits for the cleanup to finish, shows no alert, and then quits.

When McBasic™ cannot remove the partial output of a stopped import, the alert adds "Move the incomplete “<name>” to the Trash now, or keep it and delete it yourself." and offers **Move to Trash** and **Keep**. McBasic™ never trashes that folder without your answer.

If you choose a project that an interrupted import left incomplete, McBasic™ does not open it. The alert **This Project Is an Incomplete Import** offers to move it to the Trash, and after you agree the status line reads "Moved “<package>” to the Trash."

9. When the import succeeds, the status line reads "Imported “<name>”. The migration report lists what to finish." The new project opens, and the migration report opens in its own window (next sections). If the project that was open has unsaved changes and you cancel the save prompt, or McBasic™ is still busy, the new project stays closed and the status line adds " Open it with File ▸ Open."

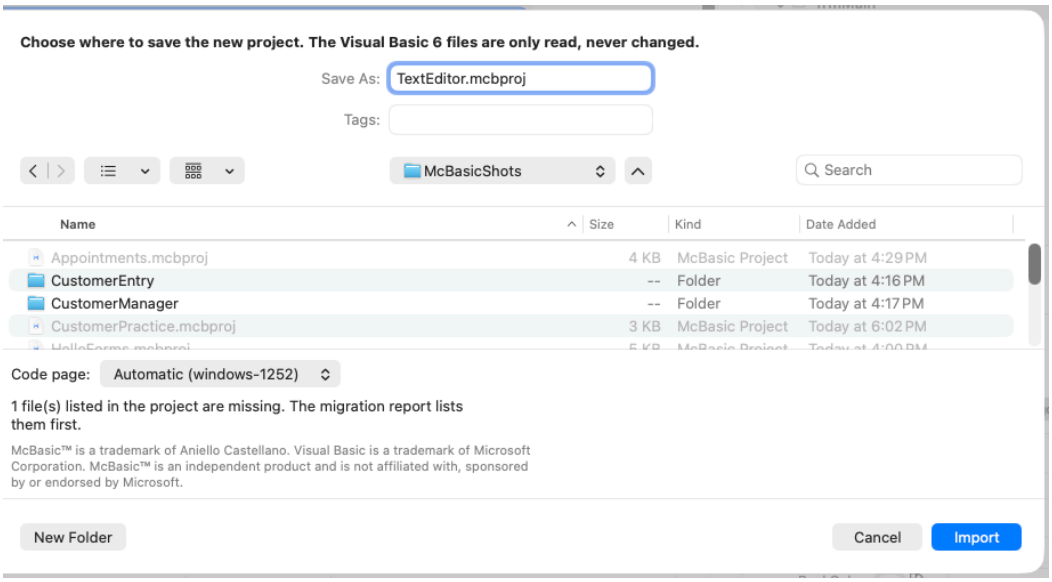


Figure 19.1: The save panel for an imported project

When the import stops

If the import cannot continue, nothing is written: no project, no half-finished folder.

- **Refusal.** An alert titled **This Project Cannot Be Imported** lists each reason as `<code>: <message>` on its own line, then says "Nothing was written." Press **OK**. A refusal means the input is wrong, and each message names the fix. When the importer refuses without giving a reason, the alert says "McBasic could not find a Visual Basic 6 project it can import here. Choose the folder that holds the project's .vbp file and try again."

- **Failure.** An alert titled **The Project Was Not Imported** shows `<code>: <message>`, a blank line, and "Nothing was written."
- **Crash.** If the importer ends without an answer, the same alert says "The importer stopped unexpectedly. Import the project again; if it happens again, report the problem and include the details below." A **Details** box holds the importer's raw output, which you can copy into a problem report.

The three messages you are most likely to meet:

Message	What to do
`MCB5600`: "… does not exist; choose a .vbp file or the folder that holds it", or "the folder … holds no .vbp file; choose the folder that holds the project's .vbp"	You chose a folder without a project, or a path that moved. Choose the folder that holds the `.vbp` file.
`MCB5603`: "… already exists; the importer never overwrites, so choose another name or move the existing project" (also: the output is inside an application, another project package or the Visual Basic folder)	Import again and choose another name, or a folder outside the Visual Basic project.
`MCB5605`: "code page … is not supported; use one of …"	Choose a code page from the list in the **Code page:** menu, or leave it on **Automatic**.

Other `MCB56xx` codes follow the same pattern: the message names what is wrong and how to fix it. Chapter 23 lists the family.

Read the migration report

After a successful import the report opens in a window titled **Migration Report — <project name>**. You can open it again at any time with **Project ▸ Show Migration Report**. That item is available only when the open project has an `Import/report.json` or `Import/Report.md` file. If the report was removed after the menu was shown, the status line reads "This project has no migration report."

The window has these parts:

- **The summary line** at the top counts the entries by status and says how many lines were converted, rewritten or commented out.
- **The filter** is a segmented control: **All**, **Manual**, **Approximated** and **Converted**. Start with **Manual**.
- **The list** has the columns (tick), **Status**, **Code**, **Location** and **What to do**. Entries are grouped. **Missing files** comes first, then **Project**, then one group for each file.
- **The detail pane** under the list shows the full text of the selected entry, with its location, status and code, so a long **What to do** never has to be read in a narrow

column. Drag the divider to make it taller. With no selection it reads "Select an entry to read its full text here."

- **Double-click an entry** to open it. An entry in a code file opens the code editor at that line, and the status line reads "Migration report: <file>, line <n>." An entry in a form opens the form in the designer, and the status line reads "Migration report: opened <file>." If the file is no longer a document of the project, nothing opens and the status line reads "Migration report: <file> is not a document of this project."
- **Keyboard.** Select an entry and press Space to tick or untick it. An entry for a missing file that has no finding cannot be ticked.
- **Open Report.md** opens the full text report, which also lists the source files that were read, the controls that were mapped, the renamed identifiers, the Windows-only calls grouped by library, and an ordered list of next steps.
- **Finish with AI…** hands the ticked entries to the assistant (next section).

Banners can appear above the list. "This report was written by a newer importer (schema <n>); some details may not show. Open Report.md for everything." means the report came from a newer version of McBasic™: the entries that can be read are shown. "This report was written by a newer or unknown importer; some details may not show. Open Report.md for everything." is the same note when the schema number cannot be read. "<n> entries could not be read and are not shown. Open Report.md for everything." (or "1 entry could not be read…") means some entries were skipped. When two notes apply, they are shown together. "The report's data file could not be read, so entries are not clickable." means the window shows Report.md as plain text instead.

The `Import/` folder (`Report.md` and `report.json`) is not a project input. The AI assistant never reads it by itself, project search ignores it, and you may delete it. Deleting it removes the report window's data, not the project.

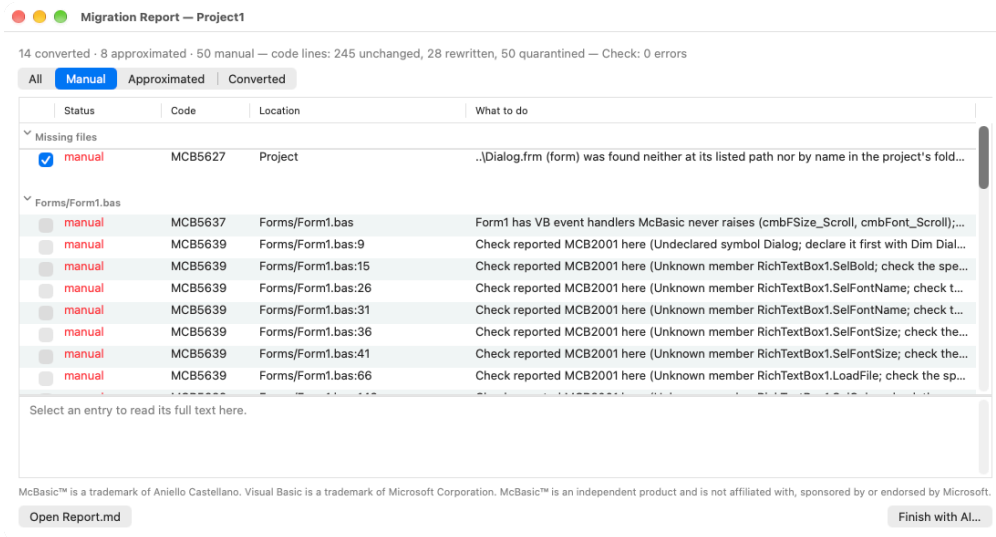


Figure 19.2: The migration report window

Finish with AI

The assistant can work through the remaining entries in the same staged way as any other plan (Chapter 16).

1. In the report window, tick the entries you want to finish. Group them by area: all the data-access entries together, for example.
2. Press **Finish with AI…**.
3. The chat opens in Plan mode. The files of the ticked entries appear as scope chips, and the message box holds the ticked entries as a request.
4. Read the message and the chips. **Nothing is sent until you press Send.** When you do, the usual consent, request budgets, staged plan, approval and per-stage Undo apply, exactly as in Chapter 16.

Limits: one request carries at most 40 entries that touch at most 32 files. The window explains each refusal:

- "Tick the report entries to finish with AI first."
- "Tick at most 40 entries for one AI request."
- "The ticked entries touch more than 32 files; tick fewer."
- "The ticked entries are too long for one AI request; tick fewer."
- "Open the imported project first."
- "Wait for the current AI request or press Stop, then add to the chat." (the assistant is still working on another request)

What to finish by hand

Open the code and search for these markers:

- A line starting `' IMPORT MCB56xx (status):` explains what the importer did or could not do, with the code of the report entry and one of the statuses converted, approximated or manual. The message names the fix.
- A line starting `' VB6:` holds the original Visual Basic statement that the importer replaced or commented out. Delete it when you are done.
- A procedure whose body only raises error 5 with "Imported code needs manual conversion" could not be converted line by line. Its signature stays, so the code that calls it still compiles.
- A function that raises error 5 "not available on macOS" is a stub for a Windows system call. Replace the calls with a McBasic™ service, or remove them.
- A placeholder panel stands for a control with no equivalent. Replace it with a McBasic™ control or a plugin.

- **Data access.** ADO and DAO code needs a new design: use SQLite (Chapter 12). The report groups the quarantined ADO, DAO and data-control lines of each file in one entry, `MCB5632`.
- **Winsock, MSComm and Inet.** Their quarantined lines are listed per file as `MCB5645`. Use the Http service or a plugin.
- `Open … For Binary` **and** `For Random`**.** `Get`, `Put` and record files have no equivalent; each statement is marked `MCB5633`. The same marker sits above an `Input #` whose target is not a plain `String` or number variable (a field of a Type, an array element, a `Variant`): declare the target as such a variable, or read the field with `Vb6FileInputField(n)` and convert the text yourself.
- **ByRef arguments the importer could not pass.** A temporary variable needs a line of its own just before the call, filled once. That is not possible in a loop condition or an `ElseIf`, which Visual Basic 6 evaluated again on each test, nor on a single-line `If` or a line with `:`, where the importer has no place for that line. In a procedure with `On Error Resume Next` it is not safe either. In these cases the line keeps its quarantine and a `MCB5642` marker says how to pass a variable by hand.
- **Object variables without a declaration.** `Set x = …` on a variable that was never declared, in a file without `Option Explicit`, is marked `MCB5641`: declare it with its class or control type.
- **Drag and drop between controls.** `DragMode` and `DragIcon` import (`DragIcon` becomes a project resource, like `Picture`), and `.Drag` calls pass through unchanged. `DragDrop` and `DragOver` handlers on controls, control arrays and forms stay as written when the body never reads `X` or `Y`, or when the form uses pixels. Otherwise the body moves to a `<Name>_DragDrop_Vb6` or `<Name>_DragOver_Vb6` Sub and an adapter passes `X` and `Y` in Visual Basic units (twips), like the mouse-event adapters. `Source.Move X, Y` on a twips form is flagged `MCB5640` (divide by 15), as in mouse handlers, and numeric offsets compared with `Source` geometry are converted. `vbBeginDrag`, `vbEndDrag`, `vbEnter`, `vbLeave` and `vbOver` are available. OLE drag (`OLEDragDrop`, `OLEDragOver` and the other OLE events) is still listed as never raised.

The report's **Next steps** (section 8 of `Report.md`) starts with a table of these grouped entries: for each file, the kind (`dataAccess`, or `windowsOnly:Winsock`, `windowsOnly:MSComm`, `windowsOnly:Inet`) and the number of quarantined lines. The same rows are in `report.json` as `manualInventory`. Use it to size the work before you start.

The importer also covers the parts of the project that need no manual work: forms and controls with their positions, menus and shortcuts, and resources from the `.frx` files. Check the entries marked **Approximated** after you run the application, because they run but may look or behave a little differently.

Check your work

- You know which files to copy before you import, and why the `.frx` files matter.
- You can start the import from **File ▸ Import Visual Basic 6 Project…** and choose the folder, the name and the **Code page:**.
- You know that **Cancel** leaves nothing behind.
- You can filter the report with **Manual** and open an entry at its line.
- You know that **Finish with AI…** sends nothing until you press Send.

Tips and tricks

- **Give the imported app a Mac icon.** Open `Resources/AppIcon.ico` in Preview, export its largest size as PNG, enlarge it to 1024 × 1024 if needed, and choose it with **Icon:** in **Project > Project Settings…**. Both the Mac and the Windows app then use it.
- **Import the project folder, not a copy of one file.** The importer needs the files the `.vbp` lists and their `.frx` files next to them.
- **Tick the Missing files first, and import again.** Copy the missing files into the Visual Basic folder, then run the import again with another name. The new report starts clean.
- **Work through the Manual entries by area.** Tick one area at a time, for example all data access, and send one **Finish with AI…** request for each. Each request becomes its own staged plan with its own Undo.
- **Keep the original Visual Basic folder untouched.** Compare the old and the new project side by side whenever an **Approximated** entry makes you doubt a result.
- **Check Code page: when the captions show wrong letters.** Cyrillic, Greek, Thai or Japanese text in the original needs the code page of its language. Import again, and choose it in **Code page:** instead of **Automatic**.
- **Search for `' IMPORT` in the code.** The search finds every spot the importer marked, in the order of the files.
- **Add `Option Explicit` before you import.** Add it to the Visual Basic 6 modules you can still compile in Visual Basic 6: implicit variables then show up as Visual Basic 6 errors that you fix at the source, instead of `Variant` variables in McBasic™.
- **Copy the data files next to the program into the app's data folder.** Files the original kept next to the program (`App.Path`) are read from `App.DataPath` now. Search the code for `App.DataPath` to find the files it opens, and copy them there.
- **Read long entries in the detail pane.** Select an entry and read its whole text under the list instead of widening the **What to do** column.

- **Import again to get the Classic look.** A project that you imported before Build 868 looks like macOS. Import the original folder again with a new name, or choose **Classic** in **Project > Project Settings…**.
- **Colors stay names after the import.** Look for `vbButtonFace` and similar constants in the code. To change a color for good, assign `#RRGGBB` text or another system color name.
- **Delete `Import/` when you are finished.** The report is a working note, not part of the project. The next import creates a new one.

Chapter 20 — Publish an app for Windows

McBasic™ runs on your Mac, but the apps it publishes can also run on Windows. You publish a Windows app from the same project, on the same Mac, with no Windows PC needed to build it. The result is one file, `<Name>.exe`, that you can copy to a Windows PC and start with a double-click.

Every service of the Mac app now works in a Windows app, the `VideoPlayer` included. This chapter lists what works, the few things that behave differently on Windows, and how to try every service on a real Windows PC with the sample **WindowsServicesTest**.

What you get

- **One self-contained file.** `MyApp.exe` carries your program, the McBasic™ runtime and the .NET runtime inside it. The Windows PC needs nothing installed. A small app is about 50 MB.
- **Two kinds of PC.** Choose **Windows · x64** for ordinary PCs. This file also runs on Windows PCs with an ARM processor, through Windows' emulation. Choose **Windows · ARM64** for a file that runs natively on ARM PCs. When you do not know which PC will use the app, publish **Windows · x64**.
- **The same look as on the Mac.** Both looks are drawn by McBasic™ itself, not by the Mac. A project with the **Classic** appearance (the default of every imported Visual Basic 6 project) looks the same on Windows as on the Mac: same colors, bevels and Microsoft Sans Serif font. A project with the **macOS** appearance keeps its Mac-styled controls on Windows. The appearance never switches by itself when the platform changes. You choose it in **Project > Project Settings…**.
- **Your icon on the `.exe`.** The icon chosen with **Icon:** in **Project > Project Settings…** goes into `MyApp.exe`, and File Explorer shows it. From a square PNG McBasic™ makes a Windows icon with the sizes 16, 24, 32, 48, 64, 128 and 256 pixels; an ICO file is used as it is. An ICNS file is a Mac icon only: the `.exe` keeps the McBasic™ icon and the publish shows the warning "MCB2407: the app icon AppIcon.icns is a macOS .icns file, so the Windows app shows the McBasic icon; choose a square PNG in Project > Project Settings to use one icon on both." Without an icon the `.exe` has the McBasic™ icon.
- **No signature.** The file is unsigned, and Windows asks for confirmation the first time you start it (see "First launch on Windows").

Only a **desktop application** project can be published for Windows. A Plugin or a UserControl library cannot.

Publish step by step

1. Save the project and choose **Check**. Fix every error.
2. Choose **Project > Publish Local App**. McBasic™ checks the project and prepares the app, then shows the Save panel.
3. In the panel, open the **Platform:** pop-up. It lists:

• **macOS** (the Mac app, `MyApp.app`, as before);

• **Windows · x64**;

• **Windows · ARM64**.

4. Choose a Windows platform. The proposed name changes to `<Project>.exe`, the panel accepts only `.exe`, and the **Build:** pop-up disappears, because **Compatible** and **Compact** apply to the Mac app only. The panel shows the note *Unsigned: Windows SmartScreen asks for confirmation on first launch.*
5. Choose the folder and the name, then confirm. The folder must be outside the `.mcbproj`. If a file with that name exists, McBasic™ asks before it replaces it.
6. The **Build Log** reports the result, for example `Published MyApp.exe (win-x64, 41 MB)`. **Project > Show Published App** shows the file in the Finder.

McBasic™ remembers the platform you chose for each project on this Mac. The choice is not stored in the project.

A failed publish leaves nothing behind: you never get a half-written `.exe`.

What works on Windows

A Windows app uses the Windows version of each service. Your BASIC code does not change.

Service	On Windows
`MsgBox`	The standard Windows message box. All button sets, the icons (`vbCritical`, `vbQuestion`, `vbExclamation`, `vbInformation`), the default buttons and `vbSystemModal` work, and `MsgBox` returns the same values as on the Mac (`vbOK` 1, `vbCancel` 2, `vbYes` 6, `vbNo` 7).
`InputBox`	A small window with the prompt, a text box that holds the default text selected, and **OK** and **Cancel** buttons. **Cancel** returns an empty string.

Service	On Windows
File and folder dialogs	`Dialogs.OpenFile`, `Dialogs.SaveFile`, `Dialogs.SelectFolder` and `ShowOpen`, `ShowSave`, `ShowFolder` of the CommonDialog use the Windows dialogs. Your `Filter` text such as `Text files|*.txt;*.md` works unchanged. Cancel is reported through `Cancelled`.
Color, font and printing dialogs	`ShowColor`, `ShowFont`, `ShowPrinter` and `ShowPageSetup` of the CommonDialog open the Windows dialogs and fill the same properties as on the Mac. Fields that Windows cannot give keep their default values.
Clipboard	`Clipboard.SetText`, `Clipboard.GetText`, `Clipboard.HasText` and `Clipboard.Clear`.
Web links	`Platform.OpenUrl` opens `http`, `https` and `mailto` links in the default program, as on the Mac.
Menus	The menus of a form appear inside the window, at the top. The window grows by the height of the menu bar, so the area for your controls keeps the size you designed. `Height` (and `MinHeight`, `MaxHeight`) is the area for your controls, under the menu bar, as on the Mac; the window is taller by the menu bar. Command shortcuts become **Ctrl** (a menu item with Shortcut `Ctrl+S` shows `Ctrl+S` and answers to Ctrl+S).
About and Exit	The McBasic™ About box is always there and an `mnuAbout` item does not replace it. It is **Help ▸ About** `<your app>` at the end of the menu bar; if the form has no Help menu, McBasic™ adds one. A form without menus has **About** `<your app>` in the system menu (click the icon at the left of the title bar, or press Alt+Space). A form without menus and without a system menu (`ControlBox = False`, or `BorderStyle = None`) gets a small menu bar with only **Help ▸ About** `<your app>`; the window grows by that bar. **File ▸ Exit** is added only when the form has no File menu of its own.

Service	On Windows
Look and font	The **macOS** appearance uses the font Segoe UI on Windows, at the same size. **Classic** keeps Microsoft Sans Serif.
MDI forms	Ctrl+Tab and Ctrl+Shift+Tab (or Ctrl+F6 and Ctrl+Shift+F6) switch between the child windows, and Ctrl+F4 closes the active child.
`AudioPlayer`	`Load`, `Play`, `Pause`, `Stop`, `Position`, `Volume`, `Loop` and the events. WAV and MP3 files always work. AIFF, M4A and CAF depend on the codecs installed on that PC; when Windows cannot open a file, the player sets `Error`, raises its `Failed` event and the call stops with error 5. Setting `Position` before `Play` works.
`VideoPlayer`	`Source` (a project resource), `AutoPlay`, `Loop`, `Volume`, `Stretch`, `ControlsVisible`, `Position`, `Duration`, `Playing`, `Paused`, `State`, `Error`, `Load`, `Play`, `Pause`, `Stop` and the events `Loaded`, `Completed`, `Failed` and `StateChanged`, as on the Mac. MP4 and M4V files with H.264 video and AAC sound, MOV files with H.264 video, WMV and AVI with common codecs play on every Windows 10 and 11 PC; HEVC (H.265) and VP9 video need the free extensions from the Microsoft Store. See "The VideoPlayer on Windows" below.
`Reports.Print`	The Windows **Print** dialog opens, on the default printer, with the paper and orientation of the report already chosen. **Print** prints the report and returns `True`; **Cancel** prints nothing and returns `False`, as on the Mac. See "Print a report on Windows" below.

The services run on the same thread as your code and wait for the answer, as on the Mac. A dialog belongs to the window that was active when it opened.

What the Check says about Windows

A Windows publish runs the normal Check and then one more pass, which looks for services that a Windows app does not have. In this version there are none, so the pass finds nothing and the Build Log shows no Windows warnings.

If a later version of McBasic™ lacks a service on Windows, the pass adds a **warning** MCB2406 to the Build Log for every use of it, with the member and the

line. A warning does not stop the publish: the `.exe` is still produced, and at run time on Windows that one call stops with error 5 and the message MCB4415, which `On Error` or `Try ... Catch` can handle. The pass runs only when you publish for Windows; a normal Check and a Mac publish never show these warnings.

The VideoPlayer on Windows

A `VideoPlayer` works in a Windows app with the same properties, methods and events as on the Mac. Your code does not change. On Windows the video plays through Media Foundation, the video engine built into Windows, which draws the picture itself, in its own area of the window. A few things differ:

- **Formats.** These play on every Windows 10 and 11 PC:
- MP4 and M4V files with H.264 video and AAC sound;
- MOV files with H.264 video;
- WMV files, and AVI files with common codecs (an AVI file can hold almost any codec; an unusual one plays only when it is installed);
- MP3 or WAV sound tracks inside the video.

Video in HEVC (H.265) or VP9 plays only when the free **HEVC Video Extensions** or **VP9 Video Extensions** from the Microsoft Store are installed on that PC. When Windows cannot open or decode a file, the player sets `Error`, raises its `Failed` event and its state becomes `Error`, as on the Mac; `Error` starts with `Windows could not open or decode the video source:` and says why, for example that no decoder for the format is installed. For a video that must play everywhere, use an MP4 file with H.264 video.

- **Windows N editions.** The N and KN editions of Windows sold in Europe and Korea come without Media Foundation. On such a PC every video fails with a message that names the **Media Feature Pack**; installing it from Windows **Settings** ▸ **System** ▸ **Optional features** (Windows 11) or **Settings** ▸ **Apps** ▸ **Optional features** (Windows 10) makes the video play.
- **Project resources only.** On Windows the `Source` must be a video stored as a project resource. An HTTPS address or a YouTube link plays on the Mac only. On Windows such a `Source` is not loaded: the player's state becomes `Error`, `Error` says why and the `Failed` event is raised once. Set in code, the statement then stops with error 5. Set in the Properties panel, the form still opens and `Failed` is raised just after the form's `Load` event.
- **The controls bar.** With `ControlsVisible = True` the player shows a bar under the video, inside the control, with a **Play**/**Pause** button, a position slider and the elapsed and total time. The bar takes about 32 points of the control's height, so the video area is a little smaller than the control. In a project with the **Classic**

appearance the bar has the Classic grey look. With `ControlsVisible = False` the video uses the whole control.

- **Nothing on top of the video.** No other control can be drawn over the video area. A label or a button that you place on top of a `VideoPlayer` is hidden behind the video on Windows. Place such controls next to the player.
- **Stretch.** `Uniform` shows the whole picture with black bands where it does not fill the control, `Fill` stretches it to the control, and `UniformToFill` fills the control and cuts off the parts that do not fit. The video follows the control when the window is resized.
- **Closing.** When the form closes, the player stops and releases the video file.

Everything else, including forms, controls, the Timer, files, settings and SQLite, is meant to work as on the Mac; test your app on a Windows PC to be sure.

Print a report on Windows

`Reports.Print` works in a Windows app with the same report file as on the Mac. Your code does not change. On Windows it prints through the printing system of Windows:

1. The Windows **Print** dialog opens in front of the active form. It starts on the default printer, with the paper size (A4 or Letter) and the orientation of the report already chosen.
2. In **Select Printer**, choose the printer. **Preferences** opens the printer's own settings, such as the paper, the orientation and the quality.
3. In **Page Range**, choose **All**, or **Pages** and type the pages, for example `1-2, 4`. Pages print in the order you type them.
4. Set **Number of copies** and **Collate**. The printer makes the copies; when a printer cannot make copies, these boxes are not available.
5. Click **Print**. The report goes to the printer and `Reports.Print` returns `True`.

Cancel prints nothing and `Reports.Print` returns `False`. **Apply** keeps the choices while the dialog is open but does not print; when you then close the dialog, `Reports.Print` returns `False`.

How the page looks on paper:

- **Size.** Each page fits into the area the printer can print, as on the Mac. Most printers cannot print at the very edge of the paper, so a page as large as the paper prints slightly smaller, centred on the sheet, and nothing is cut off. A page that is much larger than the paper, such as a Letter report on A4 paper or a landscape report on paper in portrait, is made smaller to fit. A page always keeps its proportions and is never enlarged.

- **Quality.** McBasic™ draws each page as a picture at up to 200 dpi and sends it to the printer one page at a time, in small bands, so even long reports do not overload the printer. Text and lines print clearly; on a high-resolution laser printer they are a little less crisp than text printed from a word processor. A PDF made with **Microsoft Print to PDF** holds pictures of the pages, so its text cannot be searched or copied. For a PDF with real text, use `Reports.ExportPdf`.
- **Microsoft Print to PDF.** This printer, present on every Windows 10 and 11 PC, asks for a file name after **Print**. If you cancel that question, nothing is printed and `Reports.Print` returns `False`.
- **No Print to file.** The dialog has no **Print to file** box. To get a file, choose **Microsoft Print to PDF**, or use `Reports.ExportPdf`.

When the printer or Windows fails, the call stops with error 5 and a message that starts with `Windows could not print the report:` and names the step and the reason, for example `StartPage failed (The printer is out of paper)`. McBasic™ cancels that print job. `Reports.Print` also needs an open form: without one it stops with error 5 and the message `The print dialog needs an open form to own it.` If the dialog ever returns **Print** with no page to print, the call stops with error 5 and the message `The print dialog selected no pages to print.`, so that it is not mistaken for **Cancel**. Use `On Error` or `Try ... Catch` to show the message in a label.

Try every service: the WindowsServicesTest sample

The McBasic™ samples include **WindowsServicesTest** (`samples/WindowsServicesTest.mcbproj`). Its main form, **Windows Services Test**, has one button per service; two buttons open small extra forms. Each button shows its result in the label at the top of the form, so you never need a message box to read an answer.

1. Open the sample, choose **Project > Publish Local App**, pick **Windows · x64** (or **Windows · ARM64**) and copy the `.exe` to a Windows PC.
2. Click each button and compare the label with this list.

Button	What it tries	What you should see
MsgBox OK only, **MsgBox Yes/No/Cancel**, **MsgBox Critical**, **MsgBox Exclamation**, **MsgBox Information**	The five message box styles. The Yes/No/Cancel box has a question icon and **No** as the default button.	The right icon and buttons; the label says, for example, `Yes/No/Cancel returned 7`.

Button	What it tries	What you should see
InputBox	An input box with the default `Ada`.	`InputBox returned [Ada]`, or `[]` after **Cancel**.
ShowOpen, **ShowSave**, **ShowFolder**	The file and folder dialogs; the open dialog filters text files.	The chosen path, or `Open cancelled` (`Save cancelled`, `Folder cancelled`).
ShowColor	The color dialog.	The panel under the buttons takes the chosen color.
ShowFont	The font dialog.	The sample text under the buttons takes the chosen font.
ShowPrinter, **ShowPageSetup**	The print and page setup dialogs.	The printer and paper, or the orientation and page size. A printer name saved on the Mac is not a Windows printer: the dialog opens on the default printer instead.
Clipboard round trip	`SetText`, `GetText`, `HasText`, `Clear`.	`Clipboard has text: True, read back [McBasic clipboard test]; after Clear has text: False.`
Open mcbasic.com	`Platform.OpenUrl`.	The default browser opens `https://www.mcbasic.com`.
Play, **Pause**, **Stop**	An `AudioPlayer` with a two-second tone (a WAV file stored as a project resource).	You hear the tone; the label shows the player state.
Position 0.5 + Play	`Position = 0.5` before `Play`.	The tone starts half a second in, with no error.
Volume 0.5	`Volume = 0.5`.	`Volume is 0.5`; the next **Play** is quieter.
Open second form	Opens a second form without blocking the first. The second form has `MinButton = False` and no menus.	Both windows stay usable. On the second form the minimize button is disabled, and the system menu (Alt+Space, or the icon at the left of the title bar) ends with **About WindowsServicesTest**.
Open form without box	A form with `ControlBox = False` and no menus.	No system menu and no close button; a menu bar with only **Help** ▸ **About WindowsServicesTest**. **Close** closes it.

Button	What it tries	What you should see
Print report	`Reports.Print` of a one-page A4 report.	The Windows **Print** dialog opens on the default printer. Choose **Microsoft Print to PDF**, click **Print** and give a file name: the label says `Report sent to the printer`, and the PDF file shows the page with its heading, the paragraph, the table and the total, centred on the sheet. **Cancel** in the dialog, or in the file name prompt, makes the label say `Report printing cancelled`.
Raise error 5	`Err.Raise 5` on purpose.	The run-time error dialog of the app.
Video Play, **Video Pause**, **Video Stop**	A `VideoPlayer` with a one-second clip (`demo.mp4`, H.264, stored as a project resource), at the right of the form.	The clip plays in the black area, with the controls bar under it; the label shows the video state. **Video Stop** rewinds to the start.
Video Position 0.5	`Position = 0.5`, then `Play`.	The clip plays from the middle; the label shows the position.
Video Volume 0.3	`Volume = 0.3`.	`Video volume is 0.3`. The clip has no sound track, so you hear nothing in any case.
Video Loop on/off	Turns `Loop` on or off.	With Loop on, the clip starts again at the end until you click **Video Pause** or **Video Stop**; with Loop off the label says `Video completed` at the end.
Video Stretch next	Cycles `Stretch` through `Uniform`, `Fill` and `UniformToFill`.	The picture keeps its proportions with black bands, then fills the area stretched, then fills it and is cut at the sides.

The form also has a menu bar: **File** (**Open…** Ctrl+O, **Save…** Ctrl+S, **Exit**) and **Edit** (**Copy** Ctrl+C, **Paste** Ctrl+V), plus the **Help** menu with **About** that McBasic™ adds. Choose each item and press each shortcut; the label shows, for example, `Menu: File > Save`. Press Alt+F and Alt+E: the menus open by their underlined letters. Choose **Help ▸ About WindowsServicesTest**: the McBasic™ About box opens. The notes at the bottom of the form repeat these checks, and the note under the video buttons lists the video checks: the clip shows in the black area with the controls bar below it, **Video Stretch next** changes the picture, and Loop repeats it. The sample uses the **macOS** appearance, so the controls use Segoe UI on Windows.

To try the MDI keys, publish the **MdiForms** sample for Windows too. In its main window, Ctrl+Tab and Ctrl+F6 move to the next child window, Ctrl+Shift+Tab and Ctrl+Shift+F6 to the previous one, and Ctrl+F4 closes the active child.

When the publish stops: MCB2302

If the Windows .NET packs are missing from this McBasic™ installation, the publish stops before it starts, with:

```
MCB2302: Publishing for Windows needs the Windows .NET packs, which are
missing from this McBasic toolchain. Reinstall or update McBasic.
```

Nothing is wrong with your project. Install McBasic™ again, or update to a version that includes the Windows packs. Mac publishing is not affected.

Two related messages come from the same command. MCB2301 says *publishing for Windows requires a DesktopApp project* when you choose a Windows platform for a Plugin or a library. MCB1001 says that a Windows target cannot use a Mac publication profile.

First launch on Windows

Copy `MyApp.exe` to the PC with a USB drive, a shared folder or a download. Because the file is not signed, Windows SmartScreen may show **Windows protected your PC** the first time. To start the app:

1. Click **More info**.
2. Click **Run anyway**.

A copy that you download again may ask once more.

The first start takes a little longer. The app unpacks its files on first use, into a folder under the user's temporary files (`%TEMP%\.net`), and then starts from there. Later starts reuse that folder. You can delete the folder at any time; the app makes it again.

The launch splash on Windows

A Windows app published with McBasic™ Free Edition opens with the same launch splash as a Mac app (see "The Free Edition launch splash" in Chapter 17): the McBasic™ logo on a blue panel, **Made with McBasic™**, the **Free Edition** badge, your app's name, version, build and copyright, a **mcbasic.com** link button and a countdown bar. On Windows the card is always light; on Windows 11 it has rounded corners and the system border and shadow.

The splash stays for five seconds and then closes, without a fade. As on the Mac, it steps aside while one of your app's dialogs is open and comes back for the rest of its five seconds when the dialog closes, and a click on **mcbasic.com** opens the

website in the default browser; nothing else is opened or sent. You cannot remove the splash in the Free Edition.

The splash is built so that it does not take the keyboard from your startup form and has no taskbar button. These two points have not yet been tried on a Windows PC for this version, so check them when you test your `.exe`.

Publish from the command line

The `mcbasic` tool does the same:

```
mcbasic publish MyApp.mcbproj --target win-x64 --output ~/Desktop/Release
mcbasic publish MyApp.mcbproj --target win-arm64 --output
~/Desktop/Release
```

`--target` takes `win-x64` or `win-arm64` (it is another spelling of `--rid`, which still works; give one of the two, not both). The command writes `MyApp.exe` into the folder named by `--output`, and never overwrites an existing file. Without `--output`, the file stays in the build cache, which McBasic™ clears after a day. `--format package` is for Mac plugins only and is refused for Windows (MCB2301).

Check your work

- You can name the two Windows platforms and say which one to use when you do not know the PC.
- You know that the file is unsigned and what to click when SmartScreen asks.
- You know that every service in the table "What works on Windows" has a Windows version, the `VideoPlayer` included, and that on Windows a `VideoPlayer` plays project resources only (MP4 with H.264 plays on every PC).
- You know that `Reports.Print` opens the Windows **Print** dialog and returns `False` when the person cancels.
- You tried the sample **WindowsServicesTest** on a Windows PC.
- You know that nothing can be drawn over a `VideoPlayer` on Windows, and that its controls bar sits under the video.
- You know that a warning MCB2406 does not stop a publish.
- You tested the `.exe` on a Windows PC, not only on the Mac.

Tips and tricks

- **Imported projects bring their icon.** A Visual Basic 6 project whose `IconForm` has an `Icon` keeps it as `Resources/AppIcon.ico`, and the Windows `.exe` uses it. To give the Mac app the same icon, choose a square PNG version of it in **Project > Project Settings…**.

- **Publish Windows · x64 when in doubt.** It runs on ARM PCs too. Choose **Windows · ARM64** only when the app must be as fast as possible on ARM.
- **Use MP4 with H.264 for video.** It plays on every Windows PC, as on the Mac. HEVC and VP9 video need extensions from the Microsoft Store on Windows. Store the video as a project resource: HTTPS and YouTube sources play on the Mac only.
- **Keep labels and buttons next to a `VideoPlayer`, not on top of it.** On Windows the video covers anything placed over it.
- **Read the MCB2406 warnings before you copy the file.** There are none in this version; if a later version shows one, wrap the named call with `On Error` or `Try ... Catch` and show the message in a label.
- **Try WindowsServicesTest first.** If a service fails on one PC in that sample, it will fail in your app too, and you know it is the PC (for example a missing audio or video codec), not your code.
- **Use WAV or MP3 for sound.** They play on every Windows PC. Other formats depend on the codecs installed there.
- **Write shortcuts with Ctrl in the Menu Editor.** On Windows they show and work as Ctrl, on the Mac as Command.
- **Do not count on an `mnuAbout` item for the About box.** McBasic™ always adds its own About box (Help ▸ About on Windows), so use `mnuAbout` only for extra information.
- **Try a report with Microsoft Print to PDF first.** It is on every Windows PC: print to a file and look at the pages before you use paper.
- **Use `Reports.ExportPdf` for a PDF that people search or copy from.** A PDF made by printing holds pictures of the pages.
- **Keep the Classic appearance for an imported Visual Basic 6 project.** It looks the same on Windows as the original.
- **Test the exe from a folder that is not on your Desktop.** Run it from a normal folder such as `C:\Apps`, so that you see what a user sees.
- **Allow five seconds for the splash when you time the first launch.** The Free Edition splash shows over your first form for five seconds while your form starts behind it.
- **Rename the file freely.** The `.exe` does not depend on its name. Keep the extension.
- **Do not send an unsigned exe by mail.** Mail servers often block it. Share a link or a shared folder instead, and tell people about the SmartScreen steps.

Chapter 21 — Publish an app for Linux

The apps you make with McBasic™ can also run on Linux. You publish a Linux app from the same project, on the same Mac, with no Linux machine needed to build it. The result is one file, `<Name>`, with no extension, that you copy to a Linux desktop and start.

This first Linux version covers forms, controls, dialogs, the clipboard, menus, files, settings, SQLite and reports. A few services wait for a later version: this chapter lists them, shows what the Check says about them, and explains how to try every service on a real Linux desktop with the sample **WindowsServicesTest**, which works for Linux too.

What you get

- **One self-contained file.** `MyApp` carries your program, the McBasic™ runtime and the .NET runtime inside it. The Linux machine needs nothing installed except a desktop. A small app is about 50 MB.
- **Two kinds of machine.** Choose **Linux · x64** for ordinary PCs and servers with a desktop. Choose **Linux · ARM64** for ARM machines such as a Raspberry Pi 4 or 5 with a 64-bit system, or a virtual machine on an Apple silicon Mac. Unlike Windows, a Linux x64 file does not run on an ARM machine: publish the one that matches.
- **The same look as on the Mac.** Both looks are drawn by McBasic™ itself. A project with the **Classic** appearance looks the same on Linux as on the Mac and on Windows: same colors and bevels. Its font is Microsoft Sans Serif when that font is installed; otherwise Linux uses Liberation Sans or DejaVu Sans, which are on almost every desktop and have the same sizes. A project with the **macOS** appearance keeps its Mac-styled controls and uses the desktop's default font. You choose the appearance in **Project > Project Settings…**.
- **No icon and no signature.** A Linux executable carries no icon and is not signed. The file manager shows it with the generic program icon. The icon chosen in **Project > Project Settings…** is used by the Mac app and the Windows `.exe` only; the publish shows no warning about it.

Only a **desktop application** project can be published for Linux. A Plugin or a UserControl library cannot.

Publish step by step

1. Save the project and choose **Check**. Fix every error.
2. Choose **Project > Publish Local App**. McBasic™ checks the project and prepares the app, then shows the Save panel.
3. In the panel, open the **Platform:** pop-up. It lists:

• **macOS** (the Mac app, `MyApp.app`);

• **Windows · x64** and **Windows · ARM64** (Chapter 20);

• **Linux · x64**;

• **Linux · ARM64**.

4. Choose a Linux platform. The proposed name becomes `<Project>`, with no extension, and the **Build:** pop-up disappears, because **Compatible** and **Compact** apply to the Mac app only. The panel shows the note *Unsigned: if Linux does not start the file, run chmod +x <Name> once.*
5. Choose the folder and the name, then confirm. The folder must be outside the `.mcbproj`. If a file with that name exists, McBasic™ asks before it replaces it.
6. The **Build Log** reports `Published MyApp.` **Project > Show Published App** shows the file in the Finder.

McBasic™ remembers the platform you chose for each project on this Mac. The choice is not stored in the project.

A failed publish leaves nothing behind: you never get a half-written file.

What works on Linux

A Linux app uses the Linux version of each service. Your BASIC code does not change.

Service	On Linux
`MsgBox`	A message box drawn by McBasic™: the prompt with its icon (`vbCritical`, `vbQuestion`, `vbExclamation`, `vbInformation`), the buttons of the chosen set at the bottom right, the default button focused. `MsgBox` returns the same values as on the Mac (`vbOK` 1, `vbCancel` 2, `vbAbort` 3, `vbRetry` 4, `vbIgnore` 5, `vbYes` 6, `vbNo` 7). Esc and the close button answer **Cancel** when the set has it, **No** for

Service	On Linux
	`vbYesNo`, **OK** for `vbOKOnly` and **Ignore** for `vbAbortRetryIgnore`, as Visual Basic does.
`InputBox`	A small window with the prompt, a text box that holds the default text selected, and **OK** and **Cancel** buttons. **Cancel** returns an empty string.
File and folder dialogs	`Dialogs.OpenFile`, `Dialogs.SaveFile`, `Dialogs.SelectFolder` and `ShowOpen`, `ShowSave`, `ShowFolder` of the CommonDialog use the dialogs of the Linux desktop (GTK or the desktop portal). Your `Filter` text such as `Text files
Color, font and printing dialogs	Not yet. `ShowColor`, `ShowFont`, `ShowPrinter` and `ShowPageSetup` stop with error 5 and the message MCB4415 (see "What the Check says about Linux").
Clipboard	`Clipboard.SetText`, `Clipboard.GetText`, `Clipboard.HasText` and `Clipboard.Clear`, through the desktop's clipboard.
Web links	`Platform.OpenUrl` opens `http`, `https` and `mailto` links in the default program, as on the Mac.
Menus	The menus of a form appear inside the window, at the top, as on Windows. The window grows by the height of the menu bar, so the area for your controls keeps the size you designed. Command shortcuts become **Ctrl** (a menu item with Shortcut `Ctrl+S` shows `Ctrl+S` and answers to Ctrl+S), and the underlined letters open the menus with Alt.
About and Exit	The McBasic™ About box is always there and an `mnuAbout` item does not replace it. It is **Help ▸ About** `<your app>` at the end of the menu bar; if the form has no Help menu, McBasic™ adds one. Linux windows have no system menu with an About item, so a form without menus gets a small menu bar with only **Help ▸ About** `<your app>`; the window grows by that bar. **File ▸ Exit** is added only when the form has no File menu of its own.
Look and font	The **macOS** appearance uses the desktop's default font. **Classic** uses Microsoft Sans Serif when installed, else Liberation Sans or DejaVu Sans.
MDI forms	Ctrl+Tab and Ctrl+Shift+Tab (or Ctrl+F6 and Ctrl+Shift+F6) switch between the child windows, and Ctrl+F4 closes the active child.

Service	On Linux
`AudioPlayer`	Not yet. `Load` accepts the file and reports MCB4415 once in the app's log; `Play` returns `False`, `Playing` stays `False` and nothing is heard. The app keeps running.
`VideoPlayer`	Not yet. The control shows a grey area with the text *Video is not available on Linux yet in this McBasic version.* Setting `Source` keeps the value, raises the `Failed` event with the MCB4415 message, sets `Error` and the state `Error`, and, when set in code, stops the statement with error 5.
`Reports.Print`	Opens the report as a PDF in the desktop's default viewer, where you print it with the viewer's own print command. `Reports.Print` returns `True` as soon as the viewer has taken the file. See "Print a report on Linux" below.
Windows and dialogs	`Opacity`, `TopMost`, `WindowState`, the window styles and the modal dialogs work through the desktop's window manager; a few desktops ignore `TopMost` or the position of a new window.

The services run on the same thread as your code and wait for the answer, as on the Mac. A dialog belongs to the window that was active when it opened.

What the Check says about Linux

A Linux publish runs the normal Check and then one more pass, which looks for the services that a Linux app does not have yet:

- `CommonDialog.ShowColor`, `CommonDialog.ShowFont`, `CommonDialog.ShowPrinter`, `CommonDialog.ShowPageSetup`;
- `AudioPlayer`;
- `VideoPlayer`.

For every use of one of them the pass adds a **warning** to the Build Log, with the member and the line:

```
MCB2406: AudioPlayer is not available on Linux yet in this McBasic
version; the app is built, but this call raises MCB4415 on Linux.
```

A warning does not stop the publish: the file is still produced. At run time on Linux that one call stops with error 5 and the message

```
MCB4415: CommonDialog.ShowColor is not available on Linux yet in this
McBasic version.
```

which `On Error` or `Try ... Catch` can handle. The `AudioPlayer` and the `VideoPlayer` are gentler: they report MCB4415 once and the app keeps running

(see the table above). The pass runs only when you publish for Linux; a normal Check, a Mac publish and a Windows publish never show these warnings.

Print a report on Linux

`Reports.Print` works in a Linux app with the same report file as on the Mac. Your code does not change. Linux has no McBasic™ print dialog yet, so the call hands the report to the desktop:

1. McBasic™ writes the report as a PDF into a private folder of the running

app (readable by the user only; the file is deleted when the app exits).

2. The desktop's default PDF viewer opens the file.
3. In the viewer, use its own **Print** command to choose the printer, the

pages and the copies.

`Reports.Print` returns `True` as soon as the viewer has taken the file, so it cannot tell whether the person printed or closed the viewer. When no viewer is installed, or the desktop cannot open the file, the call stops with error 5 and a message that starts with `Linux could not open the report PDF:`. Use `On Error` or `Try ... Catch` to show the message in a label.

For a PDF the person keeps, use `Reports.ExportPdf`, as on the other systems.

Try every service: the WindowsServicesTest sample

The sample **WindowsServicesTest** (`samples/WindowsServicesTest.mcbproj`), described in Chapter 20, works for Linux too. Its main form has one button per service, and each button shows its result in the label at the top of the form.

1. Open the sample, choose **Project > Publish Local App**, pick **Linux ·

ARM64 **(or** Linux · x64**) and copy the file to the Linux machine.

2. Start it (see "First launch on Linux") and click each button.

What you should see differs from Windows in these rows:

Button	On Linux
MsgBox …	The McBasic™ message box with the right icon and buttons; the label says, for example, `Yes/No/Cancel returned 7`. Esc on the Yes/No/Cancel box returns 2.
ShowOpen, **ShowSave**, **ShowFolder**	The desktop's own file dialogs.
ShowColor, **ShowFont**, **ShowPrinter**, **ShowPageSetup**	The label shows the MCB4415 message; nothing opens.

Button	On Linux
Play, **Pause**, **Stop**, **Position 0.5 + Play**, **Volume 0.5**	No sound; the label shows the player state, which stays stopped.
Open second form	Both windows stay usable. The second form, which has no menus, shows a menu bar with only **Help ▸ About WindowsServicesTest**.
Open form without box	No close button; a menu bar with only **Help ▸ About WindowsServicesTest**. **Close** closes it.
Print report	The default PDF viewer opens the one-page report; the label says `Report sent to the printer`.
Video …	The grey placeholder with its text; the label shows the video state `Error`.

The menu bar, the shortcuts (Ctrl+O, Ctrl+S, Ctrl+C, Ctrl+V), the Alt letters and **Help ▸ About** behave as on Windows. To try the MDI keys, publish the **MdiForms** sample for Linux too.

When the publish stops: MCB2302

If the Linux .NET packs are missing from this McBasic™ installation, the publish stops before it starts, with:

```
MCB2302: Publishing for Linux needs the Linux .NET packs, which are
missing from this McBasic toolchain. Reinstall or update McBasic.
```

Nothing is wrong with your project. Install McBasic™ again, or update to a version that includes the Linux packs. Mac and Windows publishing are not affected.

Two related messages come from the same command. MCB2301 says *publishing for Linux requires a DesktopApp project* when you choose a Linux platform for a Plugin or a library. MCB1001 says that a Linux target cannot use a Mac publication profile.

First launch on Linux

Copy `MyApp` to the Linux machine with a USB drive, a shared folder or a download. Then:

1. If the file manager or the terminal refuses to start it, the copy lost the

permission to run (a download or a FAT drive does that). Open a terminal in the folder and type once:

``text chmod +x MyApp ``

2. Start it from the file manager (double-click, or right-click ▸ **Run**), or

from the terminal with `./MyApp`.

Linux shows no confirmation for an unsigned file. The first start takes a little longer: the app unpacks its files on first use into a folder under the user's home (`~/.net`), and then starts from there. Later starts reuse that folder. You can delete it at any time; the app makes it again.

The app needs a desktop (an X11 session, or a Wayland session with X11 support, which every common desktop has). It does not run from a terminal without a desktop, and it does not run over SSH without a display.

The launch splash on Linux

A Linux app published with McBasic™ Free Edition opens with the same launch splash as a Mac app (see "The Free Edition launch splash" in Chapter 17): the McBasic™ logo on a blue panel, **Made with McBasic™**, the **Free Edition** badge, your app's name, version, build and copyright, a **mcbasic.com** link button and a countdown bar. On Linux the card is always light and closes after its five seconds without a fade.

As on the Mac, it steps aside while one of your app's dialogs is open and comes back for the rest of its five seconds when the dialog closes, and a click on **mcbasic.com** opens the website in the default browser; nothing else is opened or sent. You cannot remove the splash in the Free Edition.

Publish from the command line

The `mcbasic` tool does the same:

```
mcbasic publish MyApp.mcbproj --target linux-x64 --output
~/Desktop/Release
mcbasic publish MyApp.mcbproj --target linux-arm64 --output
~/Desktop/Release
```

`--target` takes `linux-x64` or `linux-arm64` (it is another spelling of `--rid`, which still works; give one of the two, not both). The command writes `MyApp` into the folder named by `--output`, with the permission to run already set, and never overwrites an existing file. Without `--output`, the file stays in the build cache, which McBasic™ clears after a day. `--format package` is for Mac plugins only and is refused for Linux (MCB2301).

Check your work

- You can name the two Linux platforms and know that, unlike Windows, each

file runs only on its own kind of machine.

• You know that the file has no extension, no icon and no signature, and what to type when Linux does not start it.

• You know which services wait for a later version (the color, font and printing dialogs, the `AudioPlayer` and the `VideoPlayer`) and that a warning MCB2406 does not stop a publish.

• You know that `MsgBox` and `InputBox` are drawn by McBasic™ and return the same values as on the Mac.

• You know that `Reports.Print` opens the PDF viewer and returns `True` once the viewer has the file.

• You tried the sample **WindowsServicesTest** on a Linux desktop.

• You tested the file on a Linux machine, not only on the Mac.

Tips and tricks

- **Publish the platform that matches the machine.** A Linux x64 file does not start on an ARM machine, and the other way round. When in doubt, ask for the output of `uname -m`: `x86_64` means x64, `aarch64` means ARM64.
- **Keep `chmod +x` in your delivery note.** Downloads and FAT-formatted drives drop the permission to run. One line in your note saves a support call.
- **Wrap the services that are not there yet.** `On Error Resume Next` or `Try ... Catch` around `ShowColor`, `ShowFont`, `ShowPrinter` and `ShowPageSetup` lets the same project run on all three systems; show the MCB4415 message in a label and offer an alternative, such as a fixed list of colors.
- **Read the MCB2406 warnings before you copy the file.** Each one names a call that will raise MCB4415 on Linux, with its line.
- **Use `Reports.ExportPdf` when the person must keep the file.** `Reports.Print` opens a temporary PDF that disappears when the app exits.
- **Write shortcuts with Ctrl in the Menu Editor.** On Linux and Windows they show and work as Ctrl, on the Mac as Command.
- **Do not count on an `mnuAbout` item for the About box.** McBasic™ always adds its own About box (Help ▸ About on Linux), so use `mnuAbout` only for extra information.
- **Test on the desktop people use.** GNOME, KDE Plasma and Xfce draw window frames, file dialogs and fonts differently; the controls inside your form look the same on all of them.
- **Install Microsoft Sans Serif for a Classic app only if you must.** Liberation Sans has the same metrics and is already there; the form keeps its layout.

- **Try WindowsServicesTest first.** If a service fails on one machine in that sample, it will fail in your app too, and you know it is the machine (for example a missing PDF viewer), not your code.
- **Rename the file freely.** The executable does not depend on its name.
- **Do not send the file by mail.** Mail servers often block executables. Share a link or a shared folder instead, and add the `chmod +x` line.

Appendix 1 — Quick reference

Use this appendix when you remember the job but not the exact syntax. It is meant to be quick to scan after you finish the lessons.

Common workflow

```
Select project -> Design form -> Name controls -> Write event code
-> Save -> Check -> Run -> Test -> Repeat
```

When the project is ready to deliver, continue with **Project > Publish Local App** or the **Publish** command in the top strip. Choose the `.app` name and destination in the Save panel (outside the project), then test the published copy outside the IDE.

IDE command map

If you want to...	Use...
start after launching McBasic™	the **Welcome to McBasic** window: **New Project…**, **Create with AI…**, **Open a Project…**, **Open a Sample…**
create or reopen a project	**New Project** or **Open** (or **File > Open Recent**)
reopen a recent project	**Recent Projects** in the Welcome window (double-click, or select and press Return)
start from a copy of a manual sample	**Open a Sample…** in the Welcome window
minimize the front window of an app you built	**Window > Minimize** (Command–M; greyed out when the form cannot minimize)
close the project and return to the Welcome window	**File > Close** (Command–W)
show the Welcome window again	**Window > Welcome to McBasic** (Shift–Command–1)
show the previous or next document tab	**Window > Show Previous Tab** (Shift–Command–[) or **Window > Show Next Tab** (Shift–Command–])

If you want to...	Use...
reach a document whose tab does not fit	» (**More Documents**) at the right of the document tabs
save the current layout and code	**Save**
add another application window	**Add** in the toolbar
add the main item to a control project	**Add** in the toolbar
line up selected controls	**Align**
build the form's menu bar	**Design > Menu Editor…** (Command–E), or **Menu Editor** on the Ribbon **Design** tab
turn an old Menu component into real menus	**Convert to Menu Editor…** in its Inspector or in the context menu of its tray button
switch between the Mac, Classic and Ribbon toolbars	**McBasic > Settings… > Appearance > Toolbar**
find a member while typing	completion or **Edit > Complete BASIC**
see a procedure's arguments	**Edit > Show BASIC Signature Help**
find where a name is used	**Edit > Find References**
go to where a name is declared	**Edit > Jump to Definition (Shift–F2), or Command–click the name**
review a larger proposed edit	**Review Structured Form Patch** or **Review BASIC Procedure Patch**
validate before running	**Check**
observe the application	**Run**
prepare a standalone local app	**Project > Publish Local App**
ask the AI assistant	**AI** in the toolbar, or **McBasic > Show AI** (Shift–Command–L)
add a class	**Project > Add Class…**, or the arrow next to **Add**
copy a module or form from another project	**Project > Add Existing File…** (Command–D)
choose or change the AI provider	**McBasic > AI Provider Profile…** or the gear in the chat panel
choose whether AI changes apply automatically	**Apply AI changes:** in the AI chat section of AI Provider Settings

If you want to...	Use...
continue a paused AI plan later	**Edit > Show Paused AI Plan**, check the proposed chip and add the plan's forms and modules with **+ Add**, then **Resume** on the plan card
see the project's latest AI chat and stage state	**Edit > Open Latest AI Activity**

AI shortcuts

Action	Shortcut or control
show or hide the AI panel	Shift–Command–L
send the message	Command–Return (while the message field has focus)
stop a running reply, or pause a running plan	Esc, or the Stop button
edit your last message	Up Arrow in an empty message field, or **Edit**
send your last message again	**Retry**
choose what the chat sends	**+ Add**, and the **×** on a chip to remove it
see exactly what will be sent	**Show exact content** in the panel header
start an AI plan	**Approve and run** on the plan card
continue a paused plan	**Resume** (or **Retry**) on the plan card
end a plan	**Stop plan** on the plan card (offered in every pause)
undo the most recent AI change	**Undo** on its card, or **Edit > Undo** (one step per stage or change)
see what an AI change did	**N files changed** on its card

Common control names

Prefix	Typical control
`frm`	Form
`txt`	TextBox, PasswordBox, or other text input
`cmd`	CommandButton

Prefix	Typical control
`lbl`	Label
`chk`	CheckBox
`opt`	OptionButton
`lst`	ListBox
`cbo`	ComboBox
`grd`	DataGrid
`pnl`	Panel
`tmr`	Timer
`tvw`	TreeView
`img`	Image, ImageList, or media image
`aud` / `vid`	AudioPlayer / VideoPlayer
`dlg`	CommonDialog

These are naming habits, not magic rules. Use a name that makes the code easy to understand.

Common event pattern

```
Private Sub control_event()
    ' Read input.
    ' Check the value.
    ' Show a useful result.
End Sub
```

For an exact event signature, select the object and event in the IDE. The families below are a memory aid, not a replacement for the generated stub:

```
' A cancellable close.
Private Sub Form_Closing(ByRef Cancel As Boolean)
End Sub

' Keyboard input with an explicit handled flag.
Private Sub txtName_KeyDown( _
    ByVal KeyCode As KeyCode, _
    ByVal Shift As KeyModifiers, _
    ByRef Handled As Boolean)
End Sub

' A cell edit that can be refused.
Private Sub grdBooks_CellBeginEdit( _
    ByVal RowIndex As Integer, _
    ByVal ColumnIndex As Integer, _
    ByRef Cancel As Boolean)
End Sub
```

Use `MouseButton` for one button, `MouseButtons` for a set of buttons, `DragData` for dropped content, and `DragEffect` for the accepted drop action. The editor's generated signature is the final authority for the selected component.

Common constants

The VB6 constants are built in. The language reference appendix lists all of them with their values.

Need	Constants
`MsgBox` buttons	`vbOKOnly`, `vbOKCancel`, `vbYesNo`, `vbYesNoCancel`, `vbRetryCancel`, `vbAbortRetryIgnore`
`MsgBox` icon and default	`vbCritical`, `vbQuestion`, `vbExclamation`, `vbInformation`; `vbDefaultButton1` to `vbDefaultButton4`
`MsgBox` result	`vbOK`, `vbCancel`, `vbYes`, `vbNo`, `vbAbort`, `vbRetry`, `vbIgnore`
Text	`vbCrLf`, `vbNewLine`, `vbCr`, `vbLf`, `vbTab`, `vbNullString`
Colors (`"#RRGGBB"` text)	`vbBlack`, `vbWhite`, `vbRed`, `vbGreen`, `vbBlue`, `vbYellow`, `vbCyan`, `vbMagenta`, `RGB(r, g, b)`, `QBColor(n)`
Keys in `KeyDown`	`vbKeyReturn`, `vbKeyEscape`, `vbKeyTab`, `vbKeyBack`, `vbKeyDelete`, `vbKeyLeft`, `vbKeyA` to `vbKeyZ`, `vbKeyF1` to `vbKeyF16`
`Shift` argument	`vbShiftMask`, `vbCtrlMask`, `vbAltMask`; test `(Shift And vbShiftMask) = vbShiftMask`
Text comparison	`vbBinaryCompare`, `vbTextCompare`

```
Private Sub txtName_KeyDown( _
    ByVal KeyCode As KeyCode, _
    ByVal Shift As KeyModifiers, _
    ByRef Handled As Boolean)
    If KeyCode = vbKeyEscape Then
        txtName.Text = ""
        Handled = True
    End If
End Sub
```

Common questions

Why does Run stop?

Choose Check and read the first diagnostic. Fix the named problem, save, and Check again.

Why does my code not find a control?

Check the control's Name in Design view. The Caption is not the code name.

Why did my export not replace the old file?

Some samples deliberately refuse to overwrite an existing file. Rename or remove the old file only after confirming that it is safe to do so.

Why is a feature marked Limited?

The feature is available in the current release, but it has a stated scope or missing acceptance evidence. Follow the release notes before relying on it in a customer product.

Glossary starter

Term	Meaning
app	the program the user runs
chip	a document selected as AI context; the chips are what the chat may send
component	a control or service added to a project
event	something that happens and can start code
form	a window designed in McBasic™
handler	the procedure that responds to an event
Inspector / Properties	the area used to change a selected object
project	the files that describe one McBasic™ application
running state	the part of the app that is active while the user uses it
source	the BASIC code and layout files you author

Use this glossary together with the component and language references when a term is unclear. The Inspector and the installed release determine the exact property and menu names available in your project.

Tips and tricks

- **Command–E opens the Menu Editor.** It is the shortcut of **Design > Menu Editor…**; double-clicking the menu strip in the designer does the same.
- **Command–M minimizes an app's window.** The automatic **Window** menu of every app you build carries it.
- **Use this chapter as a map.** The **IDE command map** and **AI shortcuts** tell you where a command lives.
- **Go to a declaration quickly.** Use **Edit > Jump to Definition** (Shift–F2) or Command–click the name.
- **Type a command first to change mode for one message.** For example `/fix` or `/plan`.
- **Use Retry to send your last message again.** It uses the current chips.

Appendix 2 — Standard component reference

Use this appendix after the guided chapters. It explains what each standard component is for and gives you a place to start. The table follows the current standard catalog used by the IDE. Read the selected component's Inspector and event list when you need the exact members for your installed release.

How to read this table

- **Visual** means the component appears on a form.
- **Nonvisual** means it appears in the component area, not as a box on the form canvas.
- **Default event** is the event McBasic™ normally opens first for that type.
- A dash means the component has no single default event.

Component	Type	Default event	Good first use
TextBox	Visual	Change	Let the user enter text.
PasswordBox	Visual	Change	Let the user enter hidden text.
MaskedTextBox	Visual	Change	Guide input with a mask.
SearchBox	Visual	Search	Start a search action.
NumericBox	Visual	Change	Enter a number with limits.
DatePicker	Visual	Change	Choose a calendar date.
TimePicker	Visual	Change	Choose a time.
CommandButton	Visual	Click	Start an action.
CheckBox	Visual	Click	Turn an option on or off.
OptionButton	Visual	Click	Choose one option in a group.
ToggleSwitch	Visual	Change	Show an on/off setting.
ComboBox	Visual	Change	Choose an item or enter a value.
ListBox	Visual	Change	Show one or more choices.

Component	Type	Default event	Good first use
ColorPicker	Visual	Change	Choose a color.
Slider	Visual	Change	Pick a value in a range.
Label	Visual	-	Show text or a message.
Badge	Visual	-	Show a short status or category.
Image	Visual	-	Show a picture or icon.
ProgressBar	Visual	-	Show progress.
Gauge	Visual	Change	Show a value as a gauge.
Sparkline	Visual	Change	Show a small value trend.
StatusBar	Visual	-	Show status at the bottom of a window.
Panel	Visual	-	Group controls.
FlowLayoutPanel	Visual	-	Arrange children in a flow.
Frame	Visual	-	Group controls with a caption.
Expander	Visual	Expanded	Show or hide a group of controls.
TabControl	Visual	Change	Organize pages of related content.
TabPage	Visual	-	Hold controls inside a TabControl.
ScrollContainer	Visual	-	Show content larger than its area.
SplitContainer	Visual	-	Give two areas an adjustable divider.
DataGrid	Visual	SelectionChanged	Show rows and columns of data.
ListView	Visual	SelectionChanged	Show a compact list with columns or icons.
TreeView	Visual	Change	Show parent and child items.
ToolBar	Visual	Click	Keep common actions near the top of a form.
Timer	Nonvisual	Timer	Run a repeated timed action.
Menu	Nonvisual	Click	Add application menu items.
ContextMenu	Nonvisual	Click	Add a menu for a target control.
ValidationProvider	Nonvisual	-	Connect validation feedback to controls.

Component	Type	Default event	Good first use
UpDown	Visual	Change	Nudge a numeric value.
MonthView	Visual	Change	Show or choose a month view.
DTPicker	Visual	Change	Choose a date/time value.
ImageList	Nonvisual	-	Supply images to image-aware controls.
ImageCombo	Visual	Change	Combine an image with a choice list.
TabStrip	Visual	Change	Show a compact strip of tabs.
FlatScrollBar	Visual	Change	Expose a flat scroll position.
Line	Visual	-	Draw a line or separator.
Shape	Visual	-	Draw a geometric shape.
CommonDialog	Nonvisual	-	Open a standard file, color, font, or print dialog.
Animation	Visual	FrameChanged	Play frames from an ImageList.
AudioPlayer	Visual	Completed	Play audio and show transport state.
VideoPlayer	Visual	Completed	Play supported video sources.

The catalog also contains `FlowLayoutPanel`, `Frame`, `Expander`, `ScrollContainer`, and `SplitContainer` as containers, plus `ToolBar` and `ContextMenu` as command surfaces. They are included above in their normal positions; the table is a quick lookup, not a recommended build order.

Form properties

Select the form to see these in the Properties panel. A property added after the original set is written to the `.form.json` file only when you change it from its default.

Property	Type	Default	Values and notes
`Caption`	String	empty	Window title; a form without one shows McBasic.
`Font`, `ForeColor`, `BackColor`	String	theme	Defaults for the controls; `BackColor` empty is #F0F0F0. A color is `#RRGGBB`, a

Property	Type	Default	Values and notes
			system color name such as `ButtonFace`, or empty (Chapter 2, "System colors").
`Width`, `Height`	Double	520, 260	Content size in DIP.
`MinWidth`, `MinHeight`, `MaxWidth`, `MaxHeight`	Double	200, 120, 100000, 100000	Resize limits.
`MDIForm`, `MDIChild`, `MDIClientTop`	Boolean, Boolean, Double	False, False, 0	MDI hosting; set in the designer only.
`KeyPreview`	Boolean	False	Form keyboard events first. A form whose controls cannot take the focus gets its keyboard events without it.
`StartUpPosition`	String	`CenterOwner`	`Manual`, `CenterScreen`, `CenterOwner`; read-only in code.
`Left`, `Top`	Double	0	Window position for `Manual`, in DIP.
`BorderStyle`	String	`Sizable`	`Sizable`, `Fixed`, `Dialog`, `None`, `ToolWindow`.
`ControlBox`, `MinButton`, `MaxButton`	Boolean	True	Title-bar buttons; greyed out where `BorderStyle` has no such button.
`WindowState`	String	`Normal`	`Normal`, `Minimized`, `Maximized`.
`TopMost`	Boolean	False	Floating window.
`Opacity`	Double	1	0–1.
`Enabled`	Boolean	True	False disables the controls. A `BorderStyle None` form has no title bar, so while disabled only

Property	Type	Default	Values and notes
			Cmd-Q or code closes it.
`Visible`	Boolean	True	Code only: Show/Hide.
`Tag`	String	empty	Any value in code.
`Cursor`	String	`Default`	Same values as controls.
`AccessibleName`	String	empty	Empty uses the Caption.
`Picture`	String	empty	An image resource name; photo button picks it.
`PictureSizeMode`	String	`Normal`	`Normal`, `Stretch`, `Zoom`, `Center`, `ZoomCrop`.

Classic properties added in Build 817

TextBox

Property	Values	Notes
`ScrollBars`	`None`, `Horizontal`, `Vertical`, `Both` (default)	MultiLine only. A bar appears when the text overflows; the caret still scrolls a disabled axis.
`CharacterCasing`	`Normal` (default), `Upper`, `Lower`	Converts typed, pasted and code-set text, like `UCase`/`LCase`.
`Locked`	Boolean, code only	The same as `ReadOnly`: macOS draws a read-only TextBox like an editable one.
`SelStart`, `SelLength`, `SelText`	code only	The selection, in UTF-16 units; assigning `SelText` replaces it.

With **WordWrap** on, the text wraps at the box's width, so a MultiLine TextBox shows no horizontal bar whatever `ScrollBars` says. The code-only members are listed under "Methods and members in code" below.

Label

Property	Values	Notes
`AutoSize`	False (default) / True	At run time the Label fits its Caption, and refits when the Caption, Font or WordWrap changes. With WordWrap the width stays and the height grows. The designer keeps the frame you drew.
`BorderStyle`	`None` (default), `Solid`	A thin gray frame. VB6 `FixedSingle` is `Solid`.

CommandButton

Property	Values	Notes
`Picture`	An image resource name	Shown with the Caption; the photo button in the Properties panel lists the project's images. Never enlarged; shrunk to fit. The designer shows a photo placeholder in its place.
`PictureAlignment`	`Left` (default), `Top`, `Right`, `Bottom`	The picture's side of the Caption.

Access keys (`&Save` in VB6) do not exist on macOS, which has no keyboard mnemonics: an `&` in a Caption is shown as written. Give important buttons a menu command with a keyboard shortcut, or make them the form's Default or Cancel button.

ComboBox and ListBox

Property	Control	Notes
`MaxLength`	ComboBox	Longest text the user can type in the `DropDown` style; 0 means no limit. Text set in code is kept whole.

Property	Control	Notes
`AutoComplete`	ComboBox	In the `DropDown` style, typing `bo` shows `Bologna` with `logna` selected. Keep typing to replace it; Backspace removes the completion. Items longer than `MaxLength` are never offered.
`ListIndex`	both, code only	The VB6 name of `SelectedIndex`: 0 for the first item, -1 for none.
`ItemData`	both, code only	A whole number kept with each item, read and written with an index.

`ItemData` keeps one whole number (Long) with each item. It is available in code only. Its index starts at 0, like `ListIndex`:

lstCustomers.Items.Add "Ada Lovelace"
lstCustomers.ItemData(lstCustomers.Items.Count - 1) = customerId customerId = lstCustomers.ItemData(lstCustomers.ListIndex)

The example assumes the list is not `Sorted`: in a sorted list a new item takes its sorted place, not the last one. The number stays with its item when the list is sorted or another item is removed, and `Items.Clear` removes it. An index outside the list raises error 381. A list filled from a `DataSource` has no `ItemData`: use `ValueMember` and `SelectedValue` instead.

The VB6 ListBox `MultiSelect` property is `SelectionMode` in McBasic™: `0 - None` is `Single`, `1 - Simple` is `Multiple`, `2 - Extended` is `Extended`.

DataGrid

Property	Values	Notes
`AlternatingRowBackColor`	`#RRGGBB`, a system color name or empty (default)	Every second row gets this background.
`GridLines`	`None`, `Horizontal`, `Vertical`, `Both` (default)	Lines between cells; `Both` is the look of earlier releases.
`FrozenColumns`	0 (default)–1000	Leftmost columns that stay in place when the grid scrolls sideways.
`ShowRowNumbers`	False (default) / True	Row numbers 1, 2, 3… in a header column.

Property	Values	Notes
		`HeadersVisible` still controls the column headers.

Row numbers are display positions, not row numbers of the data: they follow the grid's sort order and start again at 1 after a filter and on each page. The `KeyColumn` is never shown in the grid, so `FrozenColumns` counts only the columns the grid shows.

The designer previews the stripes, the lines and the row-number column.

In an editable grid, one click on a Boolean (check box) cell toggles it; you do not need to select the cell and start an edit first. `CellBeginEdit`, `CellValueChanged` and `CellEndEdit` each fire once for the click, and the click never produces Null. A grid whose `IsReadOnly` is True ignores the click.

Classic properties added in Build 863

Timer

Property	Values	Notes
`Interval`	0 to 2147483647 milliseconds	0 is valid: the timer never fires while `Interval` is 0, whatever `Enabled` says. A positive value starts it again when `Enabled` is True.

Image

Property	Values	Notes
`BorderStyle`	`None` (default), `Solid`	`Solid` draws a thin frame around the picture. VB6 `Fixed Single` is `Solid`.

In code, `BorderStyle` also accepts the Visual Basic 6 numbers on every control whose `BorderStyle` is `None` or `Solid`: `imgDice.BorderStyle = 1` gives `Solid` and `0` gives `None`. Reading it back always gives the name, `"None"` or `"Solid"`.

Panel

Property	Values	Notes
`Picture`	An image resource name	Drawn at the top-left corner at its natural size, clipped to the panel, over `BackColor` and behind the child controls. Empty shows only `BackColor`.

Pick the image with the photo button in the Properties panel, as for a Form or a CommandButton. In code, `pnlBoard.Picture = "Board"` changes it; a name that is not an image resource of the project raises error 481 (Invalid picture) and keeps the current picture.

ImageList

Member	Values	Notes
`ListImages(i).Picture`	String, code only	The resource name of the image at the one-based index `i`, or with the Key `i` (a String, any case).
`ListImages.Item(i).Picture`	String, code only	The same, written with `Item`.
`ListImages.Count`	Long, code only	The number of images, like `Count`.

The `Picture` value can be assigned to the `Picture` of an Image, a PictureBox, a CommandButton or a Form, as in Visual Basic 6:

```
imgDice.Picture = ImageList1.ListImages(face).Picture
cmdPlay.Picture = ImageList1.ListImages("play").Picture
```

An index outside 1 to `ListImages.Count` raises error 9 (Subscript out of range).

Choose by user need

The user must enter something

Start with `TextBox`, `PasswordBox`, `NumericBox`, `DatePicker`, `TimePicker`, or `MaskedTextBox`. Ask what kind of value is allowed before choosing.

The user must choose something

Use `CheckBox` for independent yes/no choices, `OptionButton` for one choice in a group, `ComboBox` for a compact list, and `ListBox` when several choices need to stay visible.

The user must start an action

Use `CommandButton`, `ToolBar`, `Menu`, or `ContextMenu`. Give the action a clear Caption and connect the correct event.

The user must understand state

Use `Label` for a message, `Badge` for a short status, `ProgressBar` for work in progress, `Gauge` for a measured value, and `StatusBar` for information that should stay at the bottom of a window.

The user must understand structure

Use containers to show groups. Use `DataGrid` for rows and columns. Use `TreeView` for parent-child relationships.

Common properties

Most visual components use some of these properties:

Need	Common properties
Position and size	`Left`, `Top`, `Width`, `Height`
Layout	`Anchor`, `Dock`, `Margin`, `Padding`
State	`Visible`, `Enabled`, `Opacity`
Appearance	`Font`, `ForeColor`, `BackColor`, borders
Keyboard	`TabIndex`, `TabStop`
Drag and drop	`AllowDrop` (files and text from outside the app); `DragMode` (0 Manual, 1 Automatic) and `DragIcon` (a project image) for dragging the control inside the app
Accessibility	`AccessibleName`, `AccessibleDescription`, `AccessibleRole`, `AutomationId`
Control arrays	`Index` (0 to 32767, set in the Properties **Index** row; absent outside an array, and –1 when read in code). Every visual component has it; non-visual components and plugin controls do not.

Not every component has every property. A property shown in the IDE is not necessarily writable at runtime. Check its description before writing code.

Event families at a glance

Most interactive visual controls can expose pointer, focus, keyboard, and drag/drop events. The most useful specialized groups are:

Component	Events worth learning first
`TextBox`	`Change`, `GotFocus`, `LostFocus`, `KeyDown`, `KeyPress`, `KeyUp`
`ComboBox`	`Change`, `DropDown`, `DropDownClosed`
`Slider`	`Change`, `DragStarted`, `DragCompleted`
`DatePicker`	`Change`, `CalendarOpened`, `CalendarClosed`
`ListBox` / `TabControl`	`SelectionChanged`
`TreeView`	`BeforeSelect`, `AfterSelect`, `BeforeExpand`, `AfterExpand`, `NodeClick`, `NodeDoubleClick`
`DataGrid`	`SelectionChanged`, cell events, row events, `Sorting`, `Sorted`, `DataError`
`Expander`	`Expanded`, `Collapsed`
`SplitContainer`	`SplitterMoved`
`AudioPlayer` / `VideoPlayer`	`Loaded`, `Completed`, `Failed`, `StateChanged`
`Animation`	`FrameChanged`, `Completed`, `Failed`
`Label` / `Image` / `Panel`	`Click`, `DoubleClick`, `MouseDown`, `MouseUp`

Every visual control also has the `Drag` method (`ctl.Drag`, `ctl.Drag vbEndDrag`, `ctl.Drag 0`) and the events `DragDrop(Source, X, Y)` and `DragOver(Source, X, Y, State)`, raised on the control under the pointer while another control is dragged. A control also has a `DragOver` for files and text dragged from outside the app; the parameter list of your handler selects which one it handles. See "Drag a control inside your app" in chapter 6.

`Click` on a `Label`, an `Image` or a `Panel` runs when the user presses and releases the left mouse button inside the control: a press that ends outside it, or a right-button click, does not raise it. These three controls have no default event, so pick `Click` from the event list in the code editor. A VB6 `PictureBox`, which the importer turns into a `Panel`, keeps its `_Click` procedure working.

A project written for McBasic™ before Build 860 may already have a Sub named like a click handler, such as `lblTitle_Click`, `imgLogo_Click` or `pnlTop_Click`, for a control with that name. Without parameters, it now runs when the user clicks the control. With parameters, or as a `Function`, Check reports MCB2201 and names the signature `Click` needs: remove the parameters to make it the click handler, or rename the procedure if it was never meant to handle a click.

The IDE creates the canonical signature for an event. Use that generated procedure as the starting point, especially for events with `ByRef Cancel`, `Handled`, row/column indexes, `TreeNode`, `DragData`, or modifier-key values.

Methods and members in code

Every component with a **TabStop** property has the `SetFocus` method: `TextBox`, `PasswordBox`, `MaskedTextBox`, `SearchBox`, `NumericBox`, `DatePicker`, `TimePicker`, `DTPicker`, `MonthView`, `UpDown`, `CommandButton`, `CheckBox`, `OptionButton`, `ToggleSwitch`, `ComboBox`, `ImageCombo`, `ListBox`, `ListView`, `TreeView`, `DataGrid`, `ColorPicker`, `Slider`, `FlatScrollBar`, `TabControl`, `TabStrip` and `ToolBar`. `TextBox` and `MaskedTextBox` also have `SelectAll`, and `SearchBox` has `Clear`. Methods take no arguments: write `cmdSave.SetFocus`, or `Call cmdSave.SetFocus()`.

A property can have a different type in code than in the Inspector. The one-item-per-line **Items** of a `ListBox`, `ComboBox`, `ImageCombo` or `ListView` (and **Images** of an `ImageList`) is the starting list; in code `Items` is a read-only `Collection` that you change with `Clear` and `Add`. A `TreeView` has `Nodes` in code instead of **Items**. Completion after the control's name, and the AI assistant's component lookup, list exactly the members code can use, with their types.

Some members exist only in code and have no Inspector row: a `TextBox` has `Locked` (the same as `ReadOnly`) and the selection members `SelStart`, `SelLength` and `SelText`, and a `ComboBox` or `ListBox` has `ListIndex` (the same as `SelectedIndex`) and `ItemData`, which takes an index like an array (`lstNames.ItemData(0)`). See "Classic properties added in Build 817" above.

`AudioPlayer` and `VideoPlayer` share `Source`, `AutoPlay`, `Loop`, `Volume`, `Position`, `Duration`, `Playing`, `Paused`, `State`, and `Error`, with `Load`, `Play`, `Pause`, and `Stop` methods. Video also has `ControlsVisible` and `Stretch`. The Designer preview is inert; the running app is where playback events and transport behavior are tested.

`Animation` uses an `ImageList`, `Frames`, `Interval`, `AutoPlay`, `Loop`, `Stretch`, and `CurrentFrame`. `Line` and `Shape` are visual drawing controls, not decoration that disappears at runtime. Their geometry and style properties can be authored in the form and changed by BASIC where the Inspector exposes them.

Naming examples

Use names that tell you the job:

```
txtCustomerName
cmdSave
lblStatus
grdCustomers
tvwCustomers
tmrRefresh
```

Good names make event code easier to read and make Find References more useful.

Tips and tricks

- **Create Control Array… works with most controls.** Select one control, choose **Design > Create Control Array…**, and the events of the new array receive `Index`.
- **Add a picture to a CommandButton.** Set **Picture** to an image resource and choose a side with **PictureAlignment**. The AI cannot create image files, so add them to the project yourself.
- **Keep a game's pictures in an ImageList.** Put the six faces of a die in one `ImageList` and show a face with `imgDice.Picture = ImageList1.ListImages(face).Picture`.
- **Draw a board behind controls with Panel.Picture.** The picture stays at its natural size, so make the image as large as the panel.
- **Look up the default event first.** The table shows which event each component raises first, and that is the one you usually need.
- **Colors take names too.** Every color property accepts a system color name such as `WindowText` as well as `#RRGGBB`. Reading it back gives the name.
- **Read the property table before asking the AI.** Naming the exact property in a request gives a better answer.

Appendix 3 — McBasic™ language quick reference

This is a quick map of the language for beginners. McBasic™ does not accept every statement from every BASIC dialect. If a statement is outside the supported language, Check should tell you. Check the language specification for the release before you build a feature around a new statement.

A form procedure

```
Private Sub cmdGreet_Click()
    Dim name As String
    name = Trim$(txtName.Text)
    lblGreeting.Caption = "Hello, " & name
End Sub
```

The procedure has a visibility keyword, a kind (`Sub`), a name, parentheses, and an ending `End Sub`.

Common declarations

Example	Meaning
`Option Explicit`	Require names to be declared.
`Dim name As String`	Create a local text variable.
`Private saveCount As Long`	Keep a private form or module value.
`Public Function CleanName(...) As String`	Define a function other code can call.
`Const MaxRows As Long = 1000`	Give a fixed value a name.

Use the smallest scope that is enough. A value needed only inside one procedure should not become global state.

Check enforces these declaration rules, and its messages name the fix:

- Every variable has its own `As` clause. In `Dim a, b As Long` only `b` is a `Long`; `a` is a `Variant`.

- Declare first, then assign on its own line: `Dim total As Long`, then `total = 5`. `Dim`, `Private`, `Public` and `Static` take no initialiser.
- Create objects with `Set` and `New`: `Dim items As Collection`, then `Set items = New Collection`. `As New` is not available.
- Write type names, not suffixes: `Dim name As String`, never `Dim name$` or `count%`. A fixed-length `String * 10` is not available; shorten text with `Left(text, 10)`.
- `Const` takes literals, other constants and operators, never a function call. In a form module, `Const` and `Enum` must be `Private`.
- `Static` declares a variable inside a procedure. `Static Sub`, `Static Function` and `Static` arrays are not available.

Common types

The everyday types are:

- `Boolean` for true or false;
- `String` for text;
- `Integer`, `Long`, and other numeric types for numbers within their defined

ranges;

- `Date` for a date or time value;
- `Variant` when the supported API needs a value that can hold different data;
- object references such as `Control`, `Database`, `Recordset`, or a Form type;
- user-defined types (`Type ... End Type`) for small records of fields.

Do not choose a type only because another language uses the same name. McBasic™ defines its own widths, conversions, and error rules.

User-defined types (Type … End Type)

A `Type` groups several fields into one value, such as a point, a row of a table or the settings of one game level. Declare it at module level, outside every procedure:

```
Public Type Player
    Name As String
    Score As Long
    Moves(1 To 3) As String
    Home As Point
End Type

Public Type Point
    X As Long
    Y As Long
End Type
```

Each field line is `Name As Type`, with optional fixed bounds such as `Moves(1 To 3)`; a field may be another `Type`, an array (fixed or dynamic) or any other type. Blank lines and comments may sit between fields, and the block ends with `End Type`.

Where a `Type` can be declared and used:

- In a standard module (`.bas`), a `Type` is `Public` unless you write `Private Type`. A `Public Type` is visible to every module, form and class of the project; a `Private Type` only in its own module, so two modules may each declare a `Private Type` with the same name.
- A form or class may declare only `Private Type`. To share a Type, move it to a standard module.
- A `Public Type` cannot share its name with a form, class, module or another `Public Type`.

A `Type` is a value, not an object, so it follows these rules:

```
Dim a As Player
Dim b As Player
a.Name = "Ada"
a.Home.X = 3
b = a              ' copies every field, nested Types and arrays too
b.Name = "Bob"     ' a.Name is still "Ada"

Dim team(3) As Player
team(1).Score = 10 ' every element is ready; no New is needed
With team(2)
    .Name = "Cy"   ' writes into team(2) itself
End With
```

- `Dim p As Player` is ready to use: a `Type` value is never `Nothing`.
- `b = a` copies every field. Changing `b` afterwards never changes `a`.
- The elements of an array of a `Type` are live values, after `Dim`, after `ReDim` and after `ReDim Preserve` (which keeps the old elements). `Erase` resets every element of a fixed array.
- A `ByRef` parameter (the default) is the caller's variable itself, so the procedure's changes are visible to the caller. A `ByVal` parameter is a copy.
- A `Function` that returns a `Type` assigns its fields through the function name (`MakePlayer.Name = "Ada"`); the caller receives a copy.
- `With p` and `With team(i)` write through to the variable or element.
- A field can be passed to a `ByRef` parameter of its own type (`Bump team(1).Score`).

Check refuses, and each message names the fix:

- `Set`, `New`, `Nothing` and `Is` with a `Type`: assign with `=`.

- = or <> between two `Type` values: compare the fields one by one.
- A `Type` value in a `Variant`, a `Collection` or a `Dictionary`: store the fields, or use a class.
- `Len(p)` of a `Type`: its byte size is not available; compute the size you need from the fields.
- `For Each` over an array of a `Type`: loop with `For i = LBound(a) To UBound(a)`.
- A `Type` that contains itself, directly or through another `Type`: use a dynamic array (`Children() As Node`) or a class for the recursive part.
- A fixed-length `String * n` field: declare it `As String`.

Operators used often

Operator	Use
=	Assign a value or compare values, depending on the statement.
&	Join text.
+, -, *, /	Numeric arithmetic.
=, <>, <, >, <=, >=	Compare values.
And, Or, Not	Combine or change logical values.
Is	Compare object identity.
Set	Assign an object reference.

`Rnd` returns a random `Double` from 0 to just below 1. Like `Now`, `Date` and `Time`, it takes no arguments and may be written without parentheses, as in Visual Basic 6: `face = Int((6 * Rnd) + 1)` gives a whole number from 1 to 6.

Decisions and loops

Use `If ... Then ... Else ... End If` for a decision. Use `For ... Next`, `For Each ... Next`, `Do While`, or `Do Until` when a supported loop matches the task. Make the end condition easy to see.

The counter of a `For` loop is a numeric variable (`Dim i As Long`) or a `Variant`, as in VB6 (`Dim i` without `As` is a `Variant`):

```
Dim v As Variant
For v = 1 To 3
    Debug.Print v
Next
```

The start, the limit and the step are read once, when the loop starts. A `Variant` counter counts in the widest type of the three (a `Double` when one of them is a `Variant`), and a value that is not a number raises run-time error 13. An `Enum` variable cannot count: declare the counter `As Long`.

Use `Exit Sub`, `Exit Function`, `Exit For`, or `Exit While` only when the early exit makes the code clearer. If an exit is hard to understand, split the code into a smaller procedure.

End

`End` on its own stops the whole program at once, as in VB6:

```
If MsgBox("Quit without saving?", vbYesNo) = vbYes Then End
```

Nothing else runs after it. No form gets `Form_QueryUnload` or `Form_Unload`, no class gets `Class_Terminate`, and the program exits with code 0. `End` can stand alone, after `Then` or `Else` on a one-line `If`, or before or after a `:`. It never conflicts with `End If`, `End Sub` and the other block ends.

`End` works anywhere in a running app, also in the startup form's `Form_Initialize` or `Form_Load`, and in a `Class_Initialize` that runs while that form is being built, for example `If Not LoadSettings() Then End`.

`End` inside a test, or in app code that a test calls, does not stop the test run (**Test** or `mcbasic test`): it raises run-time error 5, and that test is reported as failed.

To close the program normally, so that every form can ask to save its work, use `Unload Me` in the last open form, or `App.Quit`. Keep `End` for the case where nothing needs saving.

Strings

Common string operations include `Len`, `Trim$`, `CStr`, and concatenation with `&`. Decide whether spaces are meaningful before trimming them. Keep in mind that the current language contract defines how string length and indexing work.

Dates

Use `Now` to capture the local machine date and time. Use `CDate` and `CDbl` only with the conversions supported by the current contract. Use a documented format mask rather than assuming that every date-formatting pattern is accepted.

To save a date as text (a file, JSON, or `Settings`), write ISO text and read it back with `CDate`:

```
stamp = Format(day, "yyyy-MM-dd") & "T" & Format(day, "HH:mm:ss")
day = CDate(stamp)
```

`CDate` accepts only `yyyy-MM-dd` and `yyyy-MM-ddTHH:mm:ss` text, so leave out the time part for a date only. `CStr(day)` is for display, not for saving. In SQLite, use a `DATE` column and pass the `Date` value as a parameter.

Format masks

`Format(value, mask)` accepts a fixed set of masks. Check rejects any other literal mask, and a mask held in a variable raises error 5 when it runs. The value must be a declared number or `Date`; convert a `Variant` or text first with `CDbl(...)` or `CDate(...)`.

These are the only Date masks; letter case is ignored. Each line shows what `Format(d, mask)` returns for 2 October 2026 at 14:05:09:

- `yyyy-MM-dd` gives `2026-10-02`;
- `dd/MM/yyyy` gives `02/10/2026`;
- `MM/dd/yyyy` gives `10/02/2026`;
- `HH:mm:ss` gives `14:05:09`;
- `yyyy-MM-dd HH:mm:ss` gives `2026-10-02 14:05:09`;
- `dd/MM/yyyy HH:mm:ss` gives `02/10/2026 14:05:09`;
- `MM/dd/yyyy HH:mm:ss` gives `10/02/2026 14:05:09`.

Build any other layout from parts: `MonthName(Month(d)) & " " & CStr(Day(d)) & ", " & CStr(Year(d))`, or `Format(Day(d), "00")` for a two-digit day. Named formats such as `Short Date` and VB6 letters such as `mmm` or `AM/PM` are not available.

A number mask uses only these tokens, at most 64 characters:

- `0` is a digit, shown even when it is zero;
- `#` is a digit, left out when it is not significant;
- `,` groups thousands, only between two digit placeholders before the decimal point;
- `.` is the decimal point, followed by at least one digit placeholder;
- `%` is allowed only as the last character; it multiplies the value by 100.

For 1234.5, `"0"` gives `1235`, `"0.00"` gives `1234.50`, `"#,##0"` gives `1,235`, `"#,##0.00"` gives `1,234.50` and `"0%"` gives `123450%`. Currency symbols, text, sections and named formats such as `Currency` are not available: add them with `&`, as in `"$" & Format(price, "#,##0.00")`. `FormatNumber`, `FormatCurrency`, `FormatPercent` and `FormatDateTime` do not exist; use `Format` with one of these masks.

Conversions

Assignments, `ByVal` arguments and function results convert silently between numeric types, from numbers, `Boolean` and `Date` to `String`, and between any value and `Variant`. A number that does not fit its destination raises error 6.

Two conversions are always explicit:

- Text to a number, `Boolean` or `Date`: write `CLng(txtCount.Text)`, `CInt(...)`, `CDbl(...)`, `CLngLng(...)`, `CBool(...)` or `CDate(...)`. Number text uses digits, a sign, a decimal point and an exponent; anything else raises error 13. `Val(text)` returns a `Double`.
- `LongLong` to a smaller type: write `CLng(total)`; a value out of range raises error 6.

A `Variant` converts to every type, including `LongLong`, with the same errors. Recordset values and `ScalarAsync` results arrive as `Variant`. Test `IsNull` first, because converting `Null` raises error 94.

Functions that take numbers, such as `DateSerial`, `TimeSerial`, `DateAdd` and `Round`, need numeric values. Convert text first:

```
day = DateSerial(CLng(Mid(s, 1, 4)), CLng(Mid(s, 6, 2)), CLng(Mid(s, 9, 2)))
```

`CVar` and `DateValue` do not exist. Check reports any other conversion that is not allowed, and its message names the function to use.

Object references

Use `Set` when assigning an object reference:

```
Dim details As frmDetails
Set details = New frmDetails
details.Show
```

Use `Nothing` for an empty reference and `Is` to compare two references. A reference to a closed object may no longer be usable; do not keep it and assume that it points to a new object later.

`Load` creates a form and runs its `Form_Load` without showing it. It takes the form's name, `Me`, or a form variable:

```
Load frmOptions          ' Form_Load runs; the window stays hidden
frmOptions.Show          ' now it appears
```

A second `Load` of a loaded form does nothing. After `Unload`, a new `Load` creates a new instance. `Load txtCell(i)` on a control array is described under "Control arrays and the Controls collection".

Every visual control type is also a variable type, and so is `Control`:

```
Dim t As TextBox
Dim c As Control
Set t = txtName          ' a TextBox
Set c = t                ' any control fits in Control
If TypeOf c Is TextBox Then Set t = c   ' checked when the code runs
(error 13 otherwise)
Dim cells(80) As TextBox ' arrays of controls
Dim items() As Customer  ' arrays of class instances
```

A control type works for a parameter and a Function result too, as in `Function FirstEmpty() As TextBox`. `New TextBox` is not available: controls are created in the designer. Arrays of Forms are not available. A Collection or a Variant cannot hold a control. An array of class instances takes `ReDim`, can be passed to a procedure and returned by a Function, and `For Each` over it needs a loop variable of the class type (`Dim person As Customer`).

`Form` is a type too, in any project that has forms. A parameter or variable declared `As Form` takes any form of the project, so one helper can work on every window:

```
' In a standard module
Public Sub PlaceTopLeft(ByVal f As Form)
    f.Left = 40
    f.Top = 40
End Sub

' In any form
Private Sub Form_Load()
    PlaceTopLeft Me
End Sub
```

An `As Form` reference reaches the members every form has: `Caption`, `Left`, `Top`, `Width`, `Height`, `Visible`, `Enabled`, `WindowState`, `Tag`, `Show`, `Hide`, `SetFocus` and `Unload f`. To reach a form's controls or its `Public` procedures, declare the form's own type (`ByVal f As frmMain`): Check names that fix when code uses any other member through `As Form`. Pass `Me` to a `ByVal f As Form` parameter; a `ByRef` one needs a variable declared `As Form`. A console program or a library has no forms, so Check refuses `As Form` there; so does a project whose only forms are UserControls.

A UserControl is not a form: it sits inside a form and has no window of its own. Check therefore refuses `SetFocus` on a UserControl (MCB2101) and says to call `SetFocus` on a control inside it, or on the form that holds it. A UserControl cannot be held `As Form` either, so `Me` inside a UserControl is not a valid `ByVal f As Form` argument: declare the variable or parameter as the UserControl's own type.

Control item lists

A list control's items are an object in code, not text. `Items` of a `ListBox`, `ComboBox`, `ImageCombo` or `ListView` and `Images` of an `ImageList` are read-only `Collection` references; `Nodes` of a `TreeView` is a `TreeNodes`. Change the items with their members and never assign the list itself:

```
lvwTasks.Items.Clear
lvwTasks.Items.Add "Report|doc|12 KB"
lstNames.Items.Add "Ada"
Dim node As TreeNode
Set node = tvwFolders.Nodes.Add("Inbox")
```

The one-item-per-line **Items** text in the Inspector is only the starting list the form opens with. `SelectedItems` and `SelectedIndices` are read-only snapshots of the selection: read them with `Count` and `Item(index)` and change the selection with `SelectedIndex`. The `Items` of a `ToolBar`, `TabStrip`, `Menu` and `ContextMenu` stay one-item-per-line `String` text in code too.

Some control members take an index, like an array: `lstNames.ItemData(0) = 42`. `ItemData` of a `ListBox` or `ComboBox` keeps one Long with each item, 0 for the first; Check reports an error when it is used without exactly one index.

The images of an `ImageList` are also reachable the Visual Basic 6 way: `ImageList1.ListImages(i).Picture` (or `ListImages.Item(i).Picture`) is the resource name of an image, by one-based index or by Key, and `ImageList1.ListImages.Count` counts them. Assign it to the `Picture` of an Image, PictureBox, CommandButton or Form. An index out of range raises error 9.

Control arrays and the Controls collection

Write	Meaning
`txtCell(i)`, `txtCell.Item(i)`	the member with Index `i` (error 340 when there is none)
`txtCell.Count`, `txtCell.LBound`, `txtCell.UBound`	the number of members, the lowest and the highest Index
`For Each t In txtCell`	the members in Index order (`Dim t As TextBox` or `As Control`)
`Sub txtCell_Click(Index As Integer)`	the shared handler; `Index` comes first
`Me.Controls.Count`, `For Each c In Me.Controls`	every visual control of the form in layout order (`Dim c As Control`)
`Me.Controls("txtName")`, `Me.Controls("txtCell(5)")`, `Me.Controls("txtCell", 5)`	one control by name (error 730 when there is none; error 5 for an array name alone)
`Load txtCell(i)`	adds a member with Index `i`: a hidden copy of the lowest member, in its container; set `Visible = True` to show it (error 360 when `i` exists)
`Unload txtCell(i)`, `Unload t`, `Unload Me.Controls("txtCell(i)")`	removes a member that `Load` created (error 362 for a designer member, 365

Write	Meaning
	inside its own event, 340 when `i` does not exist)

A `Control` has `Name`, `Parent`, `Index` (–1 outside an array), `Enabled`, `Visible`, `Left`, `Top`, `Width`, `Height`, the minimum and maximum sizes, `ActualWidth`, `ActualHeight`, `Tag`, `TabIndex`, `Font`, `ForeColor` and `BackColor`. For the other members, declare the variable with the control's own type.

`Load` works only in the code of the array's own form. Errors 361 (can't load or unload this object) and 364 (object was unloaded) complete the set; Chapter 6 lists every message and its fix. Check refuses `Load` of a non-array control, a bare array name, a control variable, a control of another form and a literal Index outside 0 to 32767, and arrays of containers or TabPages, docked templates and templates in a SplitContainer. MCB2404 warns about a literal Index that always fails. `For Each` over such an array, or over `Me.Controls`, visits the controls present when the loop starts.

Menus and PopupMenu

Menu items are named, non-visual objects of a form, such as `mnuFile` or `mnuFileOpen`. You create them with the Menu Editor (**Design › Menu Editor…**, Command–E; Chapter 8).

Write	Meaning
`mnuFileSave.Caption`, `.Checked`, `.Enabled`, `.Visible`	read and write; the menu updates at once
`mnuFileSave.Shortcut`, `.WindowList`, `.Index`, `.Name`	read only
`Sub mnuFileOpen_Click()`	runs when the user chooses the item or presses its shortcut; a disabled or hidden item does not run
`Sub mnuRecent_Click(Index As Integer)`	the shared handler of a menu control array; `Index` comes first
`Load mnuRecent(i)`, `Unload mnuRecent(i)`, `For Each`	work on a menu control array as on a control array
`PopupMenu mnuEdit`	shows the items of the menu title `mnuEdit` as a pop-up menu at the mouse pointer
`PopupMenu mnuEdit, vbPopupMenuRightAlign, 10, 20`	the same at `x`, `y` on the form (DIP), aligned by the flags

Write	Meaning
`PopupMenu mnuEdit, , , , mnuEditCopy`	`mnuEditCopy`, an item of that menu, is drawn in bold
`Me.PopupMenu mnuEdit`	the same statement, written with `Me`

The full form is `PopupMenu menu [, flags [, x, y [, defaultMenu]]]`. Give `x` and `y` together or not at all. `PopupMenu` is a statement; it returns at once, and the chosen item's `Click` runs afterwards. `menu` must be a menu title (an item without a parent) that has items, and must belong to the form whose code calls `PopupMenu`. If every item of the title is hidden, nothing appears and no error is raised.

`flags` takes `vbPopupMenuLeftAlign` (0), `vbPopupMenuCenterAlign` (4) or `vbPopupMenuRightAlign` (8). The macOS menu honours the alignment where AppKit allows. `vbPopupMenuLeftButton` (0) and `vbPopupMenuRightButton` (2) are accepted and ignored on the Mac.

Menu items follow the rules of a form's own layout: `Unload` refuses an item placed in the layout (MCB2101); hide it with `Visible = False`, or make it a menu control array and unload only the members you loaded. Run-time errors name the fix: error 91 when the menu is `Nothing`, error 13 when it is not a menu item and error 5 for an item of another form, an item that is not a menu title, or a title with no items.

Asynchronous work

Database and network operations may take time. Use the supported `Async` and `Await` forms for those APIs, disable duplicate actions, and handle both success and failure. Do not make the window wait silently while a long task runs.

An asynchronous procedure is still an ordinary BASIC procedure: it has a clear return type, declared parameters, and a visible error path. The useful pattern is:

```
Private Async Function RefreshAsync() As Boolean
    Try
        Await RefreshDatabaseAsync()
        RefreshAsync = True
    Catch failure As McException
        lblStatus.Caption = failure.Message
        RefreshAsync = False
    End Try
End Function
```

Do not start a second request while the first one is active. Disable the button or use a form-level busy flag, and restore the control in both success and failure paths.

The rules Check enforces for asynchronous code:

- `Await` is allowed only inside an `Async Function` or an `Async Sub`. It is a statement of its own (`Await SaveAsync()`) or the whole right side of an assignment (`count = Await CountAsync()`, `Set db = Await Sqlite.OpenAsync(path)`). Move anything else into a local variable first.
- Every call to an `Async Function`, and every member whose name ends in `Async`, needs `Await` with parentheses and positional arguments. Code that is not `Async` cannot call them.
- Write `ByVal` on every parameter of an `Async Function`. Parameters are `ByRef` unless you write `ByVal`, and `ByRef`, `Optional`, `ParamArray` and array parameters are not allowed.
- An `Async Function` returns a value type (`Boolean`, the numeric types, `String`, `Date`, `Variant`) or a built-in object such as `Database`, `Recordset`, `DbCommand`, `Collection` or `Dictionary`. It cannot return an array, an `Enum`, a Form or a Class.
- `Async Sub` is only for event handlers without parameters on a form: control events such as `cmdLoad_Click`, and `Form_Shown`, `Form_Activate`, `Form_Deactivate`, `Form_Resize` and `Form_Closed`. `Form_Load`, `Form_Initialize`, `Form_Closing`, `Form_QueryUnload` and `Form_Unload` cannot be `Async`.
- Handle errors with `Try`, `Catch` and `Finally`. `On Error`, `Resume`, `GoTo` and labels are not allowed in an `Async` procedure.

Procedure signatures

The type beside a parameter tells the reader what the procedure expects. Use `ByVal` for a value the procedure only reads. Use `ByRef` when the procedure is deliberately allowed to change the caller's variable, as with a cancellable event:

```
Private Sub Form_Closing(ByRef Cancel As Boolean)
    If mBusy Then Cancel = True
End Sub
```

The function name receives the return value:

```
Private Function IsUsable(ByVal value As String) As Boolean
    IsUsable = Len(Trim$(value)) > 0
End Function
```

Keep parameter lists short. When the last parameters have sensible defaults, declare them `Optional`. When a procedure accepts any number of values, end its parameter list with a `ParamArray`:

```
Private Sub LogLine(ByVal text As String, Optional ByVal level As Long =
1, _
        Optional ByVal source As Variant)
    If IsMissing(source) Then
        Debug.Print level & " " & text
```

```
    Else
        Debug.Print level & " " & text & " [" & source & "]"
    End If
End Sub

Private Function CountFilled(ParamArray values() As Variant) As Long
    Dim item As Variant
    Dim filled As Long
    For Each item In values
        If Len(Trim$(CStr(item))) > 0 Then filled = filled + 1
    Next item
    CountFilled = filled
End Function
```

A caller can leave out optional arguments, skip one by leaving its position empty, or name the parameters it passes:

```
LogLine "Saved"                                ' 1 Saved
LogLine "Saved", 2                             ' 2 Saved
LogLine "Saved", , "import"                    ' 1 Saved [import]
LogLine source:="import", text:="Saved"        ' 1 Saved [import]
Debug.Print CountFilled("Ada", "", "Grace")    ' 2
```

The rules Check enforces:

- `Optional` parameters come after every required parameter. A typed `Optional` parameter needs a constant default, such as `= 1`, `= ""` or `= True`. Only an `Optional` parameter `As Variant` can leave the default out; inside the procedure, `IsMissing(name)` is `True` when the caller did not pass it.
- `Optional` is not available for array, Form or Control parameters.
- To skip an argument in the middle, leave its position empty. An argument list cannot end with a comma: stop after the last argument you pass.
- A named argument is written `parameter:=value`. The names are the ones in the declaration, in any order and any letter case. After the first named argument, the rest of the call must be named too.
- A `ParamArray` is the last parameter, declared as `ParamArray name() As Variant` without `ByVal` or `ByRef`. Its array always starts at 0, even with `Option Base 1`, and is empty (`UBound` is -1) when the caller passes no values. Changing an element changes the caller's variable when the caller passed a variable.
- A procedure with a `ParamArray` takes positional arguments only, and it cannot also declare `Optional` parameters.
- A statement can call a `Function` of a module, form or class and ignore its result, as in Visual Basic 6: `Candidates board, i, list` or `Call Candidates(board, i, list)` runs the function once and discards the value it returns. An `Async Function` still needs `Await` (`Await SaveAsync()`), and a `Property Get` is read in an expression.
- An `Async Function`, an `Async Sub` and an `Event` declaration cannot have `Optional` or `ParamArray` parameters, and a call with `Await` takes positional arguments only (see "Asynchronous work").

- Use named arguments with your own procedures. Most built-in functions, such as `Left`, `Mid`, `Replace` and `MsgBox`, take positional arguments only, so pass their arguments by position.

JSON values

`JsonValue` is McBasic™'s structured value for a JSON object, array, string, number, Boolean, or null. Assign it with =. Use `Json.Parse` to read text, then move through the value with the small set of operations below:

Operation	What it does
`Json.Parse(text)`	Read JSON text and return a `JsonValue`.
`Json.Get(value, "name")`	Read a named property from an object.
`Json.Has(value, "name")`	Check whether a property is present.
`Json.At(value, index)`	Read one item from a zero-based array.
`Json.Count(value)`	Count properties or array items.
`Json.Kind(value)`	Return `Object`, `Array`, `String`, `Number`, `Boolean`, or `Null`.
`Json.Value(value)`	Convert a JSON scalar to a regular McBasic™ value.
`Json.Stringify(value)`	Turn a `JsonValue` back into JSON text.

For example, this is the shape used by WeatherDesk after an Open-Meteo geocoding or forecast request. The first response is an HTTP envelope; its `body` property contains the API's JSON text. The second parse reads that text as an object.

```
Dim envelope As JsonValue
Dim document As JsonValue
Dim books As JsonValue
Dim firstBook As JsonValue
Dim title As String

envelope = Json.Parse(responseText)
document = Json.Parse(CStr(Json.Value(Json.Get(envelope, "body"))))
books = Json.Get(document, "docs")

If Json.Count(books) > 0 Then
    firstBook = Json.At(books, 0)
    title = CStr(Json.Value(Json.Get(firstBook, "title")))
End If
```

Check a property with `Json.Has` before reading an optional field. Treat a missing property and a property whose value is JSON `null` as different cases; the user may need a different message for each one.

HTTP requests

`Http.RequestAsync` sends an HTTP or HTTPS request and returns a JSON response envelope as text. Its arguments are the method, URL, headers JSON, JSON body, and timeout in milliseconds:

```
Dim responseText As String
Dim headersJson As String

headersJson = "{""User-Agent"":""McBasic WeatherDesk""}"
responseText = Await Http.RequestAsync( _
    "GET", _
    "https://geocoding-api.open-
meteo.com/v1/search?name=Singapore&count=1", _
    headersJson, _
    "", _
    10000)
```

The call belongs in an `Async` procedure and must be awaited. Disable the button while it runs so a double-click does not start two searches. Handle network errors, invalid JSON, empty results, and HTTP errors separately when the user needs different recovery advice. Never place a password or private token directly in a public sample.

Common app services

The current samples demonstrate bounded surfaces for:

- `App` and `AppPaths`;
- SQLite through `Sqlite`, `Database`, `DbCommand`, `Recordset`, and related

types;

- HTTP download and cancellation;
- JSON and file operations;
- settings and logging;
- reports and PDF export;
- form and control references.

Each service has limits. Use the matching sample as a working starting point, then read its release documentation before extending it. `Try ... Catch` is the practical failure-handling form used by the current database and network examples.

App

`App` describes the running application. Besides `App.Name`, `App.Version`, `App.Build`, the paths (`App.Path`, `App.DataPath`, `App.ResourcePath`, `App.CachePath`, `App.LogPath`) and `App.Quit`, it has the VB6 members a Mac app can honour:

Member	Type	Meaning
`App.Title`	`String`, read/write	The application's title; it starts as `App.Name`.
`App.EXEName`	`String`, read only	The application's name, as `App.Name`.
`App.PrevInstance`	`Boolean`, read only	Always `False`: macOS runs one copy of an app and brings it to the front.
`App.Major`, `App.Minor`, `App.Revision`	`Integer`, read only	The parts of the version `a.b.c`; a missing part is 0.
`App.ProductName`	`String`, read only	The product name, the application's name.
`App.CompanyName`	`String`, read only	The **Company** of the project (**Project > Project Settings…**); empty text when the project has none.
`App.LegalCopyright`	`String`, read only	The **Copyright** of the project (**Project > Project Settings…**); empty text when the project has none.
`App.FileDescription`	`String`, read only	Empty text in this version.

`App.Title` is the only `App` member code can assign; Check refuses an assignment to any other one.

Screen

`Screen` describes the main screen and the pointer. It exists in desktop apps, which have forms; in a console program Check reports that there is no screen to measure.

Member	Type	What it gives
`Screen.Width`, `Screen.Height`	`Double`, read only	the main screen's size in points, the unit of every McBasic™ position
`Screen.TwipsPerPixelX`, `Screen.TwipsPerPixelY`	`Long`, read only	always 1, because positions are points, not twips
`Screen.MousePointer`	`Integer`, read and write	the pointer over every window of the app: `vbHourglass`, `vbArrow` and the other constants under Other constants; `vbDefault` gives each control its own pointer back; controls created later, such as a member made by `Load`, get it too

Member	Type	What it gives
`Screen.ActiveForm`	`Form`, read only	the form whose window is active (the active child of an MDIForm), or `Nothing`

```
' Centre the form on the screen
Me.Left = (Screen.Width - Me.Width) / 2
Me.Top = (Screen.Height - Me.Height) / 2
```

Because `TwipsPerPixelX` and `TwipsPerPixelY` are 1, VB6 code that divides by them, such as `Screen.Width / Screen.TwipsPerPixelX`, still gives points.

Form members in code

A form's own properties are members of `Me` inside the form, and members of a form variable or of the form's name everywhere else:

Me.Caption = "Customers" Dim f As frmEdit Set f = New frmEdit f.Caption = "Edit customer" frmEdit.KeyPreview = True

Declare the variable with the form's own type (`Dim f As frmEdit`) to reach every one of these members. A generic `As Form` variable reaches only `Caption`, `Left`, `Top`, `Width`, `Height`, `Visible`, `Enabled`, `WindowState` and `Tag` (see Object references).

Every property in the Form section of the Properties panel is available this way, except the ones the panel marks as set in the designer only (`MDIForm`, `MDIChild`, `MDIClientTop`). Check reports an error that names the Properties panel when code reads or assigns one of those, or assigns a property that is read-only in code. Code also has `Visible`, which the panel does not show: `Me.Visible = False` hides the form, as `Me.Hide` does.

A bare name never means a Form member: write `Me.Left`, because `Left(s, 1)` is always the string function. If a control has the same name as a Form property (a Label named `Tag`, for example), the name means the control in that form.

Built-in constants

McBasic™ has the VB6 constants built in, with their VB6 values. Use them anywhere a value of that type fits, including `Const` expressions:

```
Private Const ConfirmStyle As Long = vbYesNo + vbQuestion +
vbDefaultButton2

Private Sub cmdDelete_Click()
    If MsgBox("Delete this customer?", ConfirmStyle, "Delete") = vbYes
Then
        lblStatus.Caption = "Deleted." & vbCrLf & "Choose Undo to restore
it."
    End If
End Sub
```

A variable, field or constant that you declare with the same name takes precedence over a built-in constant. Assigning to a built-in constant is a Check error: declare your own variable with another name.

The numeric constants are `Long`, as in VB6. The text constants, the colors and the window states are `String`.

MsgBox

Group	Constants and values
Buttons	`vbOKOnly` 0, `vbOKCancel` 1, `vbAbortRetryIgnore` 2, `vbYesNoCancel` 3, `vbYesNo` 4, `vbRetryCancel` 5
Icons	`vbCritical` 16, `vbQuestion` 32, `vbExclamation` 48, `vbInformation` 64
Default button	`vbDefaultButton1` 0, `vbDefaultButton2` 256, `vbDefaultButton3` 512, `vbDefaultButton4` 768
Modality and options	`vbApplicationModal` 0, `vbSystemModal` 4096, `vbMsgBoxHelpButton` 16384, `vbMsgBoxSetForeground` 65536, `vbMsgBoxRight` 524288, `vbMsgBoxRtlReading` 1048576
Results	`vbOK` 1, `vbCancel` 2, `vbAbort` 3, `vbRetry` 4, `vbIgnore` 5, `vbYes` 6, `vbNo` 7

Add one button set, one icon and one default button with `+`. The default button is the one that Return chooses; a number past the last button selects the first one. `vbCritical` shows the critical alert, and the other icons show a matching symbol. The modality and option constants are accepted so that VB6 code compiles, but they have no effect: macOS alerts are always application-modal and the system lays them out.

Text

Constant	Value
`vbCrLf`	`Chr(13) & Chr(10)`, the Windows line break
`vbCr`	`Chr(13)`
`vbLf`	`Chr(10)`
`vbNewLine`	`Chr(10)`, the macOS line break (the same text as `Environment.NewLine`)

Constant	Value
vbTab	Chr(9)
vbNullString	""
vbNullChar	Chr(0)
vbBack	Chr(8)
vbFormFeed	Chr(12)
vbVerticalTab	Chr(11)

Colors

McBasic™ colors are "#RRGGBB" text or a system color name, the same text that a form layout stores for ForeColor, BackColor, BorderColor and the DataGrid's AlternatingRowBackColor. The color constants, RGB and QBColor therefore return String, not VB6's Long:

Constant	Value
vbBlack	"#000000"
vbRed	"#FF0000"
vbGreen	"#00FF00"
vbYellow	"#FFFF00"
vbBlue	"#0000FF"
vbMagenta	"#FF00FF"
vbCyan	"#00FFFF"
vbWhite	"#FFFFFF"

RGB(red, green, blue) takes three numbers from 0 to 255 (a larger number counts as 255) and returns the color text: RGB(18, 52, 86) is "#123456". QBColor(index) returns one of the 16 classic colors for an index from 0 to 15: 0 black, 1 blue, 2 green, 3 cyan, 4 red, 5 magenta, 6 yellow, 7 white, 8 gray, and 9 to 15 the light versions of 1 to 7.

```
Dim warning As String

warning = RGB(255, 128, 0)
lblTotal.ForeColor = warning
lblTitle.BackColor = vbWhite
```

Keep a color in a String variable. Storing RGB(...) or vbRed in a Long, or assigning a number such as &HFF& to a color property, is a Check error.

The 27 VB6 system color constants are available too. Each one holds the name of a system color, and a color property shows that color for the project's appearance: the Windows 10 color in **Classic**, and in **macOS** the same color except dark gray text (`#1C1C1C`) and the macOS blue `Highlight` (`#007AFF`). Reading the property back gives the name, not a `"#RRGGBB"` value.

Constants	Value
`vbScrollBars`, `vbDesktop`, `vbActiveTitleBar`, `vbInactiveTitleBar`, `vbMenuBar`	`"ScrollBars"`, `"Desktop"`, `"ActiveTitleBar"`, `"InactiveTitleBar"`, `"MenuBar"`
`vbWindowBackground`, `vbWindowFrame`, `vbMenuText`, `vbWindowText`, `vbTitleBarText`	`"WindowBackground"`, `"WindowFrame"`, `"MenuText"`, `"WindowText"`, `"TitleBarText"`
`vbActiveBorder`, `vbInactiveBorder`, `vbApplicationWorkspace`, `vbHighlight`, `vbHighlightText`	`"ActiveBorder"`, `"InactiveBorder"`, `"ApplicationWorkspace"`, `"Highlight"`, `"HighlightText"`
`vbButtonFace`, `vbButtonShadow`, `vbGrayText`, `vbButtonText`, `vbInactiveCaptionText`	`"ButtonFace"`, `"ButtonShadow"`, `"GrayText"`, `"ButtonText"`, `"InactiveCaptionText"`
`vb3DHighlight`, `vb3DDKShadow`, `vb3DLight`, `vbInfoText`, `vbInfoBackground`	`"3DHighlight"`, `"3DDKShadow"`, `"3DLight"`, `"InfoText"`, `"InfoBackground"`
`vb3DFace`, `vb3DShadow`	`"ButtonFace"`, `"ButtonShadow"`

```
picBox.BackColor = vbButtonFace      ' reads back as "ButtonFace"
lblHint.ForeColor = "GrayText"       ' the name works as text too, in any
letter case
```

A color text that is neither `"#RRGGBB"`, a system color name nor `""` is a Check error when you assign it as written, such as `lbl.ForeColor = "Red"`, and run-time error MCB4403 when it comes from a variable.

Keys

The key-code constants are members of the `KeyCode` Enum, so they compare directly with the `KeyCode` argument of `KeyDown` and `KeyUp`. Their values are the VB6 values:

Constants	Values
`vbKeyA` … `vbKeyZ`	65 … 90
`vbKey0` … `vbKey9`	48 … 57
`vbKeyNumpad0` … `vbKeyNumpad9`	96 … 105

Constants	Values
vbKeyF1 … vbKeyF16	112 … 127
vbKeyCancel, vbKeyBack, vbKeyTab, vbKeyClear, vbKeyReturn	3, 8, 9, 12, 13
vbKeyPause, vbKeyCapital, vbKeyEscape, vbKeySpace	19, 20, 27, 32
vbKeyPageUp, vbKeyPageDown, vbKeyEnd, vbKeyHome	33, 34, 35, 36
vbKeyLeft, vbKeyUp, vbKeyRight, vbKeyDown	37, 38, 39, 40
vbKeySelect, vbKeyPrint, vbKeyExecute, vbKeySnapshot	41, 42, 43, 44
vbKeyInsert, vbKeyDelete, vbKeyHelp	45, 46, 47
vbKeyMultiply, vbKeyAdd, vbKeySeparator, vbKeySubtract, vbKeyDecimal, vbKeyDivide	106 … 111
vbKeyNumlock, vbKeyScrollLock	144, 145
vbShiftMask, vbCtrlMask, vbAltMask	1, 2, 4

`vbShiftMask`, `vbCtrlMask` and `vbAltMask` are members of the `KeyModifiers` Enum, for the `Shift` argument. Test one with `= vbShiftMask`, because an Enum value compares only with a member of the same Enum:

```
Private Sub txtSearch_KeyDown( _
    ByVal KeyCode As KeyCode, _
    ByVal Shift As KeyModifiers, _
    ByRef Handled As Boolean)
    If KeyCode = vbKeyReturn Then
        Handled = True
        If (Shift And vbShiftMask) = vbShiftMask Then
            lblStatus.Caption = "Search backwards"
        Else
            lblStatus.Caption = "Search"
        End If
    End If
End Sub
```

McBasic™ reports the left and right Shift, Control and Option keys separately, as `KeyCode.LeftShift`, `KeyCode.RightShift`, `KeyCode.LeftCtrl`, `KeyCode.RightCtrl`, `KeyCode.LeftAlt` and `KeyCode.RightAlt`, so VB6's `vbKeyShift`, `vbKeyControl` and `vbKeyMenu` are not available. The Command key is `KeyModifiers.Meta` in the `Shift` argument.

Window states

Constant	Value	Use
`vbNormal`	`"Normal"`	`Me.WindowState = vbNormal`
`vbMinimized`	`"Minimized"`	in the Dock
`vbMaximized`	`"Maximized"`	zoomed

McBasic™ `WindowState` is text, so these constants are `String` (VB6 types them `Integer`).

Other constants

Group	Constants and values
Comparison (`StrComp`, `Filter`)	`vbBinaryCompare` 0, `vbTextCompare` 1
Days (`Weekday`, `WeekdayName`)	`vbSunday` 1, `vbMonday` 2, `vbTuesday` 3, `vbWednesday` 4, `vbThursday` 5, `vbFriday` 6, `vbSaturday` 7
Tristate	`vbTrue` -1, `vbFalse` 0, `vbUseDefault` -2
Errors	`vbObjectError` -2147221504
`VarType` results	`vbEmpty` 0, `vbNull` 1, `vbInteger` 2, `vbLong` 3, `vbSingle` 4, `vbDouble` 5, `vbCurrency` 6, `vbDate` 7, `vbString` 8, `vbError` 10, `vbBoolean` 11, `vbVariant` 12, `vbDecimal` 14, `vbByte` 17, `vbLongLong` 20, `vbArray` 8192
`Drag` method and `DragOver`	`vbBeginDrag` 1, `vbEndDrag` 2; `State` of `DragOver`: `vbEnter` 0, `vbLeave` 1, `vbOver` 2
MDI `Arrange`	`vbCascade` 0, `vbTileHorizontal` 1, `vbTileVertical` 2
`PopupMenu` flags	`vbPopupMenuLeftAlign` 0, `vbPopupMenuCenterAlign` 4, `vbPopupMenuRightAlign` 8, `vbPopupMenuLeftButton` 0, `vbPopupMenuRightButton` 2
`Screen.MousePointer` (`Integer`)	`vbDefault` 0, `vbArrow` 1, `vbCrosshair` 2, `vbIbeam` 3, `vbSizeNESW` 6, `vbSizeNS` 7, `vbSizeNWSE` 8, `vbSizeWE` 9, `vbUpArrow` 10, `vbHourglass` 11, `vbNoDrop` 12,

Group	Constants and values
	`vbArrowHourglass` 13, `vbArrowQuestion` 14, `vbSizeAll` 15

`Weekday(d)` returns one of the day constants. As the first-day argument of `Weekday` and `WeekdayName`, only `vbSunday` and `vbMonday` are accepted. An array's `VarType` adds `vbArray` to its element type, so a `Variant` array reports `vbArray + vbVariant`. For your own component errors, add a number to `vbObjectError`: `Err.Raise vbObjectError + 1, "Store", "The list is full."`. macOS has no diagonal size pointer, so `vbSizeNESW` and `vbSizeNWSE` show the four-way `vbSizeAll` pointer. Any other `MousePointer` number (4, 5 or 99, for example) is run-time error 380.

In a control drag, `vbCancel` is still 2, the `MsgBox` result, as in Visual Basic 6, so `Drag vbCancel` ends the drag with a drop. To cancel a drag, write `Drag 0`.

VB6 features and their McBasic™ replacements

Check rejects these VB6 declarations, statements, functions, constants and types. Each message names the replacement shown here; for a constant, the message starts "VB6 constant ... is not available in McBasic™":

- `Declare`: native library calls are not available; use the McBasic™ services (`File`, `Path`, `Settings`, `Clipboard`, `Platform`, `Http`).
- `Friend`: use `Public` or `Private`.
- `Attribute` lines: remove them; McBasic™ modules do not use them.
- `DefBool`, `DefByte`, `DefInt`, `DefLng`, `DefLngLng`, `DefCur`, `DefSng`, `DefDbl`, `DefDec`, `DefDate`, `DefStr`, `DefObj`, `DefVar`: declare every variable with its own `As` clause.
- `GoSub`: move the shared code into a `Private Sub` and call it. `Return`: leave a procedure with `Exit Sub` or `Exit Function`.
- `Open`, `Close`, `Get`, `Put`, `Input`, `Write`, `FreeFile`, `EOF`, `LOF`: use `File.ReadText`, `File.WriteText` and `File.AppendText`.
- `Print`: use `Debug.Print`, a `Label` or `TextBox`, or `File.AppendText`.
- `CreateObject`, `GetObject`: COM objects are not available; use `New Collection`, `New Dictionary` or a built-in service.
- `DoEvents`: make long work an `Async Function` and `Await` it, or split it with a `Timer`.

- `Shell`: running other programs is not available; `Platform.OpenUrl` opens a web or mailto link.
- `SendKeys`: set control properties or call their methods directly.
- `Beep`: play a sound with an `AudioPlayer` component.
- `Nz`: use `IIf(IsNull(value), fallback, value)`.
- `FormatNumber`, `FormatCurrency`, `FormatPercent`, `FormatDateTime`: use `Format` with a supported mask.
- `Stop`: set a breakpoint in the editor.
- `LSet`, `RSet`: pad with `Left(text & Space(width), width)` or `Right(Space(width) & text, width)`.
- `Environ`: use `Environment.Get(name, defaultValue)`.
- `GetSetting`, `SaveSetting`, `DeleteSetting`: use `Settings.Get(key, defaultValue)` and `Settings.Set key, value`.
- `Kill`, `FileCopy`, `RmDir`, `Dir`, `FileLen`, `FileDateTime`: use the `File` and `Path` services. `MkDir`: `AppPaths.DataFile(relativePath)` creates the folders it needs. `ChDir`, `CurDir`: build full paths with `Path.Combine(App.DataPath, name)`.
- `DateValue`, `TimeValue`, `CVDate`: use `CDate` with ISO text, `DateSerial` or `TimeSerial`. `CVar`: assign the value to a `Variant`.
- `StrConv`: use `UCase` or `LCase`.
- `Error`, `Erl`: read `Err.Number` and `Err.Description` in an error handler.
- `AddressOf`: declare an `Event`, raise it with `RaiseEvent` and handle it with `WithEvents`, or call the procedure directly.
- `vbKeyShift`, `vbKeyControl`, `vbKeyMenu`: `KeyDown` reports the left and right keys separately; compare `KeyCode` with `KeyCode.LeftShift`, `KeyCode.RightShift`, `KeyCode.LeftCtrl`, `KeyCode.RightCtrl`, `KeyCode.LeftAlt` or `KeyCode.RightAlt`, or test `(Shift And vbShiftMask) = vbShiftMask`.
- `vbKeyLButton`, `vbKeyRButton`, `vbKeyMButton`, `vbLeftButton`, `vbRightButton`, `vbMiddleButton`: compare the `Button` argument of `MouseDown` and `MouseUp` with `MouseButton.Left`, `MouseButton.Right` or `MouseButton.Middle`.
- `vbModal`: show a modal form with `Await f.ShowDialogAsync(Me)` inside an `Async Sub`. `vbModeless`: call `f.Show` without an argument. `vbArrangeIcons`: arrange MDI children with `vbCascade`, `vbTileHorizontal` or `vbTileVertical`.
- `vbDatabaseCompare`: use `vbBinaryCompare` or `vbTextCompare`. `vbUseSystemDayOfWeek`: pass `vbSunday` or `vbMonday`. `vbUseSystem`,

`vbFirstJan1`, `vbFirstFourDays`, `vbFirstFullWeek`: `DateDiff` and `DatePart` take no first-week-of-year argument.

- `vbObject`: a `Variant` holds only values, so `VarType` never reports an object; test an object variable with `Is Nothing`.
- `vbReadOnly`, `vbHidden`, `vbSystem`, `vbVolume`, `vbDirectory`, `vbArchive`, `vbAlias`: file attributes are not available; use the `File` and `Path` services.
- `Object`: declare the exact type (a Form or class name, `Collection`, `Dictionary` or `Control`) or `Variant`. `LongPtr`: use `Long` or `LongLong`. `Any`: use `Variant`.

When a statement does not work

1. Read the exact diagnostic. Every Check message names what to write instead.
2. Check the spelling and the type.
3. Confirm that the member belongs to that control or object.
4. Try a smaller example.
5. Check the release compatibility notes.

Do not replace an error with a different language feature until you understand why the first one failed.

Tips and tricks

- **Declare without bounds before you ReDim.** `Dim a() As Long` can be resized by a procedure. An array with bounds, such as `Dim a(10)`, cannot.
- **Use the Controls collection for control arrays.** It lets you loop over the cells.
- **Load a member, then show it.** `Load txtCell(i)` creates a hidden copy; set `Visible = True` after you position it.
- **Write Rnd the short way.** `Int((6 * Rnd) + 1)` and `Int((6 * Rnd()) + 1)` are the same; a variable named `Rnd` in your own code takes precedence.
- **Pop up a menu on a right-click.** Call `PopupMenu mnuEdit` from a control's `MouseUp` handler to show a context menu; the pop-up takes the pointer position when you give no `x`, `y`.
- **Read the VB6 table before you port code.** It shows which old features are replaced and which are not supported.
- **Read the Check message first.** Every Check message names its fix.
- **Count with a Variant only when you port VB6 code.** `For v = 1 To 3` works with `Dim v As Variant`, but `Dim i As Long` is faster and Check catches more mistakes.

- **App.PrevInstance is always False on the Mac.** macOS never starts a second copy of an app, so code that quits when it is True can stay as it is.
- **Use a Type for small records copied by value (a point, a row of a table); use a class when several places must share one object.**
- **Centre with Screen, not with numbers.** `(Screen.Width - Me.Width) / 2` places a form in the middle of any screen.
- **End is for emergencies.** A program that ends with `End` skips every form's `Form_QueryUnload` and `Form_Unload`, so unsaved work is lost.
- **Write a helper that works on any form `As Form`.** When it needs a form's own controls, declare that form's type instead.
- **Use a system color to follow the appearance.** `vbButtonFace` or `vbWindowText` matches the look of the app in both **Classic** and **macOS**; a fixed `"#RRGGBB"` value looks the same in both.

Project files and folder layout

You can use McBasic™ without editing files by hand. It is still worth knowing what a project contains: backups are easier, and you are less likely to share a temporary file by mistake.

The project at a glance

A McBasic™ project normally has a project file, one or more forms, and the code and layout for each form.

```
WeatherDesk.mcbproj/
|-- project.mcproj
`-- Forms/
    |-- frmMain.bas
    `-- frmMain.form.json
```

In Finder `WeatherDesk.mcbproj` appears as one item. Control-click it and choose **Show Package Contents** to see the files above. Older projects that are plain folders (`WeatherDesk/WeatherDesk.mcproj`) still open, and `mcbasic convert <folder>` converts one to a `.mcbproj`.

The names may be different in your project. The relationship is the important part:

- the `project.mcproj` file inside the `.mcbproj` tells McBasic™ which documents belong to the project and how the app starts: from its startup form, or from `Sub Main` in a module (**Project > Startup Object**, see Chapter 9). Its `application` block also holds the app's identity (bundle identifier, version, build number) and the optional `copyright` and `company` texts that **Project > Project Settings…** edits (see Chapter 2). McBasic™ leaves `copyright` and `company` out of the file when they are empty.

The `application` block can also hold the key `appearance`. Its value is `"macOS"` or `"Classic"` and it sets how the controls of the app look (see Chapter 2, "Choose the appearance"). A project without the key looks like macOS, and **Project > Project Settings…** writes the key only when you choose **Classic**, so a macOS project's file does not change. Any other value is a project error: "MCB1001: application.appearance in project.mcproj must be "macOS" or "Classic", not "<value>"; write one of them or remove the key for macOS." If the value is not text at all, the message is "MCB1001: application.appearance in project.mcproj must be the text "macOS" or "Classic"; write one of them or remove the key for macOS." McBasic™ reads the two names without regard to upper or lower case.

The `application` block can also hold the key `icon`: the path, inside the project, of the app icon image, for example `"icon": "AppIcon.png"` (see Chapter 2, "Choose the app icon"). Without the key the app has the McBasic™ icon. **Project > Project Settings…** writes `AppIcon.png`, `AppIcon.icns` or `AppIcon.ico`; you may also point it at an image in a folder of the project, such as `"Art/Logo.png"`. Each wrong value is a project error that names the fix:

Problem	Message
Not text	"MCB1001: application.icon in project.mcproj must be the text path of an image in the project, such as "AppIcon.png"; write it as text or remove the key for the McBasic icon."
Outside the project	"MCB1001: application.icon in project.mcproj must be a path inside the project, such as "AppIcon.png", without ".." or a leading "/"; fix the path or remove the key for the McBasic icon."
Another format	"MCB1001: application.icon in project.mcproj must name a .png, .icns or .ico file, not "<path>"; choose such an image in Project Settings or remove the key for the McBasic icon."
File missing	"MCB1001: the app icon "<path>" named by application.icon is missing from the project; put the image in the project folder, choose it again in Project Settings or remove the key."
Over 8 MiB	"MCB1001: the app icon "<path>" is larger than 8 MiB; use a smaller image (a 1024x1024 PNG is enough)."
Not a PNG	"MCB1001: the app icon "<path>" is not a valid PNG image; export it again as PNG or choose another image."
Not square	"MCB1001: the app icon "<path>" is WxH pixels but must be square; crop it to a square PNG (1024x1024 recommended)."
Under 256 pixels	"MCB1001: the app icon "<path>" is WxH pixels; use a square PNG of at least 256x256 (1024x1024 recommended)."
Over 4096 pixels	"MCB1001: the app icon "<path>" is WxH pixels, larger than 4096x4096; scale it down to 1024x1024."
Interlaced PNG	"MCB1001: the app icon "<path>" is an interlaced PNG; save it again without interlacing."
Damaged ICNS or ICO	"MCB1001: the app icon "<path>" is not a valid .icns file; export it again or choose a square PNG instead." (or `.ico`)

McBasic™ checks the file itself when it checks, builds or publishes the project; while you edit, only the path and the extension are checked, so an icon that you chose but did not save yet is not an error.

- the `.bas` file contains the form's McBasic™ code;

- the `.form.json` file contains the form layout and control properties.

When you add a standard module, McBasic™ can place its code in a `Modules` folder:

```
WeatherDesk.mcbproj/
|-- project.mcproj
|-- Forms/
|   |-- frmMain.bas
|   `-- frmMain.form.json
`-- Modules/
    `-- Helpers.bas
```

A project that keeps its AI chats also has an `AI` folder. McBasic™ manages it; you never declare project files there:

```
WeatherDesk.mcbproj/
|-- project.mcproj
|-- Forms/
|-- Modules/
`-- AI/
    `-- Activities/
```

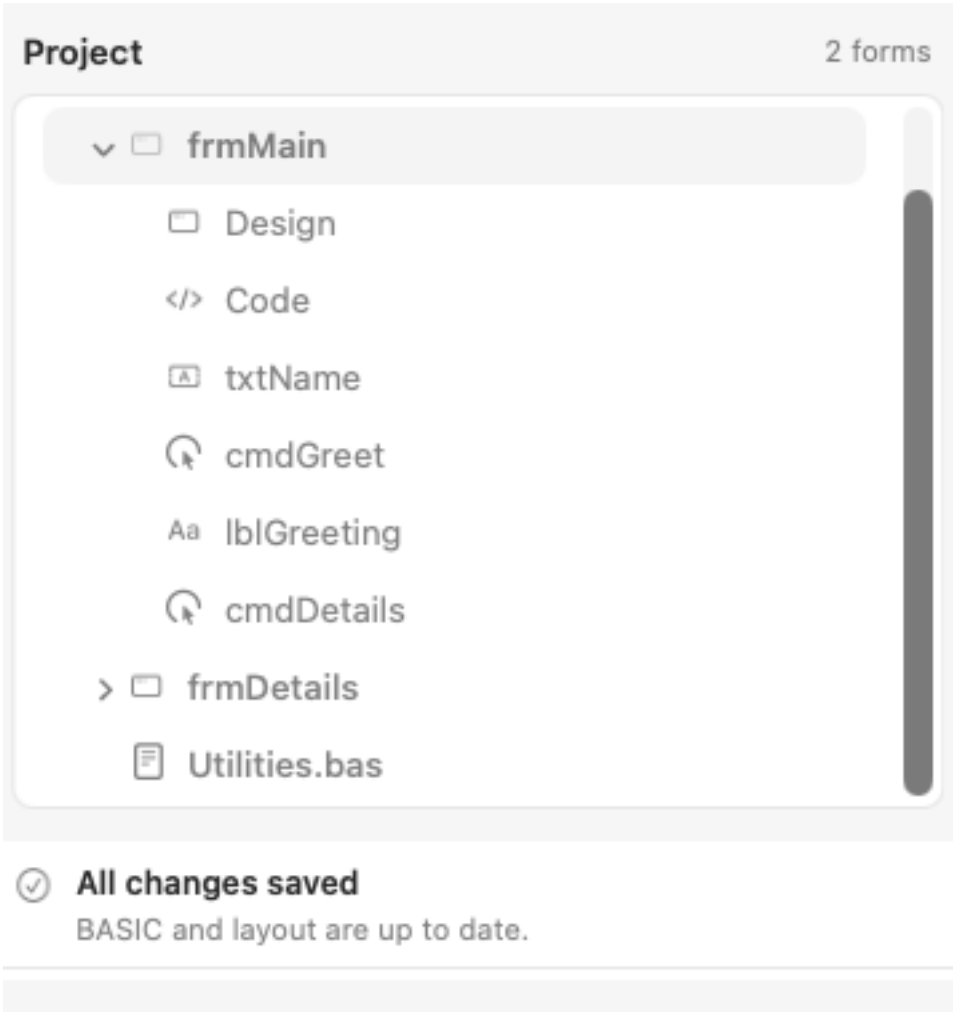


Figure 22.1: Project files in the IDE

How large a project can be

A project holds up to **1,000 documents**. Forms, UserControls, standard modules, class modules and test modules all count toward the same total; a form counts once, although it has a `.bas` file and a `.form.json` file.

- When a project reaches 1,000 documents, the **Add** toolbar button and the **Project > Add Form…**, **Add Module…**, **Add Class…** and **Add Existing File…** menu items are unavailable (in a Custom Control Plugin: **Add** and **Project > Add Control…**). The button's tooltip reads "Open a Desktop App or Custom Control Plugin with fewer than 1000 documents."
- If `project.mcproj` declares more than 1,000 documents, for example after a hand edit, McBasic™ does not open, check or build the project. It reports `MCB1001` with the message "a project may contain at most 1000 documents (forms, UserControls,

standard modules, classes and test modules together), but this one declares …". Remove unused documents from `project.mcproj`, or move shared code into a Library Plugin that the project references.

- The text of every kind of project, console applications included, has two size budgets. The code and layouts of forms and UserControls, standard modules and class modules together may use up to 8 MiB of UTF-8 text; test modules have their own 8 MiB. A single file may use up to 5 MiB. Over a budget, **Check BASIC** and **Build** report `MCB1001`, and completion, rename and the AI assistant report `MCB4600` with the fix: split large documents or move shared code into a Library Plugin, or split or shorten the test modules.

The AI chat works on projects of any size up to the 1,000-document limit. One chat sends at most 64 of those documents to the assistant at a time; choose them with the context chips. Chapter 16 lists the AI limits.

Files you edit and files McBasic™ creates

Keep these two groups separate in your mind.

File or folder	What it is for	Edit it?
`project.mcproj`	Project membership and project settings, including the app's bundle identifier, version, build number, and the optional `copyright` and `company`	Usually through McBasic™ (**Project > Project Settings…**)
`Forms/*.bas`	Code attached to a form and its events	Yes, in the Code view
`Forms/*.form.json`	Form layout and control properties	Usually through Design view
`Modules/*.bas`	Shared standard-module code	Yes, in the Code view
`*.cls`	A declared class module, when the project uses one	Yes, when supported by the release
`AI/Activities/`	Saved AI chats, when the project keeps them	No; managed by McBasic™
`Import/` (`Report.md`, `report.json`)	The migration report of a project created by **File ▸ Import Visual Basic 6 Project…** (Chapter 19)	No; it is not a project input, AI chats never read it, and you may delete it

Generated files are not part of the project. Build files live in `~/Library/Caches/McBasic/Builds` and McBasic™ removes them automatically. A published app or a plugin package goes where you tell McBasic™ when you publish or build, never inside the `.mcbproj`.

The exact folder names can change with a release. The safe rule is simple: edit source and layout through McBasic™, and treat build output as a result.

Where build output goes

Builds, prepared run output, publish staging, and package builds are created in the build cache, `~/Library/Caches/McBasic/Builds`, in a folder named after the project. McBasic™ cleans it automatically, and a later build recreates what it needs. Nothing in the cache is source, so never edit or back it up.

AI chats and the project

The per-project choice is in **AI Provider Settings** (the gear in the AI chat panel, or **McBasic > AI Provider Profile…**), in the **This project** section: **Save chats in this project**. It takes effect when you save the project, and McBasic™ then moves the existing chats.

- When it is on, the chats are kept in `AI/Activities` inside the `.mcbproj`

and travel with it, so sharing the project also shares its chats.

- When it is off, the chats stay on this Mac, in McBasic™'s folder in your user

Application Support folder.

Settings… > AI > Save AI chats in new projects sets the default for projects you create later; it is on unless you change it. Legacy folder projects always keep their chats on this Mac.

Chats that come with a project you received from someone else are never treated as approved on your Mac: any plan in them must be approved again.

A safe backup

Use this short routine before a large change:

1. Save the project in McBasic.
2. Quit or close the project if you are about to move it.
3. Copy the `.mcbproj` to a dated backup folder.
4. Open the copy and choose **Check**.
5. Make the change only after the copy opens and checks successfully.

Do not back up only the `.bas` file; copy the whole `.mcbproj`. A form also needs its layout file and the project manifest. A code-only copy may lose controls, names, or startup information.

Moving a project to another Mac

Copy the `.mcbproj` (or zip it and send it). Then open it from the new location with **File > Open Project…**.

If the project uses a local package or a file outside the `.mcbproj`, note that dependency before moving it. A project can open correctly and still fail to build if a required local input is missing.

Check the moved copy

1. Open the project from its new location.
2. Look at the project tree. Confirm that the expected forms and modules are present.
3. Open the main form in Design view. Confirm that its controls are visible.
4. Choose **Check**.
5. Run the smallest sample or action that proves the project is complete.

Expected result: the project shows the same forms and controls as the original, and Check reports no new diagnostic.

What not to share

Before sending a project to another person, inspect it for:

- database files that contain private customer data;
- exported reports with personal information;
- local paths that reveal names or folder structure;
- package files that you did not create or receive permission to distribute;
- passwords, API keys, tokens, or connection strings;
- saved AI chats, if the project keeps them (they travel with the project).

For a support report, a small copy with fake data is usually better than the original project. Keep the smallest project that still shows the problem.

Check your work

You understand the project layout when you can answer these questions:

- Which file inside the `.mcbproj` tells McBasic™ what belongs to the project?

- Where is the code for the main form?
- Where is the layout for that form?
- Which folders contain generated output?
- What must you copy to make a complete backup?

If you are unsure, return to the project tree and open one form in both Design and Code view. The two views should point to the same form document.

Tips and tricks

- **Colors in a `.form.json` can be names.** A color property holds `#RRGGBB`, an empty text or a system color name such as `"ButtonFace"`. Any other text is a project error (MCB2101).
- **Remove `icon` to go back to the McBasic™ icon.** Delete the `icon` line from the `application` block, or press **Use Default** in **Project > Project Settings…**. The image file stays in the project until you delete it.
- **Remove `appearance` to go back to macOS.** Delete the `appearance` line from the `application` block, or choose **macOS** in **Project > Project Settings…**. The file is then the same as before you used Classic.
- **Show Package Contents in Finder.** Control-click the `.mcbproj` and choose it to see `project.mcproj`, the `.bas` files and the `.form.json` files.
- **Edit the files McBasic expects you to edit.** Leave generated output alone. It lives in the build cache and is recreated when you build.
- **Back up before a large change.** Save the project, close it and copy the `.mcbproj`.
- **Check before you share.** Remove database files and exported reports that hold private data.

Diagnostics and troubleshooting

An error message is not a judgement. It tells you what McBasic™ could not do and where to look next.

When something goes wrong

When something goes wrong, use this order:

1. Stop and read the complete diagnostic.
2. Note the file, line, column, and the name of the member or control.
3. Open the location named by the diagnostic.
4. Make one small correction.
5. Save and choose **Check** again.
6. Run only after the new check succeeds.

Do not change five things at once. If the error disappears, you will not know which change fixed it. If a new error appears, the cause will be harder to find.

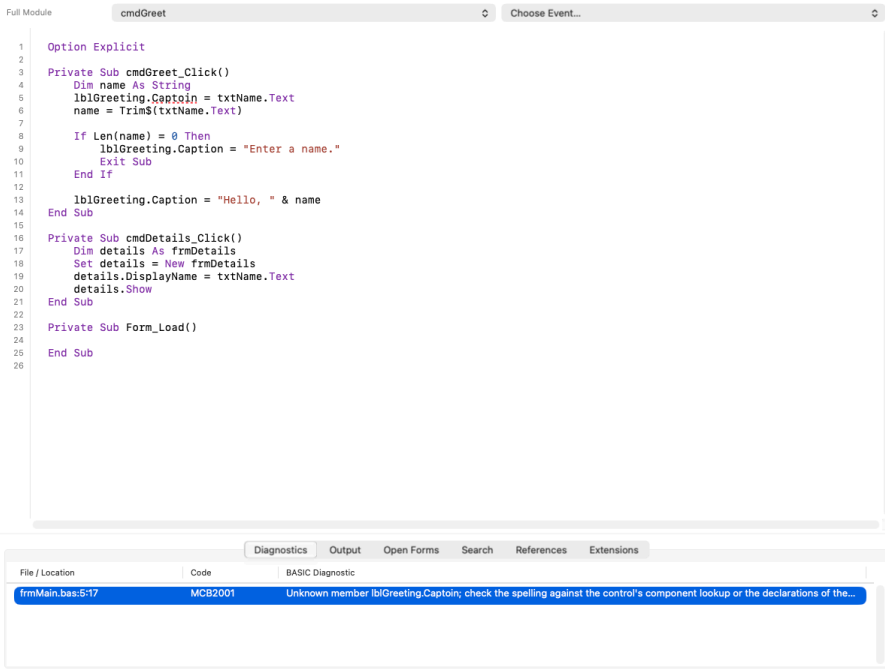


Figure 23.1: A diagnostic linked to source code

Common problems

The project does not open

Check these points:

- Are you opening the `.mcbproj` rather than a generated application?
- Did you copy the complete `.mcbproj`, not only some of its files?
- Is there a pending save or recovery message?
- Are you opening a copy that another program is still changing?

If the original opens but the copy does not, compare the complete folders. Do not start by editing the project manifest by hand.

Check reports an unknown name

An unknown name usually means McBasic™ cannot find the variable, control, form, or procedure that the line uses.

Try this example:

```
Private Sub cmdSave_Click()
    lblStatus.Caption = txtName.Text
End Sub
```

If the form actually contains `txtCustomerName` instead of `txtName`, Check will point to `txtName`. Select the diagnostic, confirm the real control name in the Inspector, and use that exact name:

```
Private Sub cmdSave_Click()
    lblStatus.Caption = txtCustomerName.Text
End Sub
```

Expected result: the unknown-name diagnostic is gone. If a second error appears, fix the next named problem in the same way.

The event does not run

First prove that the control name and event name are correct.

1. Select the control in Design view.
2. Check its **Name** property.
3. Open the event selector and choose the event you intend to handle.
4. Put one simple, visible action in the handler.
5. Choose **Check**, then run the project.

For example, make a button change a label before adding database or network code. If the label changes, the event path works and the larger problem is in the code you removed.

The form looks right but the code uses the wrong control

The text shown on a control is not necessarily its code name. `Save customer` may be the button's Caption, while `cmdSave` is its Name.

Use the Name value in code. Use the Caption value when you describe what the user sees.

A property value is rejected

Check the value's type and range. A width must be a valid number. A caption is text and needs quotes in code. A property that is read-only cannot be assigned at runtime. A color is `"#RRGGBB"` text: write `lblTotal.ForeColor = vbRed`, `RGB(255, 0, 0)` or `"#FF0000"`, never a VB6 number such as `&HFF&`.

Make a small test with a known value, then choose **Check**. If the Inspector shows the property but the code does not accept it, read the component help for that release; design-time and runtime properties are not always the same.

Build works but Run does not

Read the first Run or build diagnostic. Then check:

- whether the project was saved before the build;
- whether the output folder is writable;
- whether a required package or input file is missing;
- whether the generated app is being launched from an old output folder;
- whether the source and layout changed after the last successful build.

Run a clean, small sample if you need to separate a project problem from a machine or installation problem.

Data is missing or appears old

Check the path of the database or file first. Then check whether the code is opening a different copy than the one you are viewing.

For a write operation, confirm the transaction and save steps. For a read operation, print or display the path and the number of rows returned while you test. Never use real customer data for a first diagnostic.

The Visual Basic 6 import reports a problem

Codes `MCB5600` to `MCB5699` belong to the importer. `MCB5600` to `MCB5619` refuse an import: an alert says **This Project Cannot Be Imported** or **The Project Was Not Imported**, and nothing is written. `MCB5620` to `MCB5699` are entries of the migration report and `' IMPORT` comments in the imported code; they do not stop the import. Every message names its fix. Chapter 19 explains the import, the report window and **Finish with AI…**.

The AI assistant reports a problem

AI messages appear in the chat panel as cards, not in the Results area and not in separate windows. Each one says whether anything was sent and whether the project changed. During a plan, a problem pauses the plan: the plan card names the reason and shows the button that continues it.

Message or code	What it means	What to do
"Codex could not start. Nothing was sent…"	McBasic™ could not launch the Codex runtime from the ChatGPT app.	Open AI Provider Settings, choose **Autodetect Codex**, choose **Save**, and send again.
"No models available — sign in, then Refresh"	Codex is not signed in, or has not listed its models yet.	Choose **Sign In…** in AI Provider Settings, then **Refresh**.
"This chat has no context chips, so the AI cannot see the project…"	A plan stage was about to run with no chips.	Add the documents with **+ Add**, then press **Resume** on the plan card.
"You changed the project during stage 3: restarting stage 3 on the current version."	The project changed before the stage could be applied.	Nothing to do: the stage runs again on your current version.
"The project kept changing while this stage ran…"	The stage restarted twice in a row and the project changed again.	Finish the other change, then press **Resume**.
"Project documents changed. Review the updated context…"	A document was added, removed, or renamed.	Review the chips and press Send again.
`MCB4732 [Form/member] …`	A generated layout or stage does not fit McBasic™; the brackets name the form and member.	Let the repair attempts run; if all three fail, ask for a smaller change.
`MCB4732 [op-add-helpers addModule] targetId must be 'module:modhelpers' for module 'modHelpers' (got 'test:modTests'); addModule always adds a module (addTest adds a test), and every operation of this change set on the new module uses 'module:modhelpers'.`	A generated change used a wrong ID that McBasic™ could not correct automatically because it was ambiguous, for example two new documents of one stage used the same ID (an unambiguous wrong ID for a new module, test or resource is corrected without a message); the brackets name the	Nothing to do: the next repair attempt uses the expected value. If all three attempts fail, choose **Retry stage**.

Message or code	What it means	What to do
	operation, and the message names the expected value.	
"Stage 3 did not pass Check after 3 attempts"	No generated version of the stage passed Check.	Nothing of the stage was applied. Read the diagnostics under **Details**, then choose **Retry stage**, **Skip stage**, or **Stop plan**.
"Dependency rejected. The stage was not applied."	You rejected a plugin reference or migration file.	Press **Resume** to generate the stage again, or **Stop plan**.
"The … is N characters, X over the B-character budget; nothing was truncated or sent. …"	The request, or the context the chips select, is larger than the request budget.	Follow the advice at the end of the message, or choose a larger **Request size:**; then press **Retry**.
`MCB4702`	The toolchain refused a request over its budget. Nothing was sent.	As above.
`MCB4733` "The AI kept requesting reference lookups without finishing…"	The assistant consulted McBasic™'s reference 500 times in one request without finishing. Nothing was applied.	Send again, or rephrase the request as a smaller step.
`MCB4770`–`MCB4777` under "… layout notes from McBasic™ …" on a stage card	Layout advisories: the applied stage does not follow the Mac layout guide in one place, for example a control 12 from the window edge. They are advice, not errors.	Fix the form in the designer, or ask the assistant. Chapter 16, "Layout notes", lists every code.
"The AI sent nothing for 10 minutes, so McBasic™ stopped this request…"	The provider sent nothing for 10 minutes. Nothing was changed.	Press **Retry** on the error card or the plan card. If it happens again, split the stage or remove chips it does not need.
"This AI request ran for 60 minutes, so McBasic™ stopped it…"	One AI request reached the 60-minute limit. Nothing was changed.	Press **Retry**; if it happens again, split the stage.
"Enter an AI request up to 40,000 characters."	The message is longer than the prompt part of the budget (40,000 characters	Shorten it or split the request.

Message or code	What it means	What to do
	by default; the number follows **Request size:**).	
"AI provider changed to … Resume to continue with it."	You changed the provider or model in AI Provider Settings while a plan ran. Nothing was sent.	Press **Resume** to continue with the provider now selected, or set it back first.
"No AI provider is selected. Choose one in AI Settings, then press Resume."	The plan reached a request with no provider selected.	Choose a provider, then press **Resume**.
"Choose an AI provider in AI Settings, then press the button again. Nothing was sent."	You pressed **Resume**, **Retry**, **Retry stage**, or **Approve and run** with no provider selected.	Choose a provider, then press the button again.
"Provider settings updated. Press Resume again."	**Resume** opened AI Provider Settings because the provider's settings were incomplete, and you completed them. Nothing was sent.	Press **Resume** again.
"The checked change was set aside by your new message…"	You sent a message while a checked stage waited for you.	Press **Resume** to generate the stage again.
"Remaining stages depend on stages that cannot run."	The plan's remaining stages wait for stages that cannot complete.	Press **Resume** to run the waiting stage anyway, or **Stop plan**.
"This action is no longer available."	The plan's state changed since the card offered the button.	Use the buttons the newest cards show now.
"Wait for the current AI request or press Stop."	You pressed a card button while a request runs.	Wait, or press **Stop**.
"Use Edit → Undo to step back."	Something changed after this AI change, so its card cannot undo it.	Use **Edit > Undo**.
"This chat's project is no longer the open project."	You pressed **Undo** on a card of a project that is not open in this window.	Use **Edit > Undo** in that project's window.
"Wait for the current build or run to finish, then press Undo again."	A card's **Undo** was pressed during a build, Check, or run.	Wait, then press **Undo** again.

Message or code	What it means	What to do
"Finish or cancel the property you are editing, then press Undo again."	A designer property is still being edited.	Press Return or Esc in the property, then **Undo** again.
"Your property change was saved. Use Edit → Undo to step back."	The pending property was committed first and is now the most recent change.	Use **Edit > Undo**.
"This project has no paused AI plan to resume."	**Edit > Show Paused AI Plan** found no paused plan.	Nothing to do.
"This project has no saved AI activity yet."	**Edit > Open Latest AI Activity** found no saved chat.	Nothing to do.
"This card belongs to an earlier plan or change. Use the newest card."	You pressed a button on an older card.	Use the newest card.
"Send a message to reconnect this chat, then try again."	The chat has no provider connection yet.	Send a message, then press the button again.
"This chat is too large to save (over 2 MiB)…"	The chat record reached its size limit. The reply stays visible.	Start a new chat to keep saving.
"This chat is full. Start a new chat to continue."	The chat reached its size limit.	Choose **+ New chat** in the chat menu.

Chapter 16, "Work with the AI assistant", explains each case in more detail.

Check messages and their fixes

Every Check message names its fix. These are the most common ones, grouped as the language reference groups the rules:

- **Declarations.** "Dim initialisers are not supported": write `Dim total As Long`, then `total = 5` on its own line. "'As New' is not supported": write `Dim items As Collection`, then `Set items = New Collection`. "Undeclared symbol": declare the variable with `Dim name As Type`, or correct the spelling. In a standard module, `Me` is not a variable to declare: it exists only in the code of a form, a UserControl or a class, so name the form instead, as in `frmMain.lblScore.Caption`. A control name alone works only in its own form's code; elsewhere "lblScore is a control of frmMain" says to write `frmMain.lblScore`. A UserControl's inner controls stay private: use a `Public Property` of the UserControl. "Type suffixes are not supported": write `As String` instead of `$`.

- **Format masks.** "This Date Format mask is not supported": use `yyyy-MM-dd`, `dd/MM/yyyy`, `MM/dd/yyyy`, `HH:mm:ss`, `yyyy-MM-dd HH:mm:ss`, `dd/MM/yyyy HH:mm:ss` or `MM/dd/yyyy HH:mm:ss`, or build the text from `Day`, `Month`, `Year` and `MonthName`. "This numeric Format mask is not supported": use only `0`, `#`, `,`, `.` and a final `%`, as in `"0"`, `"0.00"`, `"#,##0"`, `"#,##0.00"` or `"0%"`.
- **Types and conversions.** "Conversion from String to Long is never implicit": convert with `CLng(...)`; the message names the function for each type.
- **Procedures and arguments.** "ByRef count requires writable Long storage": pass a variable of that type, or declare the parameter `ByVal`. "A typed Optional parameter requires an explicit constant default": write `Optional ByVal count As Long = 0`. "Parenthesized statement arguments require Call": write `Work 1, 2` or `Call Work(1, 2)`. "A Function result must be used": assign it to a variable.
- **Async.** "Await must be a statement on its own": put the awaited value in a local first. "Async procedure parameters must be ByVal": write `ByVal` on every parameter.
- **Arrays.** "For Each over arrays requires a scalar Variant variable": declare the loop variable `As Variant`.
- **Control arrays.** "'txtCell' names both a control with an Index and one without": give every control with that name an Index, or rename one. "Handler 'txtCell_Change' handles the control array txtCell, so it needs Index first": write `Sub txtCell_Change(Index As Integer)`. "Handler 'txtOne_Change' takes Index, but txtOne is not a control array": remove `Index`, or set the control's **Index** in Properties. "txtCell is a control array: write txtCell(index)…": add the Index in parentheses. "For Each over the control array txtCell needs a loop variable declared As TextBox or Control": declare the loop variable with that type. "Load txtCell needs the Index of the new member: write Load txtCell(index)…": add the Index. "txtOne is not a control array, so Load cannot add a copy of it": give the control an Index in Properties first. "Load is not available for txtCell: its first member has Dock = …", "…are … controls, which hold other controls", "…TabPage arrays…" and "…inside a SplitContainer": add those members in the designer and show them with `.Visible = True`. "Load txtCell(40000) is outside the Index range": use 0 to 32767. "Unload txtCell needs the Index of the member to unload": write `Unload txtCell(index)`. "txtName was placed in the designer and cannot be unloaded": hide it with `Visible = False`. The warning MCB2404, "Load txtCell(0) always raises error 360…" or "Unload txtCell(0) always raises error 362…", means a literal Index names a member from the designer. At run time, error 340 "Control array element doesn't exist" names the array and the missing Index. Errors 360 (Object already loaded), 361 (Can't load or unload this object), 362 (Can't unload controls created at design time), 364 (Object was unloaded) and 365

(Unable to unload within this context) come from `Load` and `Unload`; each message names the fix, and Chapter 6 explains them.

- **Menus.** The Menu Editor (Chapter 8) shows most of these live under its list and blocks **OK** on an error; Check and opening the form report the same rules. MCB2103 is raised when the form opens or is checked. "this form has the Menu component … and menu items": a form uses one or the other, so remove the Menu component or the menu items. "has Parent '…', which is not the id of a menu item": use the id of another menu item, or leave Parent empty for a menu title. "leads back to itself": give one item of the loop an empty Parent. "is N levels deep": menus have at most 5 levels, so move the item nearer the menu bar. "the menu title … has Caption -", "the separator … has items under it": a separator sits under a menu title and holds nothing; change its Parent, Caption or its items. "moves to the application menu and cannot hold items": `mnuSettings` holds no items, so give its items another Parent. "<name> moves to the application menu on macOS, so it cannot be a menu title; give <name> a Parent (for example the Help menu).": indent `mnuSettings` under another menu. (`mnuAbout` is an ordinary item and never raises this message.) "sets WindowList, which only a menu title can set", "both set WindowList": keep WindowList on one menu title. "the menu control array … has members under different parents": give all members the same Parent. MCB2104, "has the shortcut …": only an item under a menu title can have a shortcut, and two items cannot share one; set the Shortcut to None or choose another. The warning MCB2405, "has the shortcut … which macOS keeps for …": the system takes Ctrl+Q (Command–Q, Quit) and Ctrl+H (Command–H, Hide) first, so the item never receives it; choose another Shortcut. Ctrl+M and Ctrl+W are allowed: a menu item with Ctrl+M takes Command–M from the automatic **Minimize**. The warning points at the form's layout file, and does not block the Menu Editor. MCB4606 is **Convert to Menu Editor…** refused. "the form has no Menu component with id …; select the Menu component again, then convert": select the Menu component and try again. "the code of … does not parse (…); fix that line, then convert … again": correct the syntax error in the form's code first. "… is not a Menu component; Convert to Menu Editor works only on the old Menu component, so select that one": select the old Menu. "this form already has menu items, so … cannot become another menu; open the Menu Editor to add its items there, then delete …": a form uses the Menu component or menu items, not both, so add the items in the Menu Editor and delete the old Menu. MCB2105 covers `PopupMenu`: "call it in the code of the form that holds the menu": move the call into that form; "needs a menu item of this form": pass a menu item, not a control or a variable; "is not a menu title" and "has no items under it": pass a menu title that has items; "needs both x and y, or neither": write `PopupMenu mnuEdit, , x, y`; "accepts the menu title, then optional flags, x, y and defaultMenu": pass the arguments by position; "defaultMenu … needs a menu item under the menu title": pick an item of that menu. MCB2101 "is a menu item of the form layout and cannot be unloaded": hide it with `Visible = False`, or use a menu control array.

- **Objects.** "Object reference assignment requires Set": add `Set`. "With requires a declared Form or Control reference": write the variable before each member. "lvwTasks.Items is a read-only Collection in code and cannot be assigned; change its items with lvwTasks.Items.Clear and lvwTasks.Items.Add instead": the one-item-per-line text belongs only in the Inspector, so fill the list in code with `Clear` and `Add`. The same message covers `Items` of a `ListBox`, `ComboBox` or `ImageCombo`, `Images` of an `ImageList` and `Nodes` of a `TreeView`.
- **Error handling.** "Try cannot mix with classic error recovery": use either `Try ... Catch` or `On Error` in one procedure.
- **Unknown members.** "Unknown File member Copy; File members: ...": the message lists every member of that service or object.
- **VB6 features.** "CreateObject is not available in McBasic™; ...": the message names the replacement, and the language reference appendix lists every one. A VB6 constant that McBasic™ does not have says so in the same way, for example "VB6 constant vbButtonFace is not available in McBasic™: Windows system colors do not exist on macOS; ...". The VB6 constants that McBasic™ does have, such as `vbCrLf`, `vbYesNo` and `vbKeyReturn`, never need a `Dim`.
- **Built-in constants and colors.** "vbTab is a built-in VB6 constant and cannot be assigned": give your own variable another name. "lblTotal.ForeColor takes "#RRGGBB" text, not a number": assign `vbRed`, `RGB(255, 0, 0)` or `"#FF0000"`. "RGB(...) is "#RRGGBB" text in McBasic™, not a number": keep the color in a `String` variable, not a `Long`.
- **Layout.** Property messages give the allowed values or range, such as "Dock must be None, Left, Top, Right, Bottom or Fill".
- **Publishing for Windows.** "MCB2302: Publishing for Windows needs the Windows .NET packs, which are missing from this McBasic toolchain. Reinstall or update McBasic.": the installation lacks the Windows packs; install McBasic™ again or update it. Mac publishing is not affected. "MCB2301: publishing for Windows requires a DesktopApp project.": publish a Plugin or a library for the Mac only, or a desktop application for Windows. "MCB2406: <Member> is not available on Windows yet in this McBasic version; the app is built, but this call raises MCB4415 on Windows.": a warning, not an error. The `.exe` is still produced. In this version every service works on Windows, so you see it only if a future service is not ported yet. At run time on Windows such a call stops with error 5 and "MCB4415: <Member> is not available on Windows yet in this McBasic version." Chapter 20 lists what works on Windows.
- **Publishing for Linux.** The same three messages name Linux: MCB2302 when the Linux .NET packs are missing (install McBasic™ again), MCB2301 "publishing for Linux requires a DesktopApp project", and the warning MCB2406 for each use of a service that Linux does not have yet (the color, font and printing dialogs, the

`AudioPlayer` and the `VideoPlayer`): the file is still produced, and on Linux that call stops with error 5 and MCB4415. Chapter 21 lists what works on Linux.

- App icon. "MCB2407: the app icon AppIcon.icns is a macOS .icns file, so the Windows app shows the McBasic icon; choose a square PNG in Project > Project Settings to use one icon on both." (or the same for a Windows .ico on a Mac publish): a warning, not an error; the app is published with the McBasic™ icon on that system. A wrong application.icon is a project error (MCB1001); Chapter 22 lists each message and its fix, and the appendix "Check messages by code" lists every code in one line.

A useful minimal test

When a large handler fails, reduce it to one visible result:

```
Private Sub cmdTest_Click()
    lblStatus.Caption = "The event works"
End Sub
```

If this works, add back one operation at a time:

1. read the input;
2. validate the input;
3. call the data or file operation;
4. display the result;
5. handle the failure case.

Check after each addition. This turns one large mystery into a short list of small questions.

When to ask for help

Other McBasic™ developers are the quickest place to ask. **Help > McBasic Community** opens the community page with the invite to the McBasic™ Discord server; open one post per question in its help channel. For a private request, **Help > Support** opens the contact page.



Figure 23.2 - Scan to join the McBasic™ community (mcbasic.com/community.html)

Include the smallest useful report:

- the McBasic™ version;
- the macOS version and Mac architecture;
- the steps that reproduce the problem;
- the exact diagnostic text;
- the smallest project that still fails;
- what happened after you chose **Check**, **Build**, or **Run**.

Remove passwords, keys, private data, and personal paths before sharing the project.

Check your work

You have handled a problem well when you can explain:

- what McBasic™ reported;
- where the problem was;
- what one change you made;
- how Check or Run proved the change.

Keep that explanation in a project note. The next error may have a different cause, but the same recovery loop will still help.

Tips and tricks

- **Run Check first.** The **Diagnostics** list fills after a Check. Errors are also underlined in the editor, and hovering shows the message. The automatic Check can be turned off in **Settings… > Code > Check automatically**.
- **If your app seems fine but behaves oddly, read the Output pane.** Errors your code catches show there as "Handled runtime error …" with file and line when you Run from the IDE.
- **Check the version when a fix does not seem to work.** After rebuilding McBasic™, open **McBasic > About McBasic** or look at the Welcome window.
- **Read a menu message as a path.** The names in an MCB2103 message are the menu items; follow the Parent chain from the menu title down to the item it names.
- **Reduce a failing handler to a minimal test.** One button and one visible result shows the cause quickly.
- **Include the version in a help request.** Add the macOS version and the steps too.

Appendix 4 — Check messages by code

Every message that McBasic™ shows for your project, your code, a layout, an import or an AI change carries a code, `MCB` and four digits. This appendix lists every code in order, one line each: the area it comes from, what it means and what to do. Chapter 23 explains the most common ones at length, with examples; use this appendix when you meet a code that chapter does not cover.

How to read a message:

- **The code names a rule, the text names the case.** `MCB2101` alone has

hundreds of texts, one for each thing that cannot be assigned or called. The text always says what to change; the line below says where the rule comes from.

- **The area says who reports it.** *Code* is the compiler, on your BASIC code

and declarations. *Project* is the project file, the build and the publish. *Editor* is the designer and the code editor. *Advisory* is a layout or code rule that the AI assistant's changes pass before they are applied. *AI* is the assistant's exchange with the toolchain. *VB6 import* is **File ▸ Import Visual Basic 6 Project…**. *Packages* is plugins and offline packages. *Protocol* is the exchange between the IDE and its toolchain, which you should not see in normal use.

- **Where the line is.** Diagnostics in the **Results** area show the file,

the line and the column, and a click selects the token in the code (Chapters 15 and 23).

- **Warnings and advisories do not stop you.** A publish warning (MCB2406,

MCB2407) and an advisory (MCB4770 to MCB4799) describe something to improve; the app is still built.

Code	Area	Meaning	What to do
MCB1001	Code	A statement, declaration or file is not written as McBasic™ expects: a syntax error, a missing `End If` or `Next`, a file over 5 MiB, a resource the project declares but does not have, a `Call` without parentheses, a block nested more than 128 levels. Also a project-file error (a wrong	Read the text: it names the construct and the correct form. Fix the line it points to, then **Check** again.

Code	Area	Meaning	What to do
		`application.icon`, an unknown target).	
MCB2001	Code	A name is unknown: a symbol that is not declared (`Option Explicit` is always on), a member the control does not have, an event not declared with `Public Event`, `Me` outside a form, a console program without `Sub Main()`.	Declare the variable with `Dim ... As`, check the spelling against the control's component lookup, or declare the event before `RaiseEvent`.
MCB2002	Code	A name is ambiguous or collides: two public procedures, properties or Enum types with the same name in different modules, a constant with the name of a local, a class that implements itself.	Qualify the name with its module (`Module1.Total`), or rename one of the two.
MCB2101	Code	A value cannot be used that way: a wrong type in an assignment or call, a read-only property assigned, a missing argument, an array with more than 60 dimensions, a leading dot outside `With`, an MDI form shown as a modal dialog.	The text gives the rule and the accepted form; change the expression or the declaration.
MCB2102	Code	A `ByRef` or typed array parameter is not given storage it can write: an expression, a property, a temporary value, or a fixed-size array passed to a procedure that resizes it.	Pass a variable declared with the same type, or an array declared without bounds; pass the array itself, without parentheses.
MCB2103	Code	The menu of a form is not consistent: an item whose `Parent` does not exist or loops back, an item more than 5 levels deep, a separator with items under it, a form with both the old Menu component and menu items.	Open the **Menu Editor**: give each item an existing parent, keep at most 5 levels, give a separator no children, remove the old Menu component.
MCB2104	Code	A shortcut is misplaced or duplicated: a menu title or a separator with a shortcut, or two items with the same shortcut.	Set the shortcut of the title to `None`; give each item its own shortcut.
MCB2105	Code	`PopupMenu` is called wrongly: as a function, outside the form that holds	Write `PopupMenu mnuEdit` as a statement in the form's

Code	Area	Meaning	What to do
		the menu, with a name that is not a menu title, or with only one of `x` and `y`.	code; give both `x` and `y`, or neither.
MCB2201	Code	A class or a `WithEvents` handler does not honour an interface or an event: a procedure, property or event with a different signature, a handler with the wrong parameters, a subscription to a private event.	Give the procedure or handler exactly the parameters of the interface or event; declare the event `Public`.
MCB2301	Project	The project or the command do not match: a publish of a Plugin as an app, a Build Package of a desktop app, a project name or template that is not allowed, a target that is not one of the six, a Mac publication profile or `--format package` with a Windows or Linux target.	The text names the allowed values; change the command, the project kind or the target in `project.mcproj`.
MCB2302	Project	The Windows or Linux .NET packs are missing from this McBasic™ installation, so a publish for that platform cannot start.	Nothing is wrong with your project: install McBasic™ again or update it. Mac publishing is not affected (Chapters 20 and 21).
MCB2401	Code	A `Function` never assigns its return value, so the caller gets the type's default.	Assign the function's name before `End Function`, or make it a `Sub`.
MCB2402	Code	A variable has no `As` clause, so it is a `Variant`.	Write `Dim total As Double`; each variable needs its own `As`.
MCB2403	Code	A name in your project is also the name of a control type, so `As <Name>` means your object, not the control.	Rename your class, form or variable, or qualify the control type.
MCB2404	Code	`Load` or `Unload` would always fail at run time: `Unload` on a control placed in the designer (error 362), or `Load` with an index the array already has (error 360).	Hide the designer-placed control instead of unloading it; load a new index.

Code	Area	Meaning	What to do
MCB2405	Code	A menu shortcut is one that macOS keeps for itself (such as Command+Q or Command+H), so your item never receives it.	Choose another shortcut in the **Menu Editor**.
MCB2406	Project	*Warning.* A Windows or Linux publish found a call to a service that platform does not have yet; the app is still built, and the call raises MCB4415 there.	Wrap the call with `On Error` or `Try ... Catch`, or offer an alternative on that platform (Chapter 21 lists the Linux services).
MCB2407	Project	*Warning.* The app icon is a `.icns` file on a Windows publish, or a `.ico` file on a Mac publish, so that app shows the McBasic™ icon.	Choose a square PNG in **Project > Project Settings…** to use one icon on all systems.
MCB4001	Protocol	The toolchain could not start or run the generated app, or a required toolchain file (such as the Windows icon or manifest) is missing.	Rebuild from Xcode in a development checkout, or install McBasic™ again; the text names the missing file.
MCB4002	Protocol	The generated app ended with an error the toolchain reports as it is.	Read the text that follows the code; retry after fixing what it names.
MCB4100	Protocol	A request from the IDE to the toolchain is malformed, too large (over 32 MiB) or has fields that do not belong to its operation.	Update McBasic™ so the IDE and the toolchain are the same version.
MCB4101	Protocol	The IDE protocol version is not 1.	Update McBasic™.
MCB4102	Protocol	A new project must be a `.mcbproj` package in an existing parent folder, not inside another project or app, and never over existing content.	Choose another location or name.
MCB4103	Protocol	The project changed while the toolchain was opening or reading it.	Reopen the project, or retry.
MCB4201	Protocol	The toolchain reported an internal error; the text follows the code.	Retry; if it repeats, report the message.
MCB4300	Editor	An event handler cannot be created: the handler or event name is not an	Use a plain name; split a very large module.

Code	Area	Meaning	What to do
		ASCII identifier of 1 to 128 characters, or the file would exceed 5 MiB.	
MCB4301	Editor	The event handler the IDE wanted to open does not exist; the source was not changed.	Create it from the **Event** selector.
MCB4302	Editor	The module must be repaired before an event can be edited: a procedure header without its `End Sub` or `End Function`, or a structural error the text names.	Fix the structure in **Code**, then try again.
MCB4303	Editor	The handler's name already belongs to a module constant or variable; the source was not changed.	Rename the constant or variable.
MCB4304	Editor	The event requires a `Sub` with the parameters the text names; the source was not changed.	Write the handler with exactly those parameters.
MCB4415	Project	*Run time.* A service that is not available on Windows or Linux yet was called; the statement stops with error 5.	Handle it with `On Error` or `Try ... Catch`; see MCB2406, which warned about it at publish time.
MCB4503	Project	A save of the project was interrupted, and its journal must be recovered before the project is read or built.	Open the project in the IDE, which recovers the save; then **Check**.
MCB4600	Protocol	The IDE sent a document, patch or layout operation the toolchain cannot accept (a duplicate or unsupported field, a check without every authoritative document).	Save the project and retry; update McBasic™ if it repeats.
MCB4601	Protocol	A source or layout hash does not match the supplied text, or an event the editor does not bind yet was requested.	Save the project and retry.
MCB4602	Editor	A semantic rename needs a project without diagnostics; no source was changed.	Fix the errors, then rename again.
MCB4603	Editor	The rename would capture another symbol or change what existing code refers to; no source was changed.	Choose another name.

Code	Area	Meaning	What to do
MCB4606	Editor	**Convert to Menu Editor** cannot run: the selected control is not the old Menu component, its code does not parse, the form already has menu items, or an item name is invalid or taken.	Select the Menu component, fix the line the text names, or choose another name.
MCB4700	AI	The AI request is not valid: an invalid model, endpoint or credential, more than 64 context documents, a duplicate document, a plan that does not belong to the activity, a context document that is not in the saved project.	Check **AI Provider Settings…** and the context chips; save the project and send again.
MCB4701	AI	The AI authorization or the context selection is stale: the project changed after you approved them.	Send the message again; approve the context again if asked.
MCB4702	AI	The prompt, its preamble or the authorized context exceed the request budget; nothing was sent.	Remove context chips, shorten the message, or raise **Request size** in the AI chat settings (Chapter 16).
MCB4703	AI	An AI mode, preamble or project-map value is not one the toolchain knows.	Update McBasic™; the IDE and the toolchain must be the same version.
MCB4730	AI	The AI change set is missing from the request.	Let the AI plan again.
MCB4731	AI	The AI change set is stale: a document changed after the AI planned, or an operation names the wrong original hash.	Let the AI plan again from the current project; do not edit the project while a stage runs.
MCB4732	AI	An AI operation failed validation: an invalid or reserved BASIC name, a layout operation the compiler rejects, an invalid migration id, or any Check error inside the AI's change.	Read the text after the code: it is the compiler's message for the AI's change. The assistant retries up to three times; then the stage card shows the failure (Chapter 16).
MCB4740	AI	The stage request is incomplete, or the attempt number is not 1 to 3.	Update McBasic™.

Code	Area	Meaning	What to do
MCB4741	AI	The AI candidate project is refused: a symbolic link in the project, a copy over its size bound, an input outside the project folder.	Replace the links in the project with real files and folders.
MCB4742	AI	The AI candidate cannot become a plan: no manifest, a failed candidate, a document outside the workspace, or a binary file the plan tries to change.	Keep binary resources out of AI plans; let the AI plan again.
MCB4743	AI	The unsaved AI project files that accompany a stage are wrong: too many, too large, not declared by the project, already saved, or with a wrong SHA-256.	Reopen the project before the next stage.
MCB4770	Advisory	Two visible controls in the same parent overlap.	Move or resize one of them.
MCB4771	Advisory	A control extends beyond the client area of its parent.	Move it inside or enlarge the parent.
MCB4772	Advisory	A control is closer than 20 to a window edge.	Keep 20 between controls and the window edges.
MCB4773	Advisory	A single-line label on the row of a field does not have the row's `Top` and `Height`.	Give the label both (Height 32 beside a multi-line TextBox).
MCB4774	Advisory	Buttons in one row have different `Top` or `Height`.	Give them the same `Top` and `Height`.
MCB4775	Advisory	An OK, Save or Done button is not the default button, or Cancel is not the cancel button.	Set `defaultButtonId` and `cancelButtonId` in the layout actions.
MCB4776	Advisory	A form holds a list and three or more input fields.	Move the editing fields into a modal dialog opened with `ShowDialogAsync`.
MCB4777	Advisory	A DataGrid shows a column name with _ as its header.	Write `name
MCB4778	Advisory	A long procedure has no comment.	Add a comment block that says what it does, its parameters and its result.

Code	Area	Meaning	What to do
MCB4779	Advisory	A form, module, class or user control has code but no header comment.	Add a comment block at its top.
MCB4780	AI	The AI called its `ask_user` tool with wrong arguments (no question, bad options, an invalid key).	Send your message again.
MCB4781	AI	The AI sent more than two invalid tool calls in one reply, so McBasic™ stopped it.	Send your message again.
MCB4782	AI	The AI kept calling tools after ending its turn.	Send your message again.
MCB4783	Advisory	A `MsgBox` asks a question without an icon.	Add `vbQuestion` for a confirmation, `vbExclamation` for a warning, `vbCritical` for an error.
MCB4785	AI	The AI scope is malformed: it must be 1 to 32 items, each a control, array, form, document or procedure with its document.	Remove the chips and add them again.
MCB4786	AI	A scope chip names a control, form, document or procedure that is not in the project any more.	Remove the chip and send again.
MCB4787	Advisory	A caption is much wider than its control, so its text is cut.	Make the control wider, or shorten the text.
MCB4788	Advisory	Too many buttons in one container.	Keep at most 4 buttons for the main commands and move the rest to a menu.
MCB4789	Advisory	Buttons in one row have different widths.	Give them one `Width`, the widest.
MCB4790	Advisory	A button, field, check box, option button or switch is too small to click.	Give it `Height` 32, the McBasic™ control height.
MCB4791	Advisory	Labels and fields of one column do not line up, or a label is far from its field.	Give them one `Left`; keep 8 between a label and its field.

Code	Area	Meaning	What to do
MCB4792	Advisory	A form with four or more command buttons has no menu bar.	Add menu items for the commands.
MCB4793	Advisory	A dialog has no default or no cancel button.	Set `defaultButtonId` and `cancelButtonId` in the layout actions.
MCB4794	Advisory	More than one button is the default button.	Keep exactly one.
MCB4795	Advisory	The default button is destructive (Delete, Remove, Clear, Reset…).	Make Cancel or the safe choice the default.
MCB4796	Advisory	A font size is below 10.	Use 13, the system size (17 or 22 bold for headings).
MCB4797	Advisory	`ForeColor` on `BackColor` has a contrast below 4.5:1 (3:1 for bold or large text).	Change one colour, or remove both.
MCB4798	Advisory	A caption may be clipped: it is slightly wider than its control.	Make the control wider.
MCB4799	Advisory	A control is closer than 8 to an edge of its container, so it looks glued.	Keep 8 or more from the edges.
MCB4800	Editor	**Find references** cannot answer: the position is outside the compiled source, the project has diagnostics, or two documents share a path.	Fix the diagnostics first and place the cursor on a name.
MCB4801	Editor	There is no bound symbol at this position.	Place the cursor on a local, parameter, constant, field, procedure, form type or control name.
MCB4802	Editor	A reference found by the editor does not identify its token, or lies outside its source snapshot.	Save the project and try again.
MCB4900	Editor	Code completion cannot answer at this position: a leading dot outside `With`, a `Catch` that must be repaired, or an internal error the text names.	Fix the line, then type again.
MCB4901	Editor	Code completion failed with the message the text carries.	Save the project and try again.

Code	Area	Meaning	What to do
MCB5001	Protocol	The toolchain reported an internal error; the text follows the code.	Retry; if it repeats, report the message.
MCB5201	Code	An `Assert` in a test is called with the wrong arguments: a mixed Variant and typed comparison, a wrong number of arguments, or an unsupported `Assert` member.	Use `AreEqual`, `IsTrue`, `IsFalse`, `IsNull` or `Fail` with the arguments the text names.
MCB5600	VB6 import	The chosen `.vbp` or folder cannot be used: it does not exist, it is a symbolic link, or it is a project group without a usable project.	Choose the `.vbp` file itself, or one of the projects the text lists.
MCB5601	VB6 import	A file the `.vbp` lists is outside the project's folder, has an absolute Windows path, is a symbolic link, is empty, or matches several files on disk.	Copy the file next to the `.vbp`, list it by a relative path, keep one file per name.
MCB5602	VB6 import	The project type is not an Exe project.	Import an Exe project; a Library or Custom Control plugin is the McBasic™ equivalent of the other types.
MCB5603	VB6 import	The output cannot be written: it must end with `.mcbproj`, must not exist, and must be in an existing folder outside an app or another package.	Choose another name or folder.
MCB5604	VB6 import	A form file is not a VB5 or VB6 text form, or its `Begin` and `BeginProperty` blocks are not closed.	Open the form in VB6 and save it again as a text form (first line `VERSION 5.00`).
MCB5605	VB6 import	The code page given with `--codepage` is not supported.	Use one of the code pages the text lists.
MCB5606	VB6 import	The project lists no form, or only MDI children without an MDIForm, so no form can open first.	Add a form or the MDIForm to the project in VB6.
MCB5609	VB6 import	The imported project was rejected before Check, or still fails Check after the quarantine.	Report the project: the importer wrote something McBasic™ does not accept.

Code	Area	Meaning	What to do
MCB5620	VB6 import	A VB6 control has no McBasic™ equivalent: it became a placeholder Panel, or, if it was non-visual, it was not imported.	Replace it with a McBasic™ component and move the code.
MCB5621	VB6 import	A control became another McBasic™ component.	Check that it behaves as needed.
MCB5622	VB6 import	A property value was dropped or changed during the import.	Set it again in the **Properties** panel if it matters.
MCB5623	VB6 import	Something was adapted: `&` access keys removed, a non-visual control array split into controls, a Timer without `Interval` disabled, `ItemData` set in `Form_Load`, a menu item moved to the application menu.	Read the text; change the code that depended on the old behaviour.
MCB5624	VB6 import	A picture could not be imported, or the project would exceed 256 resources.	Add the image as a project resource and set the property again.
MCB5625	VB6 import	A file was not at its listed path, so a file with the same name in the project folder was used.	Check that it is the right file.
MCB5626	VB6 import	A control or procedure was renamed because its name is not allowed in McBasic™; the imported code uses the new name.	Use the new name in new code too.
MCB5627	VB6 import	A file was not found and was not imported; the code that uses it is quarantined.	Move the file next to the `.vbp` and import again.
MCB5628	VB6 import	A form has more controls or deeper menus than McBasic™ allows, or holds controls VB6 does not create; the extra ones were not imported.	Split the form, or keep menus within 5 levels.
MCB5630	VB6 import	A Windows API `Declare` has no macOS equivalent, or is no longer called because every call became a McBasic™ service.	Replace or remove the call; delete the stub.
MCB5631	VB6 import	A class uses a VB6 feature McBasic™ does not have: a global default instance, a default member, or a `For Each` enumerator.	Add a Public instance in a module, name the member in calls, or

Code	Area	Meaning	What to do
			expose a `Collection`.
MCB5632	VB6 import	Quarantined lines use ADO, DAO or a data control.	Rebuild the data access with SQLite (Chapter 12).
MCB5633	VB6 import	A statement has no McBasic™ equivalent; the text says what to use instead.	Follow the text.
MCB5637	VB6 import	An event handler does not match a McBasic™ event, or VB6 raised an event McBasic™ never raises (such as `Click` on a menu title); the handler was renamed or adapted.	Write the McBasic™ handler from the control's event list and move the code.
MCB5639	VB6 import	Code could not be converted: its body raises error 5, or the procedure or the module-level code is commented out.	Convert the original lines below the comment marks by hand, then remove the marks.
MCB5640	VB6 import	Lines set a position or size from variables or a `ScaleMode` McBasic™ cannot convert.	Rewrite them in points.
MCB5641	VB6 import	An object variable VB6 created implicitly has no known type.	Declare it with `Dim name As <class or control type>`.
MCB5642	VB6 import	A value is passed `ByRef` where McBasic™ cannot pass it by reference; a temporary is used.	Pass a variable, or copy the result back.
MCB5643	VB6 import	Fixed-length strings became ordinary `String`s, or another small change was made or must be made by hand; the text says which.	Follow the text.
MCB5644	VB6 import	`#If` lines were evaluated as VB6 would on Windows, or nest too deep to evaluate.	Check the lines that were left out.
MCB5645	VB6 import	Quarantined lines use a library McBasic™ does not have (for example WinInet).	Use the `Http` service for web requests, or the service the text names.

Code	Area	Meaning	What to do
MCB5650	VB6 import	The startup object changed: no usable startup object, a start from an MDI child, a `Sub Main` that was not `Public`, or a start from `Sub Main` that ends when the last form closes.	Check **Project > Project Settings…** and the first form.
MCB7015	Project	The `packages` list of `project.mcproj` is malformed: not an array, an entry without `id` and `version`, a version that is not three numbers, unsorted ids, or more than 256 references.	Fix the entry the text names; **Project > Add Plugin Reference…** writes the list correctly.
MCB7056	Packages	A package operation needs exact offline inputs: an absolute `.mbpkg` archive, a cache root, a lock file and a target together; plugin certification supports only `osx-arm64`.	Use **Project > Add Plugin Reference…**, which supplies the inputs (Chapter 14).
MCB7058	Packages	An external component's events or visual anchors are malformed: more than 128 bindings, duplicate names, anchors other than Left, Top, Right or Bottom.	Fix the component's layout JSON; the text names the field.
MCB7075	Packages	The trusted publisher keys are invalid, duplicated or more than 256.	Edit the publisher keys file or choose **Developer Mode**.
MCB7080	Packages	A package change needs explicit trust settings, absolute paths and a project hash, and no stale lock file.	Repeat the operation from the IDE panel.
MCB7085	Packages	A package does not support the chosen target, or the installation needs an absolute package store.	Choose a package built for `osx-arm64` or `osx-x64`.
MCB7110	Packages	**Certify Plugin** needs a Library or Custom Control project, or the certification tool is missing from the installation.	Open a plugin project; install McBasic™ again if the tool is missing.
MCB7111	Packages	The certification returned no receipt, an invalid gate matrix, or a package that does not match the certified archive.	Run **Certify Plugin** again; report the message if it repeats.

Glossary

Use this list when a word in the guide is new. The explanations use ordinary language first and technical detail second.

Term	Plain-English meaning
AI assistant	The optional chat helper in the AI chat panel. It proposes answers and changes; McBasic™ checks them before they are applied, and you can undo each one.
Application	The program that the user runs.
Argument	A value that you pass to a procedure or function.
Array	A group of values of the same kind, accessed by position.
Ask me first	A choice of the **Apply AI changes:** setting. Each checked AI stage or change waits on its card for **Apply** or **Discard** instead of being applied automatically.
Budget	The largest request McBasic™ sends to the AI provider, in characters: one part for the prompt and one for the project context. Set by **Request size:**.
Build	The process that checks a project and creates an application result.
Build cache	The folder outside the project, `~/Library/Caches/McBasic/Builds`, where McBasic™ keeps build output.
Caption	The text that a user sees on a control, form, or window.
Card	A framed box in the AI chat that shows a plan, a stage, a change, or an error, with the buttons you can use. It updates in place.
Change card	The AI chat card for a single change outside a plan, such as a new version of one procedure or form. It shows the changes and **Undo**, or **Apply** and **Discard**.
Chat	One conversation with the AI assistant. A project can have several; each keeps its own messages and context chips.
Check	McBasic™'s validation of the complete project. Every AI-generated stage and change must pass Check before it is applied.

Term	Plain-English meaning
Class	A code definition used to create objects with related data and behavior.
Component	A visual control or nonvisual service that you can use in a project.
Context chip	A small label above the AI message field naming a document and what is sent from it (code, form, or both). The chips are your consent for that chat.
Control	A visible object, such as a button, text box, list, or grid, that the user can use.
Conversation	See Chat.
Database	An organized store of data that an application can read and, when allowed, change.
Dependency card	The AI chat card that asks you to approve a plugin reference or a database migration file before a stage is applied.
Diagnostic	A message that explains a problem and points to its location.
Event	Something that happens, such as a click, change, or form closing.
Event handler	A procedure that runs in response to an event.
Failure card	The red AI chat card shown when a stage or change does not pass Check after 3 attempts, with the diagnostics and **Retry stage**, **Skip stage**, and **Stop plan**.
Form	A window or screen that the user sees and uses.
Function	A procedure that returns a value.
IDE	The McBasic™ application where you design, write, check, and run projects.
Inspector	The part of the IDE where you view and change properties.
Instance	A live object created from a form or class definition.
Layout	The saved arrangement of a form and its controls.
Manifest	A structured file that describes project members or package contents.
Member	A property, method, or event that belongs to an object.
Module	A file that holds reusable code. A standard module is not a visual form.
Package	A prepared group of reusable project or component material.
Parameter	A named input declared by a procedure.

Term	Plain-English meaning
Pause	A stop in an AI plan that waits for you. The plan card names the reason and shows the button that continues it, such as **Resume** or **Retry**, and always **Stop plan**. A plan also pauses when you change the AI provider while it runs.
Plan	The AI assistant's proposal for a larger change: the forms, modules, and stages to build. It appears as a plan card; nothing is generated until you choose **Approve and run**.
Procedure	A named block of code that performs a task. A `Sub` and a `Function` are procedures.
Project	The complete set of files and settings that make one McBasic™ application.
Project package	A `.mcbproj` item that Finder shows as one file; it holds the project file, source, layouts, resources, and optional saved AI chats.
Property	A value that describes an object, such as its Name, Caption, or Width.
Provider	The AI service the assistant uses: External Codex, a Responses API endpoint, or LM Studio on this Mac.
Publish	The process of creating an application output that can be copied or shared.
Reference index	McBasic™'s own summary of its components, forms, functions, and language, sent with every AI request. It contains no project data.
Restart (kick-in)	What McBasic™ does when the project changes while an AI stage runs: it drops the stage's proposal and generates the stage again on the current version, at most twice in a row.
Running state	The part of the application that is active while the app is open.
Scope	The part of a project where a name can be used.
Source code	The human-readable McBasic™ instructions that you write.
Stage	One step of an approved plan. A stage is generated, checked, and applied automatically as one Undo action, unless you chose **Ask me first**.
Standard module	A `.bas` code file used for shared procedures and values.
Transaction	A group of database changes treated as one unit.
Variable	A named place that holds a value while the program runs.
Visual designer	The part of the IDE where you place controls and set layout properties.

A useful distinction

Some words describe what the user sees. Others describe what the program does.

```
Caption  -> text shown to the user
Name     -> identifier used by your code
Event    -> something that happens
Handler  -> code that responds to it
```

For example, a button may show **Save customer** as its Caption, have `cmdSave` as its Name, raise a Click event, and run `cmdSave_Click` as its event handler.

Check your understanding

When a new term appears, say it in your own words before continuing. If you cannot, look for it here and then return to the example that used it. The example is usually more useful than memorizing the definition alone.

Tips and tricks

- **Return to the example after you read a definition.** The example is usually more useful than the definition.
- **A control array is many controls with one name.** Each member has an `Index`.
- **A project is a package.** The `.mcbproj` is one item in Finder.

Workshop 1 - Build a small app from an empty form

The first chapters used a supplied sample. Here you start with a blank form and build a small appointment card. It does not need a database, so you can see the design, naming, event, Check, and test steps without extra setup.

What you will build

The user will enter a name, a subject, and a note. When the user chooses **Preview**, the form will show a short confirmation. **Clear** will remove the input and return focus to the first field.

The finished form has this small contract:

Control	Name	Job
TextBox	`txtName`	enter the person's name
TextBox	`txtSubject`	enter the appointment subject
TextBox	`txtNotes`	enter optional notes
Label	`lblStatus`	show a result or a correction message
CommandButton	`cmdPreview`	validate and show a preview
CommandButton	`cmdClear`	clear the form

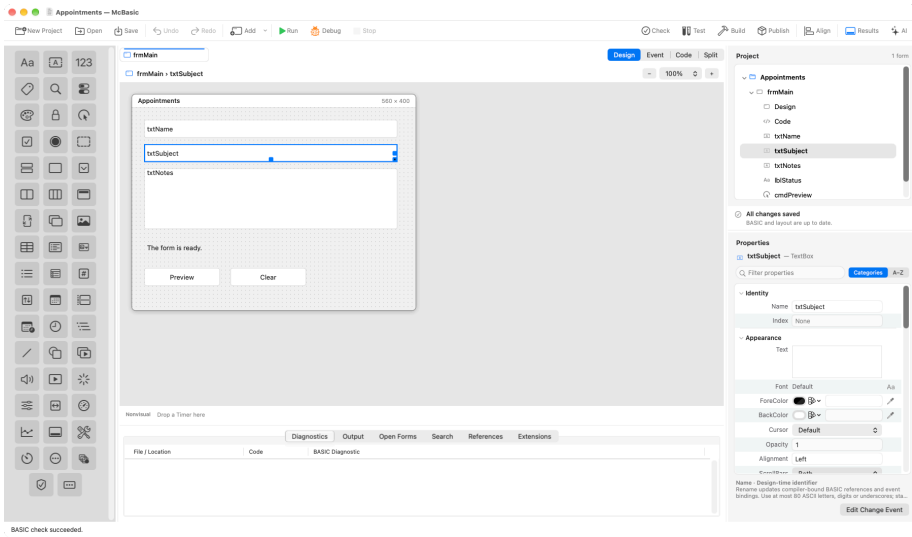


Figure W1.1: A diagnostic linked to source code

Before you start

Create a copy of a blank form project or use the new-project workflow for the release you are documenting. The exact menu name may change. The important result is an editable project with one form and `Option Explicit` in its code.

Do not add a database or a second form yet. A smaller project makes each test easy to understand.

Step 1: Design the form

Place the controls in this order:

1. Add a TextBox and name it `txtName`.
2. Add a TextBox and name it `txtSubject`.
3. Add a multiline TextBox and name it `txtNotes`.
4. Add a Label and name it `lblStatus`.
5. Add two CommandButton controls and name them `cmdPreview` and `cmdClear`.
6. Set the button Captions to **Preview** and **Clear**.
7. Set the TabIndex values so the user reaches the name, subject, notes,

Preview, and Clear controls in a sensible order.

The Caption is for the user. The Name is for your code. Do not use the visible word `Preview` as the code name unless that is the naming convention you have chosen for the whole project.

Checkpoint: design only

Save and choose **Check** before writing the event code.

Expected result: the project checks successfully. The form contains six controls, each has one clear purpose, and Tab moves through the inputs in the same order as the task.

If Check reports a duplicate name, select the control named in the diagnostic and give it a unique Name. Do not fix a duplicate by changing only its Caption.

Step 2: Add the Preview event

Open the Click event for `cmdPreview` and write this small version first:

```
Private Sub cmdPreview_Click()
    Dim name As String
    Dim subject As String

    name = Trim$(txtName.Text)
    subject = Trim$(txtSubject.Text)

    If Len(name) = 0 Then
        lblStatus.Caption = "Enter a name."
        txtName.SetFocus
        Exit Sub
    End If

    If Len(subject) = 0 Then
        lblStatus.Caption = "Enter a subject."
        txtSubject.SetFocus
        Exit Sub
    End If

    lblStatus.Caption = "Ready: " & subject & " for " & name
End Sub
```

Read the procedure in four parts:

1. it reads and trims the two required inputs;
2. it stops at the first missing value;
3. it puts focus on the field that needs attention;
4. it shows a result only when both values are valid.

The procedure does not save anything. The word **Ready** is honest: this first workshop proves the form and validation path, not a storage system.

Step 3: Add the Clear event

Open the Click event for `cmdClear`:

```
Private Sub cmdClear_Click()
    txtName.Text = ""
    txtSubject.Text = ""
    txtNotes.Text = ""
    lblStatus.Caption = "The form is ready."
```

```
    txtName.SetFocus
End Sub
```

Choose **Check** again. If the diagnostic names a control, compare the spelling with the Name shown in Design view. The code must use `txtName`, not the Caption shown above the control.

Step 4: Test the normal path

Run the app and test one ordinary appointment:

1. type `Ada` in the name field;
2. type `Project review` in the subject field;
3. type `Bring the sample notes` in the notes field;
4. choose **Preview**.

Expected result: the status label says:

```
Ready: Project review for Ada
```

The notes are not shown yet. That is acceptable. The workshop is intentionally small, and the next challenge will decide how to present longer notes.

Step 5: Test the correction paths

Test each case separately. Clear the form between cases.

Input	Action	Expected result
both required fields empty	choose Preview	status asks for a name; focus moves to `txtName`
name filled, subject empty	choose Preview	status asks for a subject; focus moves to `txtSubject`
name contains spaces around `Ada`	choose Preview	result uses `Ada`, without the extra spaces
subject contains spaces around the text	choose Preview	result uses the trimmed subject
all fields filled	choose Clear	all fields are empty and name has focus

This table is more useful than a single successful click. A desktop app is a conversation with the user, and the empty cases are part of that conversation.

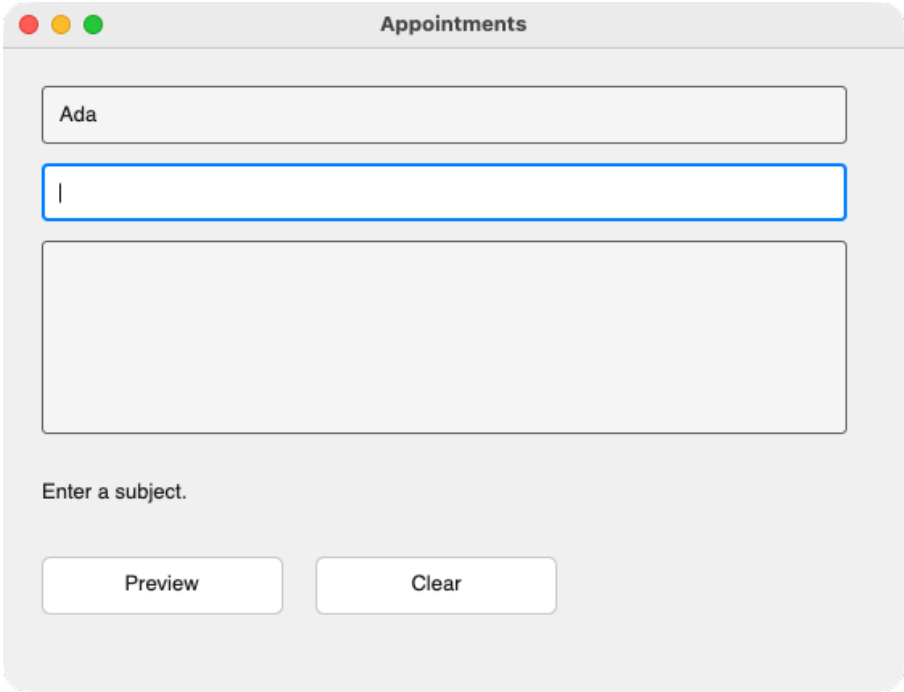


Figure W1.2: Workshop validation result

Step 6: Make the result easier to read

Now improve one part at a time. For example, show the notes in the status label only after the required values have passed validation:

```
Dim notes As String
notes = Trim$(txtNotes.Text)

If Len(notes) = 0 Then
    lblStatus.Caption = "Ready: " & subject & " for " & name
Else
    lblStatus.Caption = "Ready: " & subject & " for " & _
        name & " - " & notes
End If
```

Test a short note and a long note. If the label is not suitable for long text, use a separate multiline display control in the next design revision. Do not make a status label responsible for every kind of content.

After the workshop

You used the same cycle that larger projects use:

```
design -> name -> write one event -> check -> run -> test -> improve
```

The project has no hidden magic. The form is the layout, the controls expose members, and the event procedures decide what happens after user actions.

Check your work

The workshop is complete when:

- the form opens without a diagnostic;
- Preview rejects each missing required value;
- valid input produces the expected status message;
- Clear removes all three input values;
- Tab order follows the task;
- you can explain which code line changes the visible result.

Next challenge

Add a `CheckBox` named `chkConfirmed` with Caption **Confirmed**. Do not change the success message yet. First decide what the unchecked state should mean and write the rule in one sentence. Then add the smallest validation branch that implements that rule.

The habit matters: decide the user rule before writing the property or event code.

Tips and tricks

- **Finish the design before the code.** Add the controls and names, then add the events.
- **Test the normal path first.** Then test the correction paths.
- **Use Undo while you experiment. Edit > Undo** steps back one change at a time.
- **Name controls as you add them.** You will need the names in code.

Component cookbook

The component reference tells you what exists. This cookbook starts with the job you want to do and then helps you choose. For each recipe, ask:

1. What job does the component do?
2. What is a good first use?
3. What should you test?
4. What common mistake should you avoid?

The component names in this chapter come from the standard component catalog. The exact properties and events can change between releases, so use the Inspector and the component help for the complete contract.

Start with the user's job

Choose a component because of the user's task, not because it looks familiar.

User needs to...	Start with...
enter short text	TextBox
enter hidden text	PasswordBox
choose one action	CommandButton
read a message	Label or StatusBar
choose one value from a list	ComboBox or ListBox
choose yes or no	CheckBox or ToggleSwitch
choose one item from a small group	OptionButton
find a value	SearchBox
choose a date or time	DatePicker or TimePicker
edit numbers	NumericBox
see rows of data	DataGrid
show progress	ProgressBar, Gauge, or Sparkline

User needs to...	Start with...
organize controls	Panel, Frame, TabControl, or SplitContainer
show a document or picture	Image
offer commands outside the form	Menu, ToolBar, or ContextMenu

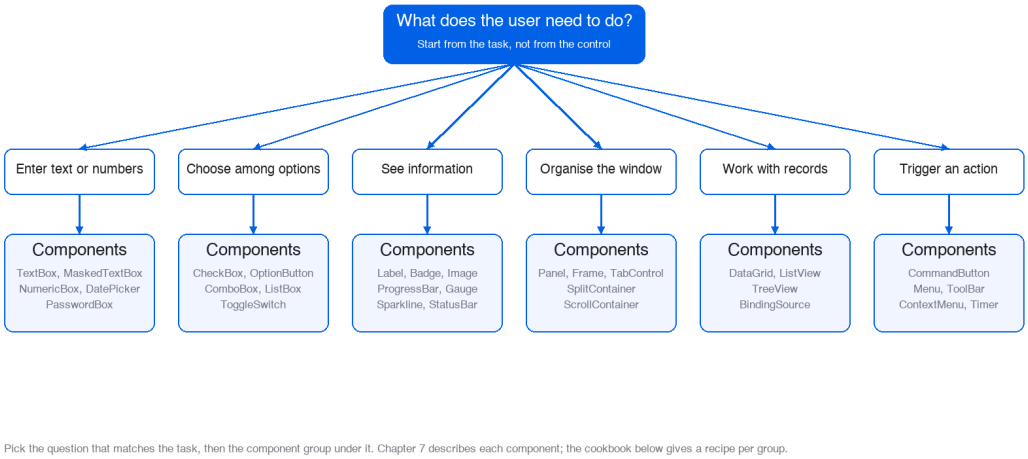


Figure 26.1: Choosing a component

Text input

TextBox

Use a TextBox for ordinary text such as a name, title, code, or short note. Give it a meaningful Name, set a useful Placeholder when appropriate, and use `Required` or explicit validation when empty input is not allowed.

First test: type normal text, delete it, type spaces only, and paste a long value. Decide what the user should see in each case.

Common mistake: reading the Caption instead of the Text value. A TextBox usually displays its value through `Text`.

```
Private Sub txtName_Change()
    lblStatus.Caption = "Characters: " & CStr(Len(txtName.Text))
End Sub
```

PasswordBox

Use PasswordBox for a secret that should not be displayed as ordinary text. Keep the secret out of status labels, debug output, and error messages.

First test: enter text and confirm that the display is masked. Test Clear, focus, and the error path without printing the value.

Common mistake: treating masking as encryption. A masked input is still sensitive data. Use the product's supported secure storage or service boundary when the value must be saved.

MaskedTextBox

Use MaskedTextBox when the shape of the input matters, such as a phone number or a short code. A mask can guide a user, but it is not a complete business validation rule.

First test: type a complete value, an incomplete value, and paste text that does not match the expected shape.

Common mistake: storing mask characters as if they were part of the user's real value. Decide whether your data layer wants the formatted value or a cleaned value.

Commands and feedback

CommandButton

Use CommandButton for an action the user can name: Save, Search, Preview, or Cancel. Its Click event should have one clear job.

First test: activate it with the mouse and keyboard. Test Enabled while a long operation is in progress.

Common mistake: letting one Click handler perform unrelated operations. A short handler is easier to test and easier to explain.

Label

Use Label for text that explains a field or reports a short result. A label is not an input and should not be the only place where important data can be edited.

First test: show a normal message, an error message, and a message with accented text.

Common mistake: using one label for instructions, warnings, progress, and final results at the same time. Give those messages clear roles.

Badge

Use Badge for a compact status such as **New**, **Pending**, or **Offline**. Keep its text short and make sure the meaning does not depend on color alone.

First test: view each status with a keyboard, in a light theme, and in a dark theme if the release supports both.

Common mistake: putting a full sentence into a component designed for a short marker.

StatusBar

Use StatusBar for information that should remain at the bottom of a window, such as connection state or a short operation result.

First test: resize the window and confirm that the message remains visible.

Common mistake: hiding an important error only in the status bar. Repeat important errors near the control that needs correction.

Choices

CheckBox

Use CheckBox when each option can be independently on or off. Examples include **Include inactive customers** or **Remember this choice**.

First test: read the value when checked and unchecked. Test the initial value and the Click event.

Common mistake: using several unrelated CheckBox controls when exactly one choice is allowed. Use OptionButton for one choice from a group.

OptionButton

Use OptionButton for mutually exclusive choices such as **Card**, **Cash**, or **Invoice**. Put related options in a clear group.

First test: choose each option and confirm that the previous option is no longer selected.

Common mistake: failing to explain which option is selected by default. The user should not have to guess what will happen if they choose Continue.

ToggleSwitch

Use ToggleSwitch for a setting whose state is meaningful immediately, such as **Sync enabled** or **Show archived items**.

First test: change the state with the pointer and keyboard. Check that the label or nearby text makes the current state clear.

Common mistake: using a switch for an action that should happen only after the user presses Save. A CheckBox may communicate that pending choice better.

ComboBox

Use ComboBox when the user chooses one value from a list and the list should not occupy the whole form.

First test: open the list, choose an item, clear or change the selection, and test an empty selection if it is allowed.

Common mistake: displaying a friendly caption but saving the wrong internal value. Keep the display value and the stable stored value separate.

ListBox

Use ListBox when several choices should remain visible or when selecting more than one item is part of the task.

First test: select the first, last, and no item. If multiple selection is used, verify the count and the order of the selected values.

Common mistake: using the visible row position as a database ID. Keep a stable identity with each data row.

Search and numeric or time input

SearchBox

Use SearchBox for a search action, not as a generic TextBox with a magnifying glass icon. Explain what is searched and when the search starts.

First test: search for an exact value, a partial value, no value, and a value with different letter case.

Common mistake: performing a slow search on every keystroke without a clear progress or cancellation rule.

NumericBox

Use NumericBox when the input must be a number with a known range, such as a quantity or percentage.

First test: enter the minimum, maximum, an empty value, and an invalid value. Confirm the message and the stored type.

Common mistake: treating formatted text as a number without checking the release's conversion rules.

DatePicker and TimePicker

Use DatePicker for a calendar date and TimePicker for a time. Display the chosen value in the user's locale, but store it in the data format your app uses consistently.

First test: choose the first and last allowed values, cancel a selection, and reopen the control.

Common mistake: comparing localized date strings instead of comparing date values.

Display and data

DataGrid

Use DataGrid for rows and columns that the user must scan, select, or edit. Give the grid a clear empty state and show which row is selected.

First test: no rows, one row, many rows, sorting, selection, and invalid editing. Save only after the user's edit passes validation.

Common mistake: assuming the displayed row number is the record identity.

Image

Use Image for a logo, illustration, or result that is easier to understand visually than as text. Give important images useful accessible text.

First test: missing image, large image, small image, and a path that is valid in the IDE but not in the published app.

Common mistake: storing a development-machine path in a project that will run on another Mac.

ProgressBar, Gauge, and Sparkline

Use ProgressBar for completion of a known task, Gauge for a current measured value, and Sparkline for a compact trend.

First test: zero, middle, maximum, and an unknown or cancelled operation. Pair the visual with text when the exact value matters.

Common mistake: showing movement that does not represent real progress. Indeterminate work should be labelled as waiting, not given a fake percentage.

Layout containers

Panel and Frame

Use Panel to group controls and Frame when a bordered group or visual boundary helps the user understand the form. Select the intended parent before adding a child control.

First test: move or resize the parent and confirm that the child controls remain in the expected group.

Common mistake: placing a control on the form when it should belong to the selected container.

TabControl and TabPage

Use TabControl when several related groups do not need to be visible at once. Use clear TabPage names and keep the most important task on the first page.

First test: switch pages with the pointer and keyboard and return to the page after a validation error.

Common mistake: hiding a required field on a page the user does not know exists.

Expander

Use Expander for optional detail that would make the normal form too busy. Put a useful summary in the collapsed header.

First test: expand and collapse it while preserving the entered values.

Common mistake: putting the only Save or Continue action inside a collapsed area without telling the user where it is.

ScrollContainer and SplitContainer

Use ScrollContainer for content that may be taller than the window. Use SplitContainer when the user benefits from two resizable areas, such as a list on the left and details on the right.

First test: use a small window, resize the divider, and move focus through both areas with the keyboard.

Common mistake: using scrolling to hide a form that should instead be redesigned into smaller tasks.

Menus and nonvisual helpers

Menu, ToolBar, and ContextMenu

Use Menu for application-wide commands, ToolBar for frequently used visible commands, and ContextMenu for commands related to one selected object.

First test: activate each command with the pointer and keyboard. Disable a command when it cannot be used, and explain why in nearby status text when the reason is not obvious.

Common mistake: placing destructive commands next to common commands with no confirmation or undo path.

Timer

Use Timer for a repeated, bounded action such as refreshing a clock display. Keep the interval and the work small because timer events run on the UI thread.

First test: enable it, observe one or two ticks, disable it, and unload the form. Confirm that it stops when the form is gone.

Common mistake: using a timer to hide a slow database or network design. Use an asynchronous operation for work that can take time.

ValidationProvider

Use ValidationProvider when several controls need a consistent way to show validation messages. Keep the message close to the field it explains.

First test: show one error, correct it, and confirm that the old message is cleared. Test two errors at the same time.

Common mistake: marking a field invalid without telling the user how to make it valid.

Check your component choice

Before adding a component, write one sentence:

> *The user uses this component to _____.*

Then write one test:

> *I will know it works when _____.*

If you cannot complete either sentence, the design decision is not ready. Pick the smallest component that supports the task and add complexity only after a working test exists.

The 51-control recipe cards

The recipes above teach the most common combinations. The cards in this section make the chapter a reference as well: every standard control has a small, usable starting point. They are intentionally written in the same order you use in McBasic™:

1. choose the control for a job;
2. place it inside the correct parent in the Designer;
3. give it a meaningful Name;
4. set the important properties;
5. add one event procedure;
6. run Check, then test the normal case and the awkward case.

The code in a card is not a disconnected demo. Add it to the current project, then keep the working version in the matching sample folder. For the larger progressive example, use WeatherDesk: the same project grows from form layout to SQLite data, two external API calls, JSON, search, validation, and export.

TextBox

Use it for: editable text that is visible while the user types.

Designer recipe: add a TextBox, name it `txtSearch`, set `Placeholder` to `Search books`, and set `MaxLength` when the database has a fixed limit.

Key members: `Text`, `Placeholder`, `ReadOnly`, `MultiLine`, `Required`, `MaxLength`; use `Change`, `SetFocus`, and `SelectAll`.

```
Private Sub txtSearch_Change()
    lblStatus.Caption = "Characters: " & CStr(Len(txtSearch.Text))
End Sub
```

Expected result: the status text changes as the user types. Test an empty value, spaces, pasted text, and the maximum length.

Watch for: read `Text`, not `Caption`. Trim or validate the value at the boundary where it enters your data or API layer.

Repository starting point: WeatherDesk.

PasswordBox

Use it for: a password or other secret that must be masked on screen.

Designer recipe: add a PasswordBox, name it `txtPassword`, and keep the secret out of labels, logs, and error messages.

Key members: the input `Text`, focus and keyboard events, and the control's normal enabled/read-only state.

```
Private Sub cmdSignIn_Click()
    If Len(txtPassword.Text) = 0 Then
        lblStatus.Caption = "Enter your password."
        txtPassword.SetFocus
    Else
        lblStatus.Caption = "Ready to sign in."
    End If
End Sub
```

Expected result: an empty password is caught without displaying the secret.

Watch for: masking is not encryption. Do not save or print the value unless the application has a deliberate secure-storage design.

Repository starting point: PasswordBox.

TabControl

Use it for: several related areas that do not need to be visible at once.

Designer recipe: add a TabControl, add named TabPage children, and put the most common task on the first page.

Key members: `SelectedIndex`, `SelectedTab`, `SelectionChanged`, and the shared layout, visibility, and accessibility properties.

```
Private Sub tabMain_SelectionChanged()
    lblStatus.Caption = "Page: " & tabMain.SelectedTab.Caption
End Sub
```

Expected result: changing pages updates the status text. Test returning to a page after a validation error so entered values are not lost.

Watch for: a required field hidden on another page still needs a clear message telling the user where to fix it.

Repository starting point: TabControl.

TabPage

Use it for: one named page inside a TabControl.

Designer recipe: select the TabControl before adding the page, set a short caption, then place only related controls inside it.

Key members: the page caption/name, visibility and enabled state, and its child controls. The parent TabControl owns page selection.

```
Private Sub cmdShowDetails_Click()
    tabMain.SelectedIndex = 1
    txtDetails.SetFocus
End Sub
```

Expected result: the command moves the user to the details page and places the cursor in the first useful field.

Watch for: do not use pages as unrelated storage boxes. If the user cannot guess what belongs on a page, split the task differently.

Repository starting point: TabControl.

CommandButton

Use it for: one named action such as Save, Search, Preview, or Cancel.

Designer recipe: set `Caption`, choose `Default` or `Cancel` only when the keyboard meaning is correct, and give the button a clear Name.

Key members: `Caption`, `Enabled`, `Default`, `Cancel`, `Click`, and focus methods.

```
Private Sub cmdSearch_Click()
    If Len(Trim(txtSearch.Text)) = 0 Then
        lblStatus.Caption = "Enter a search term."
        txtSearch.SetFocus
        Exit Sub
    End If
    lblStatus.Caption = "Searching for " & txtSearch.Text & "..."
End Sub
```

Expected result: the button gives immediate feedback and refuses an empty search.

Watch for: one Click handler should have one clear job. Disable the button while a slow operation is genuinely in progress.

Repository starting point: WeatherDesk.

Label

Use it for: instructions, field captions, and short results.

Designer recipe: put a Label beside the control it explains and set its `Caption` to a useful sentence, not only a decorative title.

Key members: `Caption`, visibility, enabled state, alignment, wrapping, and accessible text.

```
Private Sub txtTitle_Change()
    lblTitleCount.Caption = CStr(Len(txtTitle.Text)) & " characters"
End Sub
```

Expected result: the label explains the current state without becoming an input field.

Watch for: do not use one label for instructions, warnings, progress, and final results at the same time. Give each message a role.

Repository starting point: WeatherDesk.

ValidationProvider

Use it for: a consistent error message attached to another control.

Designer recipe: add the provider, set `Target` to the control Name, and leave the provider beside the field it describes.

Key members: `Target`, `ErrorText`, `IsValid`, `ErrorCount`, and `FirstError` when reading the validation result.

```
Private Sub cmdCheck_Click()
    valTitle.Target = "txtTitle"
    If Len(Trim(txtTitle.Text)) = 0 Then
        valTitle.ErrorText = "Enter a title."
    Else
        valTitle.ErrorText = ""
    End If
    lblStatus.Caption = valTitle.FirstError
End Sub
```

Expected result: the empty field receives a useful message, and the message disappears after the value is corrected.

Watch for: “invalid” is not an instruction. Tell the user how to fix the value.

Repository starting point: ColorPicker validation sample.

Timer

Use it for: a small repeated action, such as refreshing a clock or polling a short-lived UI state.

Designer recipe: add a Timer, set a moderate `Interval`, and leave it disabled until the form has loaded.

Key members: `Interval`, `Enabled`, and the `Timer` event.

```
Private Sub Form_Load()
    tmrClock.Interval = 1000
    tmrClock.Enabled = True
```

```
End Sub

Private Sub tmrClock_Timer()
    lblClock.Caption = CStr(Now())
End Sub
```

Expected result: the label changes about once per second and stops when the timer is disabled.

Watch for: a timer is not a replacement for an asynchronous database or network operation. Keep its event short.

Repository starting point: BuiltInAcceptance.

Menu

Use it for: application-wide commands that should remain discoverable.

Designer recipe: group commands by user goal, keep labels consistent with the buttons, and put destructive actions away from common actions.

Key members: the menu items collection, item captions and enabled state, and the menu `Click` event.

```
Private Sub mnuFileSave_Click()
    SaveCurrentRecord
    staStatus.Text = "Saved"
End Sub
```

Expected result: the same Save action is available from the menu and from the form's main button.

Watch for: a menu item that is unavailable should be disabled, not silently ignored.

Repository starting point: Menu.

CheckBox

Use it for: an independent yes/no option.

Designer recipe: write the option as a statement, such as “Include archived items”, so the checked state has an obvious meaning.

Key members: `Value`, `Checked`, `Click`, and the normal enabled state.

```
Private Sub chkArchived_Click()
    gridBooks.FilterText = IIf(chkArchived.Value, "", "active")
End Sub
```

Expected result: turning the option on and off changes the filter without affecting unrelated choices.

Watch for: use OptionButton when exactly one option in a group is allowed.

Repository starting point: WeatherDesk.

Panel

Use it for: a group of controls that should move, resize, or hide together.

Designer recipe: select the Panel before adding children; set its layout and border only after the group works.

Key members: common layout properties plus `BorderStyle`, `BorderColor`, `BorderWidth`, `CornerRadius`, `ClipToBounds`, and `Resize`.

```
Private Sub pnlDetails_Resize()
    lblPanelSize.Caption = CStr(pnlDetails.Width) & " x " &
CStr(pnlDetails.Height)
End Sub
```

Expected result: the group responds as one visual region when resized.

Watch for: a child dropped on the form instead of the Panel will not follow the group when it moves.

Repository starting point: PanelGallery.

FlowLayoutPanel

Use it for: a row or wrapped group of controls whose positions should adapt to available width.

Designer recipe: set `FlowDirection`, `WrapContents`, `Spacing`, and `CrossAlignment`, then add children in their intended reading order.

Key members: `FlowDirection`, `WrapContents`, `Spacing`, `CrossAlignment`, and the normal container properties.

```
Private Sub Form_Load()
    flpTags.FlowDirection = "LeftToRight"
    flpTags.WrapContents = True
    flpTags.Spacing = 8
    flpTags.CrossAlignment = "Center"
End Sub
```

Expected result: tags or buttons wrap instead of overlapping when the form becomes narrow.

Watch for: a flow panel is not a substitute for a clear form hierarchy. Keep the order meaningful when the layout wraps.

Repository starting point: FlowLayoutPanel.

Frame

Use it for: a visually bounded group, often with a title or border.

Designer recipe: add the Frame first, set its caption/border, and then add the controls that belong to that group inside it.

Key members: common layout and appearance properties, border settings, visibility, enabled state, and child controls.

```
Private Sub fraFilters_Resize()
    lblStatus.Caption = "Filter area: " & CStr(fraFilters.Width) & " px
wide"
End Sub
```

Expected result: the boundary makes the relationship between the fields easy to scan.

Watch for: do not add a border merely for decoration; it should explain a real grouping.

Repository starting point: PanelGallery.

Expander

Use it for: optional detail that would make the normal view too busy.

Designer recipe: give the header a useful summary and keep the primary action outside the collapsed area.

Key members: `Header`, `IsExpanded`, `ExpandDirection`, `Expanded`, and `Collapsed`.

```
Private Sub expAdvanced_Expanded()
    lblStatus.Caption = "Advanced options are visible"
End Sub

Private Sub expAdvanced_Collapsed()
    lblStatus.Caption = "Advanced options are hidden"
End Sub
```

Expected result: expanding reveals optional settings and collapsing keeps their values.

Watch for: never hide the only Save or Continue action without telling the user where it went.

Repository starting point: Expander.

OptionButton

Use it for: one choice from a small mutually exclusive group.

Designer recipe: place related OptionButton controls in one group, set one clear default, and use complete labels such as “Monthly” and “Yearly”.

Key members: selected value/state, grouping, `Click`, and keyboard focus.

```
Private Sub optMonthly_Click()
    lblPlan.Caption = "Plan: monthly"
End Sub

Private Sub optYearly_Click()
    lblPlan.Caption = "Plan: yearly"
End Sub
```

Expected result: selecting one option makes the old selection clear.

Watch for: if the user must select several independent options, use CheckBox instead.

Repository starting point: CompatibilityControls.

DataGrid

Use it for: rows and columns that users scan, select, sort, or edit.

Designer recipe: define stable columns, set `EmptyText`, keep the primary key in `KeyColumn`, and decide explicitly whether the grid is read-only.

Key members: `Columns`, `KeyColumn`, `FilterColumn`, `FilterText`, `Loading`, `EmptyText`, `ErrorText`, `IsReadOnly`, sorting options, `SelectionChanged`, `Sorting`, `Sorted`, and `DataError`.

```
Private Sub grdBooks_SelectionChanged()
    If grdBooks.SelectedIndex < 0 Then
        lblStatus.Caption = "Select a book."
    Else
        lblStatus.Caption = "Selected row: " &
CStr(grdBooks.SelectedIndex)
    End If
End Sub
```

Expected result: the user can select a row and understand whether the grid is loading, empty, or showing an error.

Watch for: the displayed row number is not a record identity. Bind a stable key and use it when loading the detail form.

Repository starting point: WeatherDesk.

ComboBox

Use it for: one value chosen from a list that should remain compact.

Designer recipe: set `Items` or bind data, keep `DisplayMember` separate from `ValueMember`, and choose the right `DropDownStyle`.

Key members: `Text`, `Items`, `DisplayMember`, `ValueMember`, `Sorted`, `DropDownStyle`, `SelectedIndex`, `Change`, `DropDown`, and `DropDownClosed`.

```
Private Sub cboCategory_Change()
    lblStatus.Caption = "Category: " & cboCategory.Text
End Sub
```

Expected result: the visible caption is friendly while the saved value can remain a stable ID.

Watch for: never save a caption as a database key when the caption can be renamed or translated.

Repository starting point: WeatherDesk.

ListBox

Use it for: a visible list, including a task that may allow multiple selection.

Designer recipe: decide `SelectionMode`, set `DisplayMember` and `ValueMember` for bound data, and define an empty-list message in nearby text.

Key members: `Items`, `DisplayMember`, `ValueMember`, `SelectionMode`, `Sorted`, `SelectedIndex`, and `SelectionChanged`.

```
Private Sub lstAuthors_SelectionChanged()
    If lstAuthors.SelectedIndex >= 0 Then
        lblStatus.Caption = "Author selected: " & lstAuthors.Text
    End If
End Sub
```

Expected result: a selection is visible and can be translated to a stable value before a query or save.

Watch for: do not use a visible row position as the record ID.

Repository starting point: CompatibilityControls.

ContextMenu

Use it for: commands that belong to the object under the pointer or the current selection.

Designer recipe: attach it to the relevant control and keep its commands short, specific, and safe to repeat.

Key members: item captions, enabled state, target control, and `Click`.

```
Private Sub ctxBooks_Open_Click()
    OpenSelectedBook
End Sub

Private Sub ctxBooks_Copy_Click()
    CopySelectedBookId
End Sub
```

Expected result: the menu acts on the selected book, not on whichever row was selected earlier.

Watch for: a context menu should not be the only way to discover an important command.

Repository starting point: ContextMenu.

Badge

Use it for: a compact status marker such as New, Pending, or Offline.

Designer recipe: keep the text short, and make the meaning clear without depending on color alone.

Key members: its displayed text, visual state, visibility, and accessible description.

```
Private Sub SetSyncState(ByVal IsOnline As Boolean)
    If IsOnline Then
        bdgSync.Caption = "Online"
    Else
        bdgSync.Caption = "Offline"
    End If
End Sub
```

Expected result: the status is readable in text and visually noticeable.

Watch for: do not put a paragraph into a component designed for a marker.

Repository starting point: Badge.

SearchBox

Use it for: a deliberate search field, especially when the search action has a clear scope.

Designer recipe: explain what is searched, set a useful placeholder, and choose whether search starts on Change or on a button/Enter key.

Key members: `Text`, placeholder, `Change`, focus, keyboard events, and the associated result status.

```
Private Sub schBooks_Change()
    If Len(Trim(schBooks.Text)) >= 3 Then
        lblStatus.Caption = "Ready to search: " & schBooks.Text
    Else
        lblStatus.Caption = "Type at least three characters."
    End If
End Sub
```

Expected result: short or empty input does not start an expensive search.

Watch for: do not run a slow API request on every keystroke without a progress and cancellation plan.

Repository starting point: SearchBox.

ToggleSwitch

Use it for: a setting whose state has an immediate, persistent meaning.

Designer recipe: write a clear label such as “Sync enabled” and make the current state understandable without color.

Key members: checked/value state, `Click`, enabled state, and accessible text.

```
Private Sub swSync_Click()
    lblStatus.Caption = IIf(swSync.Value, "Sync is on", "Sync is off")
End Sub
```

Expected result: the nearby message confirms the state after either mouse or keyboard activation.

Watch for: use a CheckBox when the choice is only saved later rather than applied immediately.

Repository starting point: ToggleSwitch.

ColorPicker

Use it for: choosing a color while still keeping the stored value explicit.

Designer recipe: provide a preview or text value, and choose a stable color format such as a hex string for storage.

Key members: `Value`, `Change`, and validation state.

```
Private Sub clrAccent_Change()
    lblColor.Caption = "Selected: " & clrAccent.Value
    pnlPreview.BackColor = clrAccent.Value
End Sub
```

Expected result: changing the picker updates both the readable value and a small preview.

Watch for: validate the color before writing it to a file, API, or theme.

Repository starting point: ColorPicker.

Image

Use it for: a logo, illustration, preview, or result that is clearer as a picture than as text.

Designer recipe: use a project resource or a path that will exist after publishing; add useful accessible text for meaningful images.

Key members: source, stretch/size behavior, visibility, and accessible description.

```
Private Sub Form_Load()
    imgLogo.Source = Resources.Logo
    lblStatus.Caption = "Logo loaded"
End Sub
```

Expected result: the image loads from a project resource on another machine.

Watch for: a path that works on the development computer may not exist in the published app.

Repository starting point: ImageList sample.

ProgressBar

Use it for: completion of a task with a known range.

Designer recipe: set a meaningful minimum and maximum, show text when the exact number matters, and provide a cancelled or failed state.

Key members: value/range, visibility, and the nearby status text.

```
Private Sub UpdateProgress(ByVal Done As Integer, ByVal Total As Integer)
    If Total <= 0 Then Exit Sub
    prgImport.Value = CInt((Done * 100) / Total)
    lblStatus.Caption = CStr(Done) & " of " & CStr(Total)
End Sub
```

Expected result: zero, middle, and complete values all communicate the same progress in a useful way.

Watch for: do not show a fake percentage for work whose length is unknown.

Repository starting point: ProgressBar.

Slider

Use it for: choosing a value along a continuous range, such as volume, zoom, or a threshold.

Designer recipe: label the minimum and maximum meaning and show the current value in text when precision matters.

Key members: minimum/maximum/value, `Change`, keyboard focus, and enabled state.

```
Private Sub sldVolume_Change()
    audTone.Volume = sldVolume.Value
    lblVolume.Caption = "Volume: " & CStr(sldVolume.Value)
End Sub
```

Expected result: moving the slider changes the setting and the displayed number at the same time.

Watch for: do not make a slider the only way to enter an exact value when the exact value is important.

Repository starting point: CompatibilityControls.

NumericBox

Use it for: a number with a known range or step.

Designer recipe: set the range and step, show the unit beside the control, and decide whether empty input is allowed.

Key members: numeric `Value`, range/step, change event, and validation.

```
Private Sub numCopies_Change()
    If numCopies.Value < 1 Or numCopies.Value > 99 Then
        lblStatus.Caption = "Copies must be from 1 to 99."
    Else
        lblStatus.Caption = "Copies: " & CStr(numCopies.Value)
    End If
End Sub
```

Expected result: an out-of-range number is explained before it reaches the database or an API.

Watch for: do not convert formatted display text blindly; use the numeric value exposed by the control.

Repository starting point: NumericBox.

DatePicker

Use it for: a calendar date, such as a due date or publication date.

Designer recipe: state whether the date is local, UTC, or just a calendar day, then set the allowed range.

Key members: selected date/value, change event, minimum/maximum rules, and keyboard focus.

```
Private Sub dtDue_Change()
    lblStatus.Caption = "Due: " & CStr(dtDue.Value)
End Sub
```

Expected result: selecting a date updates the form without comparing localized date strings.

Watch for: store a date value with a defined convention, not whatever text happens to be visible in the current locale.

Repository starting point: CompatibilityControls.

TreeView

Use it for: hierarchical data such as folders, categories, or departments.

Designer recipe: give each node a readable text and a stable tag/key, then decide whether child nodes are loaded immediately or on expand.

Key members: `Items`, `SelectedNode`, `ExpandAll`, `Change`, `ExpandRequested`, `BeforeSelect`, `AfterSelect`, `BeforeExpand`, and `AfterExpand`.

```
Private Sub tvwCategories_Change()
    If tvwCategories.SelectedNode Is Nothing Then
        lblStatus.Caption = "Choose a category."
    Else
        lblStatus.Caption = "Category: " &
tvwCategories.SelectedNode.Text
    End If
End Sub
```

Expected result: the selected node is clear and can drive a filtered view.

Watch for: lazy loading must show a loading/failure state; an empty branch should not look like a broken tree.

Repository starting point: TreeView.

StatusBar

Use it for: a short, persistent state message at the bottom of a window.

Designer recipe: reserve it for connection state, selection summaries, or operation results; repeat important errors near the field that needs attention.

Key members: `Text`, appearance, visibility, and layout behavior.

```
Private Sub SetStatus(ByVal Message As String)
    staStatus.Text = Message
End Sub
```

Expected result: the user can glance at the current state while continuing to work.

Watch for: do not hide the only explanation of a failed action in a tiny status area.

Repository starting point: StatusBar.

ToolBar

Use it for: a visible row of frequently used commands.

Designer recipe: mirror the most common menu commands, keep the order stable, and provide text or tooltips so icons are not the only clue.

Key members: `Items`, `ClickedIndex`, `Click`, enabled state, and keyboard accessibility.

```
Private Sub barMain_Click()
    Select Case barMain.ClickedIndex
        Case 0
            SaveCurrentRecord
        Case 1
            RefreshBooks
    End Select
End Sub
```

Expected result: a toolbar click runs the same tested command as the menu.

Watch for: do not duplicate business logic in toolbar and menu handlers; call one shared procedure.

Repository starting point: ToolBar.

ScrollContainer

Use it for: content that may be taller or wider than the available window.

Designer recipe: put the content inside it, test a small window, and keep the main action visible when possible.

Key members: scroll behavior, content size, visibility, and focus order.

```
Private Sub Form_Resize()
    scrDetails.Width = Me.InsideWidth
    scrDetails.Height = Me.InsideHeight - staStatus.Height
End Sub
```

Expected result: the user can reach every field at a small window size.

Watch for: scrolling should solve a real size problem, not hide a form that needs to be split into smaller tasks.

Repository starting point: CompatibilityControls.

TimePicker

Use it for: a time of day or a time value whose precision is known.

Designer recipe: state whether seconds matter and whether the value is local time or part of a full date/time.

Key members: selected time/value, change event, allowed range, and focus.

```
Private Sub tmReminder_Change()
    lblStatus.Caption = "Reminder: " & CStr(tmReminder.Value)
End Sub
```

Expected result: the selected time is shown in a readable way and stored using one consistent convention.

Watch for: do not compare localized time strings when a time value is available.

Repository starting point: CompatibilityControls.

SplitContainer

Use it for: two related areas, such as a list on the left and details on the right, with a user-adjustable divider.

Designer recipe: set `Orientation`, `Ratio`, both minimum sizes, and decide whether either pane may collapse.

Key members: `Orientation`, `Ratio`, `SplitterWidth`, pane minimum sizes, `SplitterPosition`, collapse flags, and `SplitterMoved`.

```
Private Sub splLibrary_SplitterMoved()
    lblStatus.Caption = "Details pane: " & CStr(splLibrary.Ratio * 100) &
"%"
End Sub
```

Expected result: the divider moves without making either pane unusable.

Watch for: preserve a minimum useful width; a pane that collapses to a few pixels is not a usable responsive layout.

Repository starting point: SplitContainer.

MaskedTextBox

Use it for: a value with a visible shape, such as a phone number or code.

Designer recipe: choose a mask that guides entry without pretending to be complete business validation.

Key members: `Text`, mask, `Required`, length rules, change/focus events.

```
Private Sub txtPhone_Change()
    lblStatus.Caption = "Phone entered: " & CStr(Len(txtPhone.Text)) & "
chars"
End Sub
```

Expected result: the user sees the expected shape while the application still validates completeness before saving.

Watch for: decide whether mask characters are stored or removed before the value reaches the database.

Repository starting point: MaskedTextBox.

Gauge

Use it for: a current measured value, such as memory use, temperature, or an import rate.

Designer recipe: define the range and units and show the numeric value near the visual when accuracy matters.

Key members: value/range, label/unit, appearance, and update procedure.

```
Private Sub ShowHealth(ByVal Percent As Integer)
    gauHealth.Value = Percent
    lblHealth.Caption = "Health: " & CStr(Percent) & "%"
End Sub
```

Expected result: the visual and the exact number agree at zero, normal, and warning values.

Watch for: a Gauge is a current value, not proof that a long operation is progressing toward completion.

Repository starting point: Gauge.

Sparkline

Use it for: a compact trend beside a number or table row.

Designer recipe: keep the time window consistent and label the adjacent number; a sparkline alone has no scale.

Key members: series/values, range, appearance, and the update operation.

```
Private Sub UpdateTrend()
    spkSales.Values = "12,18,15,22,27"
    lblSales.Caption = "Sales today: 27"
End Sub
```

Expected result: the trend gives context while the label gives the exact current value.

Watch for: do not compare sparklines with different scales unless the scale is stated.

Repository starting point: CompatibilityControls.

ImageList

Use it for: a named collection of images shared by controls or repeated items.

Designer recipe: add project resources, give each image a stable name, and keep the collection close to the form or reusable component that owns it.

Key members: `Images`, `Count`, named/one-based lookup, and `Error`.

```
Private Sub Form_Load()
    If imgListMain.Count = 0 Or imgListMain.Error <> "" Then
        lblStatus.Caption = "Images not ready: " & imgListMain.Error
    Else
        lblStatus.Caption = CStr(imgListMain.Count) & " images loaded"
    End If
End Sub
```

Expected result: the app reports a missing resource instead of silently showing empty icons.

Watch for: resource names are part of the project contract. Check them after moving or renaming image files.

Repository starting point: ImageList.

ListView

Use it for: a list of items that needs a richer visual row than a ListBox, often with an icon, subtitle, or several columns.

Designer recipe: decide the row template first, then define the stable identity and the selection behavior.

Key members: items/columns, selected item, view mode, scrolling, and selection/pointer events.

```
Private Sub lvwFiles_SelectionChanged()
    If lvwFiles.SelectedIndex >= 0 Then
        lblStatus.Caption = "File: " & lvwFiles.Text
    End If
End Sub
```

Expected result: the selected row exposes both its readable text and its stable value to the next action.

Watch for: do not assume that the visible order is the database order after sorting or filtering.

Repository starting point: CompatibilityControls.

ImageCombo

Use it for: a compact choice list where each option benefits from an icon.

Designer recipe: make the icon support the text rather than replacing it, and keep a stable value behind each displayed choice.

Key members: `Items` (one `Text|ImageKey` line per item), `ImageList` (the nonvisual ImageList that supplies the item images), selected index, and

change/drop-down events. The closed box shows the image and the text of the selected item, left-aligned.

```
Private Sub icoStatus_Change()
    lblStatus.Caption = "Status: " & icoStatus.Text
End Sub
```

Expected result: the user can recognize the option quickly and the app can save a stable value.

Watch for: never make color or an icon the only difference between options.

Repository starting point: CompatibilityControls.

UpDown

Use it for: small step-by-step numeric changes, such as quantity or a version number.

Designer recipe: pair it with a visible numeric field, set a sensible step, and define the lower and upper bounds.

Key members: value, minimum, maximum, increment, change event, and focus.

```
Private Sub updCopies_Change()
    lblCopies.Caption = "Copies: " & CStr(updCopies.Value)
End Sub
```

Expected result: each click changes the value by one known increment and the displayed number remains in range.

Watch for: use NumericBox when users need to type a large or precise number rather than repeatedly clicking an arrow.

Repository starting point: CompatibilityControls.

MonthView

Use it for: choosing a month or browsing a month-based calendar view.

Designer recipe: show what selecting a month changes and keep the selected month separate from a specific day.

Key members: selected month/value, navigation, change event, and visible range.

```
Private Sub mvReport_Change()
    lblStatus.Caption = "Report month: " & CStr(mvReport.Value)
End Sub
```

Expected result: moving between months updates the report/filter context without inventing a day value.

Watch for: do not use MonthView when the task really needs an exact date; use DatePicker for that job.

Repository starting point: CompatibilityControls.

DTPicker

Use it for: a combined date/time selection when both parts belong to one value.

Designer recipe: state the expected precision and display format, then validate the complete date/time before saving.

Key members: date/time value, format, range, change event, and focus.

```
Private Sub dtpPublish_Change()
    lblStatus.Caption = "Publish at: " & CStr(dtpPublish.Value)
End Sub
```

Expected result: one control supplies one consistent date/time value.

Watch for: avoid splitting and rejoining localized strings; use the control's value.

Repository starting point: CompatibilityControls.

TabStrip

Use it for: a lightweight strip of tabs when you need tab selection without building a full page container around it.

Designer recipe: keep tab captions short and connect each selection to a clear content region.

Key members: tab collection, selected index, selection event, and keyboard navigation.

```
Private Sub tabsView_SelectionChanged()
    lblStatus.Caption = "View: " & tabsView.SelectedIndex
End Sub
```

Expected result: selecting a tab changes the associated view and keeps the selection visible.

Watch for: if each tab owns many controls, use TabControl and TabPage so the parent-child structure stays explicit.

Repository starting point: CompatibilityControls.

FlatScrollBar

Use it for: a deliberately visible horizontal or vertical scroll value, often alongside a custom visual or drawing surface.

Designer recipe: define the range and orientation and connect the value to the content position or zoom level.

Key members: orientation, minimum/maximum/value, change event, and enabled state.

```
Private Sub fsbPreview_Change()
    imgPreview.Left = -fsbPreview.Value
End Sub
```

Expected result: moving the bar moves the preview by a predictable amount.

Watch for: prefer ScrollContainer for ordinary form content; use this control when the bar itself is part of the design.

Repository starting point: CompatibilityControls.

Line

Use it for: a simple divider, connector, or visual guide.

Designer recipe: set direction, angle, stroke thickness, line style, and line cap before adjusting the position.

Key members: `Direction`, `Angle`, `StrokeThickness`, `LineStyle`, `LineCap`, and pointer/drag events.

```
Private Sub lnSeparator_MouseDown()
    lblStatus.Caption = "Separator selected"
End Sub
```

Expected result: the line remains visually clear at different sizes and does not pretend to be a data control.

Watch for: use a real layout container for grouping; a drawn line only explains a relationship visually.

Repository starting point: BuiltInAcceptance graphics.

Shape

Use it for: a simple geometric marker, background, or clickable visual.

Designer recipe: choose `ShapeKind`, then decide whether it is filled or outlined and set the stroke and radius.

Key members: `ShapeKind`, `StrokeThickness`, `LineStyle`, `Filled`, `Radius`, and pointer/drag events.

```
Private Sub shpAlert_MouseDown()
    lblStatus.Caption = "Alert marker selected"
End Sub
```

Expected result: the shape communicates state without becoming the only way to understand the action.

Watch for: pair a clickable shape with text, keyboard access, or another clear control when it performs an action.

Repository starting point: BuiltInAcceptance graphics.

CommonDialog

Use it for: standard open, save, folder, color, font, printer, or page setup workflows.

Designer recipe: set the title and filter before showing the dialog, then always check `Cancelled` before using the result.

Key members: `Title`, `Filter`, `FileName`, `Color`, `Font`, `PrinterName`, `PaperName`, `PageOrientation`, `Cancelled`, `LastAction`, and the `Show...` methods.

```
Private Sub cmdOpen_Click()
    dlgCommon.ShowOpen
    If dlgCommon.Cancelled Then
        staStatus.Text = "Open cancelled"
    Else
        staStatus.Text = "Opening " & dlgCommon.FileName
    End If
End Sub
```

Expected result: cancelling is a normal path, not an error, and the file is used only after the dialog returns a value.

Watch for: do not assume that a previous file name is a new selection. Clear or replace it according to the job.

Repository starting point: CommonDialog.

Animation

Use it for: a sequence of images that communicates motion or state.

Designer recipe: provide the image list/frames, choose `Interval`, decide whether it loops, and add Play/Pause/Stop commands when the user needs control.

Key members: `ImageList`, `Frames`, `Interval`, `AutoPlay`, `Loop`, `CurrentFrame`, `FrameCount`, `Playing`, `FrameChanged`, `Completed`, `Failed`, `Play`, `Pause`, `Stop`, `NextFrame`, and `PreviousFrame`.

```
Private Sub aniLogo_FrameChanged()
    lblStatus.Caption = "Frame " & CStr(aniLogo.CurrentFrame) & " of " &
CStr(aniLogo.FrameCount)
End Sub

Private Sub cmdPlay_Click()
    aniLogo.Play
End Sub
```

Expected result: the status explains the current frame and completion is visible to the user.

Watch for: stop or pause an animation when the form closes or when motion would distract from the task.

Repository starting point: Animation.

AudioPlayer

Use it for: loading and playing an audio resource or supported source.

Designer recipe: set `Source`, decide `AutoPlay` and `Loop`, set a useful `Volume`, and include Play, Pause, and Stop commands when appropriate.

Key members: `Source`, `AutoPlay`, `Loop`, `Volume`, `Position`, `Duration`, `Playing`, `Paused`, `State`, `Error`, `Loaded`, `Completed`, `Failed`, `StateChanged`, `Load`, `Play`, `Pause`, and `Stop`.

```
Private Sub cmdPlay_Click()
    audTone.Position = 0
    audTone.Play
End Sub

Private Sub audTone_Failed()
    lblStatus.Caption = audTone.Error
End Sub
```

Expected result: playback starts when requested and a missing or invalid source produces a readable message.

Watch for: never assume a resource loaded because the Play button was pressed; handle Loaded and Failed.

Repository starting point: AudioPlayer.

VideoPlayer

Use it for: showing a video source with optional controls and stretching.

Designer recipe: set `Source`, choose `ControlsVisible` and `Stretch`, and keep the video area large enough to be useful at the smallest window size.

Key members: audio-player members plus `ControlsVisible`, `Stretch`, `Loaded`, `Completed`, `Failed`, `StateChanged`, `Play`, `Pause`, and `Stop`.

```
Private Sub cmdPlay_Click()
    If vidDemo.Duration > 0 Then vidDemo.Position = 0
    vidDemo.Play
End Sub

Private Sub vidDemo_Loaded()
    lblStatus.Caption = "Loaded: " & CStr(vidDemo.Duration) & " seconds"
End Sub
```

Expected result: the user can start and stop the video and understands its duration and failure state.

Watch for: test the published resource path and a missing-file path; both are normal release scenarios.

Repository starting point: VideoPlayer.

Tips and tricks

- **Choose a component by the user's job.** Start from what the user must do, then pick the component.
- **Use Create Control Array… for repeated controls.** Right-click the control and choose it, then handle all members in one procedure with `Index`.

- **Give a CommandButton a Picture.** Set **Picture** and **PictureAlignment** in Properties. The picture must be an image resource of the project.
- **Test the missing-file path.** A component that shows a file should also handle the file being absent.

Workshop 2 - Practice the language with small exercises

Change a working example and watch what happens. The exercises are deliberately small: finish one, choose Check, run one visible test, and then move on.

Use a copy of `HelloForms` or the appointment form from the previous workshop. The examples use the names shown in those chapters. Rename them if your form uses different names.

Exercise 1: make a clean greeting

Change the greeting handler so that leading and trailing spaces do not change the result.

```
Private Sub cmdGreet_Click()
    Dim name As String
    name = Trim$(txtName.Text)

    If Len(name) = 0 Then
        lblGreeting.Caption = "Enter a name."
        Exit Sub
    End If

    lblGreeting.Caption = "Hello, " & name
End Sub
```

Try: empty input, three spaces, and `  Ada  `.

Expected result: empty input asks for a name; the other two inputs produce the same greeting for Ada.

Exercise 2: show a count

Add a label named `lblCount`. Update it in the text box Change event:

```
Private Sub txtName_Change()
    lblCount.Caption = "Characters: " & CStr(Len(txtName.Text))
End Sub
```

Try: type, delete, and paste text.

Check: the count changes while you type and returns to zero when the field is empty.

Exercise 3: keep the normal path short

Write a Save handler that validates first:

```
Private Sub cmdSave_Click()
    Dim name As String
    name = Trim$(txtName.Text)

    If Len(name) = 0 Then
        lblStatus.Caption = "Enter a name before saving."
        txtName.SetFocus
        Exit Sub
    End If

    lblStatus.Caption = "Saved: " & name
End Sub
```

The word **Saved** is only a demonstration until a real data operation exists. Use honest messages in a production app.

Exercise 4: choose between two messages

Use a CheckBox named `chkConfirmed`:

```
Private Sub cmdContinue_Click()
    If chkConfirmed.Value = False Then
        lblStatus.Caption = "Confirm the choice first."
        Exit Sub
    End If

    lblStatus.Caption = "The choice is confirmed."
End Sub
```

Try: run the handler with the box unchecked and checked.

Check: the user can understand what is missing without reading the code.

Exercise 5: use a function for a rule

Move the rule into a function when two events need the same answer:

```
Private Function HasName() As Boolean
    HasName = Len(Trim$(txtName.Text)) > 0
End Function

Private Sub cmdSave_Click()
    If HasName() = False Then
        lblStatus.Caption = "Enter a name."
        Exit Sub
    End If

    lblStatus.Caption = "Ready to save."
End Sub

Private Sub cmdPreview_Click()
    If HasName() = False Then
        lblStatus.Caption = "Enter a name before previewing."
        Exit Sub
    End If
```

```
    lblStatus.Caption = "Preview is ready."
End Sub
```

Why this helps: when the rule changes, you update one function rather than two handlers.

Exercise 6: count a small set of values

Use a `For` loop when the number of iterations is known:

```
Private Sub cmdCount_Click()
    Dim index As Long
    Dim total As Long

    total = 0
    For index = 1 To 5
        total = total + index
    Next index

    lblStatus.Caption = "Total: " & CStr(total)
End Sub
```

Expected result: the label shows `Total: 15`.

Change the upper limit to `0` and then to `10`. Write down the result before running each version. This small habit catches many loop mistakes.

Exercise 7: choose a branch with Select Case

When one value selects among several named cases, `Select Case` can be easier to read than many nested `If` statements:

```
Private Sub cmdPriority_Click()
    Select Case txtPriority.Text
        Case "High"
            lblStatus.Caption = "Handle today."
        Case "Normal"
            lblStatus.Caption = "Handle this week."
        Case Else
            lblStatus.Caption = "Choose High or Normal."
    End Select
End Sub
```

Test the exact values and an empty value. Decide whether your real app should accept different letter case before you add a conversion rule.

Exercise 8: keep database work behind a message

When an operation can take time, show the state before and after it. The exact database call depends on the release contract and sample you are using, but the user-facing sequence should be stable:

```
Private Async Sub cmdRefresh_Click()
    cmdRefresh.Enabled = False
    lblStatus.Caption = "Loading..."

    ' Call the supported database operation here.
```

```
    ' rows = Await LoadCustomersAsync()

    lblStatus.Caption = "Loaded."
    cmdRefresh.Enabled = True
End Sub
```

Do not copy the comments into a finished handler and claim that data loaded. Replace them with the exact supported call, then add an error and cancellation path before shipping.

Exercise 9: make a second form honest

Open a new form only after the first form has a clear reason to do so. Pass the minimum value it needs:

```
Private Sub cmdDetails_Click()
    Dim details As frmDetails
    Set details = New frmDetails
    details.DisplayName = Trim$(txtName.Text)
    details.Show
End Sub
```

Test an empty name and a normal name. Decide whether the details form should show an empty state or whether the first form should reject the action.

Exercise 10: write a diagnostic on purpose

In a copy of the project, temporarily change a control name in one line of code. Choose **Check** and read the diagnostic. Then restore the correct Name and Check again.

This exercise teaches the recovery path without waiting for a random error. Never leave the intentional error in the version you publish.

Exercise 11: make a test table

For one event, fill out this table before writing more code:

Case	Input	Expected result	Actual result
empty			
normal			
spaces			
long value			
cancelled action			

A test table turns a vague feeling that "it works" into evidence you can repeat after the next change.

Exercise 12: explain your code in plain English

Choose one handler and write five lines beginning with:

1. When...
2. The code reads...
3. If...
4. Otherwise...
5. The user sees...

If you cannot write those five lines, the handler may be doing too much. Split out a function or a smaller procedure before adding another feature.

Answer key: what good results look like

There is not one correct user interface, but a good solution has these traits:

- the input rule is visible to the user;
- the event name matches the control and event;
- the code handles the invalid case before the normal case;
- the success message describes what really happened;
- Check is run after each meaningful change;
- the test includes a failure or empty case, not only a happy path.

Final challenge

Combine Exercises 1, 3, and 11 into a small form. Ask another person to use it without seeing the code. Do not explain the labels while they work. Record the first hesitation, change one thing, and repeat the test.

Tips and tricks

- **Type the exercise yourself.** Typing builds the habit better than pasting.
- **Choose Check after every exercise.** Read the **Diagnostics** list before you run.
- **Make a mistake on purpose in Exercise 10.** Seeing the message once makes the next one easier to read.
- **Compare with the answer key last.** Try first and check afterwards.

Workshop 3 - Review a project before delivery

This workshop takes a working prototype and prepares it for another person to test. It does not replace the release process; it gives you a second pair of eyes for the mistakes that are easy to miss when you built the project yourself.

Start with the user story

Write the main task in one sentence:

> *A support worker can find a customer, update the contact note, and export a copy without losing the original record.*

Every control, procedure, and export in the project should support that task. If a feature does not support it, move the feature to a later version instead of making the first release harder to test.

Make a feature map

Create a short table before opening the code:

Feature	Screen	Event	Data or output	Test
find a customer	main form	Search	filtered rows	exact and partial search
edit a note	edit area	Save	SQLite row	valid and empty note
export a list	main form	Export	CSV or PDF file	new and existing file
close safely	app/form	Close	saved or cancelled state	busy operation

This map gives a tester a route through the app. It also exposes missing events: if a feature has a control but no event, it is not finished.

Review the form as a first-time user

Perform these steps without reading the source:

1. open the app;
2. identify the main task;
3. use the keyboard to reach the first input;
4. enter an invalid value;
5. read the message and recover;
6. complete the normal task;
7. close and reopen the app;
8. confirm what was saved.

Write down the exact words that cause hesitation. Replace vague labels such as **Go**, **Do it**, or **Process** with the result the user wants.

Review the source as a maintainer

Look at one event at a time:

- Does its name match its control and event?
- Does it read the input before changing data?
- Does it reject invalid input before starting slow work?
- Does it disable a duplicate action while work is running?
- Does it restore the enabled state after success, error, and cancellation?
- Does it show a useful message without exposing private data?

Do not judge code only by its length. A short handler that hides a destructive operation can be more dangerous than a longer handler with clear checks.

Test data boundaries

Use fake data with these properties:

- an empty required field;
- one normal record;
- a very long name or note;
- accented characters;
- an apostrophe and a comma;
- a duplicate record;

- a missing file;
- a locked or unavailable database;
- an existing export file.

Keep the test data in a separate folder. Do not use real customer data while you are still changing the schema or error handling.

Review the published output

Run the published app outside the IDE when the release process supports it. Check:

1. first launch;
2. form layout and text readability;
3. resources and images;
4. user-data path;
5. database creation or opening;
6. export permissions;
7. close and relaunch;
8. version information.

An app that works only because the IDE supplies a file or package is not yet a complete customer test.

Prepare a support note

Keep a plain-text note with:

```
Product:
Version:
Mac model and architecture:
macOS version:
Project name:
Steps to reproduce:
Expected result:
Actual result:
Exact diagnostic:
```

Remove private paths, tokens, passwords, and customer data before sharing the note. A clear report saves more time than a large screenshot with no context.

Sign-off table

Do not use a check mark until you have evidence:

Review	Evidence	Done
source checks	clean Check output	[ ]
normal workflow	completed by a new user	[ ]
invalid workflow	message and recovery observed	[ ]
data persistence	close and reopen test	[ ]
output app	run outside the IDE	[ ]
private data	project and screenshots inspected	[ ]
version record	release and build written down	[ ]

Expected result

Another person can follow the feature map, complete the main task, recover from one mistake, and explain what happened. You have a dated record of the release you tested and the evidence that supports the result.

Check your work

Ask a tester to repeat the main story from a clean copy. Do not help unless the tester is blocked by a real product defect. After the session, fix the largest source of confusion, not every small preference.

Next challenge

Turn the feature map into one page of help inside the app. Use the same words for the button Caption, the status message, the manual, and the support note. Consistent language is one of the cheapest usability improvements.

Tips and tricks

- **Review as a first-time user.** Do not explain the labels while someone tries your app. Note where they hesitate.
- **Test the data boundaries.** Try empty, very long and unusual values.
- **Test the published output.** Check the published app separately from the IDE.
- **Write the support note while the project is fresh.** Add the version, the steps and the expected result.

Practical workflow recipes

The earlier chapters introduced the parts of an application. These recipes put those parts together around ordinary user problems. Each one ends with a test you can run.

Use one recipe at a time. Copy the idea into a small project before combining it with database, network, or package code.

Recipe 1: validate before a slow operation

User problem: the user presses Search with an empty query, and the app starts work that cannot produce a useful result.

Design: validate the query first, then disable the button while the search is active.

```
Private Async Sub cmdSearch_Click()
    Dim query As String
    query = Trim$(txtSearch.Text)

    If Len(query) = 0 Then
        lblStatus.Caption = "Enter a search term."
        txtSearch.SetFocus
        Exit Sub
    End If

    cmdSearch.Enabled = False
    lblStatus.Caption = "Searching..."

    ' Call the supported search operation here.
    ' results = Await SearchAsync(query)

    lblStatus.Caption = "Search complete."
    cmdSearch.Enabled = True
End Sub
```

Before shipping, add the supported error and cancellation path. The button must also be re-enabled when the operation fails.

Test: empty query, normal query, slow response, failed response, and a second click while the first search is active.

Recipe 2: show an empty state

User problem: a grid shows no rows, and the user cannot tell whether the search failed or whether there are simply no matching records.

Design: give the empty state its own message and action.

```
Private Sub ShowCustomerState(ByVal rowCount As Long)
    If rowCount = 0 Then
        lblStatus.Caption = "No customers match this search."
        cmdAdd.Enabled = True
        Exit Sub
    End If
```

```
    lblStatus.Caption = CStr(rowCount) & " customer(s) found."
    cmdAdd.Enabled = True
End Sub
```

The empty state should not look like a broken form. Tell the user what to try next: change the search, clear the filter, or add a new record.

Test: no rows, one row, many rows, and a filter that removes the last row.

Recipe 3: keep a selected record visible

User problem: the user selects a row, refreshes the grid, and the edit area displays values from a different record.

Design: store the stable record identity, refresh the rows, then either restore the same record or clear the edit area.

```
Private selectedCustomerId As Long

Private Sub cmdRefresh_Click()
    Dim oldId As Long
    oldId = selectedCustomerId

    ' Load the rows here. Awaiting a load needs an Async Sub:
    ' Set rows = Await LoadCustomersAsync()

    If oldId = 0 Then
        ClearCustomerEditor
    Else
        ' Select the row with oldId when it still exists.
        ' Otherwise call ClearCustomerEditor.
    End If
End Sub
```

The comments mark release-specific data binding. The design rule is stable: never confuse a visible row position with a record identity.

Test: refresh with the selected row still present, refresh after it is deleted, and refresh with no selection.

Recipe 4: confirm a destructive action

User problem: Delete is next to Save, and a mistaken click can remove data.

Design: make the record and consequence clear before the operation.

```
Private Sub cmdDelete_Click()
    If selectedCustomerId = 0 Then
        lblStatus.Caption = "Select a customer first."
        Exit Sub
    End If

    lblStatus.Caption = "Delete the selected customer, then " & _
        "choose Confirm."
    cmdDelete.Enabled = False
    cmdConfirmDelete.Enabled = True
End Sub
```

The exact confirmation surface depends on the release and the form design. For a destructive action, give the user a clear second step or an undo path. Do not write a fake confirmation message and delete immediately.

Test: no selection, one selection, cancel, confirm, and a second click.

Recipe 5: keep unsaved input safe

User problem: the user edits a form, changes selection, and loses the text without warning.

Design: track a dirty state and ask what to do before replacing the input.

```
Private editorDirty As Boolean

Private Sub txtNotes_Change()
    editorDirty = True
    lblStatus.Caption = "Unsaved changes"
End Sub

Private Sub MarkSaved()
    editorDirty = False
    lblStatus.Caption = "Saved"
End Sub
```

Use the supported close or navigation event to check `editorDirty`. Offer **Save**, **Discard**, and **Cancel** only when all three choices are real.

Test: edit then close, edit then save, edit then discard, and edit then cancel the close.

Recipe 6: make a search easy to undo

User problem: the user filters a list and cannot get all records back.

Design: place a visible Clear action beside the SearchBox.

```
Private Sub cmdClearSearch_Click()
    txtSearch.Text = ""
    lblStatus.Caption = "Showing all customers."
    txtSearch.SetFocus
End Sub
```

If the search begins on every Change event, consider a short explicit Search button or a release-supported delay. A search that starts too often can make a small app feel slow.

Test: search, clear, search again, and close and reopen the form.

Recipe 7: keep a file export honest

User problem: an export fails because the target file already exists.

Design: check the destination and tell the user what happened.

```
Private Sub cmdExport_Click()
    Dim path As String
    path = AppPaths.DataFile("exports/customers.csv")

    If File.Exists(path) Then
        lblStatus.Caption = "The export already exists. " & _
            "Choose a new name."
        Exit Sub
    End If

    ' Call the supported export operation here.
    lblStatus.Caption = "Export complete."
End Sub
```

The exact path and export API must match the release contract. Do not delete the existing file automatically just to make a demo pass.

Test: new destination, existing destination, missing parent folder, and data containing commas, quotes, and accented characters.

Recipe 8: report a cancelled operation

User problem: a user chooses Cancel, but the screen still says that the operation succeeded.

Design: give success, failure, and cancellation different outcomes.

```
Private Async Sub cmdDownload_Click()
    cmdDownload.Enabled = False
    cmdCancel.Enabled = True
    lblStatus.Caption = "Downloading..."

    ' Await the supported download operation here.
    ' A cancellation result must not be shown as success.

    lblStatus.Caption = "Download complete."
    cmdCancel.Enabled = False
    cmdDownload.Enabled = True
End Sub
```

Add the real error and cancellation handling before using this pattern. Remove partial output after a failed or cancelled transfer when the API contract requires it.

Test: complete, fail, cancel early, cancel late, and close while active.

Recipe 9: pass only what a second form needs

User problem: a details window reads many controls directly from its parent, so the two forms are hard to change independently.

Design: pass a small value or a stable record identity.

```
Private Sub cmdDetails_Click()
    Dim details As frmDetails
    Set details = New frmDetails
    details.DisplayName = Trim$(txtName.Text)
    details.Show
End Sub
```

For a real data screen, pass the record ID and let the details form load the record through the supported service. Avoid passing an entire form as a bag of unrelated values.

Test: open two detail windows with different values, close one, and confirm that the main form remains usable.

Recipe 10: use a timer for display, not hidden work

User problem: a clock or status indicator needs a regular update.

Design: keep the Timer event short and stop it when the form unloads.

```
Private Sub tmrClock_Timer()
    lblClock.Caption = CStr(Now)
End Sub
```

Use the supported date formatting function for the public release. Do not put a database query or network request directly into every timer tick.

Test: start, stop, resize, and unload the form. Confirm that no timer event continues to update a form that no longer exists.

Recipe 11: make accessibility part of the design

User problem: a form looks clear in a screenshot but is difficult to use with the keyboard or assistive technology.

Design: give controls a logical TabIndex, a useful Name, accessible text, and a visible message for important state.

```
Private Sub cmdSave_Click()
    If Len(Trim$(txtName.Text)) = 0 Then
        lblStatus.Caption = "Name is required."
        txtName.SetFocus
        txtName.SelectAll
        Exit Sub
    End If

    lblStatus.Caption = "Ready to save."
End Sub
```

The exact accessibility properties belong in the Inspector and release reference. Code cannot fix a confusing task order by itself.

Test: complete the main task with the keyboard only, increase text size if the environment allows it, and check that color is not the only error signal.

Recipe 12: make a component replaceable

User problem: a form depends on a control's internal details, so changing the visual design requires rewriting the business rule.

Design: keep the rule in a function and let the event translate control values into that rule.

```
Private Function CanSave(ByVal name As String) As Boolean
    CanSave = Len(Trim$(name)) > 0
End Function

Private Sub cmdSave_Click()
    If CanSave(txtName.Text) = False Then
        lblStatus.Caption = "Enter a name."
        Exit Sub
    End If

    lblStatus.Caption = "Ready to save."
End Sub
```

Later, the input could come from a different control without changing the meaning of `CanSave`.

Test: call the same function from Preview and Save. Both actions should agree about whether the input is valid.

Recipe 13: design the error message first

User problem: an error message says only that something failed.

Design: write the message with three parts: what failed, why it matters, and what the user can do next.

```
Could not export the customer list because the file already exists.
Choose a new file name, or remove the old export if it is no longer
needed.
```

Avoid internal exception names in the first message. Keep technical detail in the diagnostic or support note where it helps investigation.

Test: ask someone who did not build the app to recover using the message alone.

Recipe 14: keep a release note beside the project

User problem: a later build changes a menu or property, and nobody knows which version the screenshots and tests described.

Design: record the release, build, Mac architecture, and test date.

```
McBasic version: [PUBLIC MCBASIC VERSION]
Build: [BUILD NUMBER]
Mac architecture: [ARM64 OR OTHER SUPPORTED TARGET]
macOS: [VERSION]
Tested: [DATE]
Known limits: [SHORT LIST]
```

Do not replace the bracketed fields with guesses. Complete them when the public release is chosen.

Recipe 15: turn a bug report into a test

User problem: a report says "it sometimes loses data" but cannot be reproduced.

Design: write the report as a sequence with an expected and actual result.

```
1. Open the WeatherDesk sample.
2. Find the city "Singapore".
3. Save the place and its current weather.
4. Choose Refresh after closing and reopening the app.
Expected: the saved location remains in the SQLite-backed grid.
Actual: the saved location disappears without a message.
```

This is now a test case. Keep it failing while you investigate, then keep it as a regression test after the fix.

Recipe 16: add a feature with the AI assistant

User problem: you want a small new feature, such as a Clear button, but you do not want an AI change you have not read to end up in the project.

Design: give the assistant only the form involved, describe the behaviour in user terms, and read the change card before you keep the change. If you want to approve the change before it is applied, set **Apply AI changes:** to **Ask me first** in the AI chat section of AI Provider Settings.

The request:

```
Add a button named cmdClear with the caption Clear. When it is clicked,
empty txtFirst and txtSecond, set lblResult to an empty string, and put
the focus in txtFirst. Do not change the other controls.
```

The steps:

1. Open the form in the editor and open the AI chat (Shift–Command–L).
2. Check the chips: keep only `frmMain · code+form` (or the form you are changing). Choose **Show exact content** if you want to see what will be sent.
3. Send the precise request shown above the steps.
4. Read the change card that appears in the chat, for example "Changed form frmMain" or "Changed procedure `cmdClear_Click` in frmMain". Open **N files changed** to see the form operations and the BASIC lines. If the request became a plan, read the plan card and choose **Approve and run**. Keep the change only if every line is one you asked for; otherwise choose **Undo** on the card (or **Discard**, in Ask me first mode).
5. Choose **Check**, then **Run**, and test the new button.
6. Save. If the result is not what you wanted, **Edit > Undo** removes the whole change in one step.

A finished handler for this request looks like this:

```
Private Sub cmdClear_Click()
    txtFirst.Text = ""
    txtSecond.Text = ""
    lblResult.Caption = ""
    txtFirst.SetFocus
End Sub
```

Test: fill both boxes and a result, choose Clear, and confirm the boxes and the label are empty and the cursor is in the first box. Then choose **Edit > Undo** once in the IDE and confirm the button and its handler are gone.

A recipe review checklist

For every new workflow, ask:

- What is the user's first action?
- What is the empty or invalid case?
- What can take time?
- What can be cancelled?
- What data must remain safe?
- What should the user see after success?
- What should the user do after failure?
- How will a tester prove the result?

If the answers are not clear, the feature is not ready for a large form or a customer release.

Test workbook

Use this workbook after the tutorials and before you share an application. It turns the advice in the book into tests you can repeat. Print the pages or copy the tables into a project note.

How to use the workbook

For each test:

1. start from the stated condition;
2. follow the steps without changing the order;
3. write the expected result before you run the test;
4. record the actual result in your own copy;
5. save the exact diagnostic if the result is different;
6. repeat the test after the fix.

Use a clean copy of the project when the test says **clean start**. Do not call a test successful because the developer already knows how to work around the problem.

Test 1: project opens

Purpose: prove that the project is complete enough to begin.

Step	Expected result	Actual result
Open the `.mcbproj` project	McBasic™ opens the project	
Read the project tree	expected forms and modules are present	
Open the main form in Design view	the layout and controls are visible	
Choose Check	no new diagnostic appears	

Pass rule: every expected result is observed from a clean copy.

Test 2: first launch

Purpose: prove that a new user can start the application.

Step	Expected result	Actual result
Run the project or published app	the startup window opens	
Read the title and main action	the purpose is understandable	
Press Tab	focus moves to the first useful control	
Choose the main action	a visible result appears	
Close the app	it closes without an unexplained error	

Record the release and build used for this test.

Test 3: required input

Purpose: prove that an empty required value is handled safely.

Step	Expected result	Actual result
Leave the field empty	no real data is written	
Choose Save or Continue	a useful message appears	
Read the message	it says what to enter next	
Observe focus	focus moves to the field that needs attention	
Enter a valid value	the message clears or changes	

Pass rule: the app does not report success when it did not save anything.

Test 4: normal record

Purpose: prove one complete data path.

Step	Expected result	Actual result
Enter a normal record	the fields accept the values	

Step	Expected result	Actual result
Choose Save	the app reports success	
Refresh or return to the list	the record appears once	
Select the record	the edit area shows the same values	
Close and reopen	the record is still present	

Use fake data such as `Ada Example`, not a real customer's details.

Test 5: unusual text

Purpose: prove that ordinary international text is not damaged.

Try each value in a safe test project:

```
Zoë Example
O'Connor, Jr.
São Paulo
Name with a very long note that wraps across more than one line.
```

Check	Expected result	Actual result
Display	text remains readable	
Save	value is accepted or clearly rejected	
Reload	value is unchanged	
Export	quotes, commas, and accents are handled	

Test 6: duplicate or existing data

Purpose: prove that the app explains a conflict.

Step	Expected result	Actual result
Save a record with a stable identity	the first save succeeds	
Try the same identity again	the app explains the conflict	
Edit a stale copy	a newer value is not silently overwritten	

Step	Expected result	Actual result
Cancel the conflict	the existing data remains safe	

If the product has not implemented conflict handling, mark this test **Not available in this release**. Do not write a success claim into the manual.

Test 7: search and empty results

Purpose: prove that the user can find data and recover from no matches.

Step	Expected result	Actual result
Search for an exact value	the matching row appears	
Search for a partial value	the documented partial-match behavior appears	
Search for nonsense text	the empty state explains what to do	
Clear the search	the complete list returns	
Search again	the previous empty state does not remain	

Test 8: more than one form

Purpose: prove that windows keep the right data.

Step	Expected result	Actual result
Open the details form for Ada	it shows Ada	
Return to the main form	it remains responsive	
Open details for another value	the second window shows the new value	
Close one details window	the other window behaves as documented	
Close the main form	child windows follow the release rule	

Test the exact window behavior documented for the release. Modal and MDI rules are not interchangeable.

Test 9: file export

Purpose: prove that an output file is useful and safe.

Step	Expected result	Actual result
Export to a new path	the file is created	
Open the file	its content matches the visible data	
Export to the same path	the app explains the existing file	
Cancel an export if supported	no misleading success message appears	
Inspect the file path	it is writable and does not contain a secret	

Test 10: slow work and cancellation

Purpose: prove that a long operation communicates its state.

Step	Expected result	Actual result
Start the operation	progress or waiting state is visible	
Observe the action button	duplicate start is disabled or handled	
Choose Cancel	the operation reports cancellation	
Inspect the output	no invalid partial result remains	
Start again	the app can recover	

Use a controlled local test input when a real network service would make the result unpredictable.

Test 11: keyboard path

Purpose: prove that the main task does not require a mouse.

Step	Expected result	Actual result
Open the form	focus is visible	
Use Tab and Shift-Tab	order follows the task	
Change a value	the correct event runs	
Activate the main action	the same result as a pointer click appears	
Correct an error	focus returns to the useful field	

Test 12: published output

Purpose: separate source success from delivery success.

Step	Expected result	Actual result
Build or publish from a clean project	output is created	
Run outside the IDE	the app starts	
Use the main workflow	resources and packages are available	
Create user data	the path is writable	
Quit and relaunch	saved data and state follow the documented rule	
Inspect version information	it matches the release record	

Test 13: accessibility review

Purpose: find problems that a screenshot cannot show.

Check	Expected result	Actual result
control Names	they are useful and unique	
visible labels	every input has a clear label	
focus order	it follows the task	

Check	Expected result	Actual result
error messages	they do not rely on color alone	
accessible names/descriptions	important controls have useful text	
long text	it remains readable or scrollable	

Test 14: privacy review

Purpose: make sure a sample, screenshot, or support report contains no private material.

Inspect	Expected result	Actual result
project files	no passwords or tokens	
database	fake or approved test data only	
screenshots	no private paths or names	
logs	no secrets or full customer records	
exported files	removed or stored in a safe test folder	
saved AI chats (`AI/Activities` in the `.mcbproj`)	none, or only chats you intend to share	

Test 15: release record

Fill in one row for every proof run:

Date	McBasic version	Build	Mac	Test	Result	Evidence

Do not write **Pass** when the test was skipped. Use **Not run** or **Not available in this release**, then explain why.

A failure report template

```
Test name:
Date:
McBasic version and build:
Mac and macOS:
Starting project or sample:
Steps:
Expected result:
Actual result:
Diagnostic text:
Files or screenshots attached:
Private data removed:
```

Final test cycle

The project is ready for a complete test cycle when:

- the main story passes from a clean start;
- every important invalid case has a useful recovery;
- data remains safe after close and reopen;
- the published output is tested outside the IDE;
- the test record names the exact release;
- no private data is included in screenshots, logs, or example data.

Keep the completed workbook with the project. When a new McBasic™ release changes a menu, component, or language rule, repeat the affected tests before sharing the application again.

Topic finder

Use this appendix when you are returning to the book with a specific question. The first list is organized by goal. The second list is alphabetical.

I want to...

Start and build

Goal	Start here
understand the McBasic™ mental model	Welcome to McBasic
install or open a project	Installation, projects, and the first launch
run the first example	Your first working application
find a panel in the IDE	The IDE at a glance
place a control	Designing forms
write a Click event	Writing McBasic event code
choose the right control	Choosing and using components and the component cookbook
open a second window	Working with more than one form
split code into modules	Forms, modules, classes, and reusable logic
validate user input	Handling input and validation

Work with data and delivery

Goal	Start here
learn a language feature	Dates, arrays, collections, and control flow
save data in SQLite	SQLite and application data
read files or call a service	HTTP, JSON, files, settings, and reports
reuse a component package	Reusable controls and offline packages

Goal	Start here
understand an error	Checking, testing, debugging, and recovering from errors
ask other McBasic™ developers (Discord)	Chapter 2 — Welcome window links; Chapter 23 — When to ask for help
get help from the AI assistant	Chapter 16 — AI assistant, workflow recipes
control what the AI may send	Chapter 16 — AI assistant
decide whether AI changes apply automatically	Chapter 16 — AI assistant
continue or stop a paused AI plan	Chapter 16 — AI assistant, Quick reference
publish an app	Building and publishing an app
study a larger example	Capstone: build WeatherDesk
practise from an empty form	Workshop 1
practise BASIC syntax	Workshop 2
review a project for delivery	Workshop 3
find a reusable workflow pattern	Practical workflow recipes
plan a proof run	Test workbook
look up a component	Standard component reference
look up a statement or type	McBasic language reference
understand project files	Project files and folder layout
find a recovery step	Diagnostics and troubleshooting
check a technical word	Glossary

Alphabetical index

This is a learning index, not a promise that every item has identical support in every release. When a feature is marked Limited or Planned in the chapter, follow the release note before using it in a customer product.

Term	Where to look
AI assistant	Chapter 16 — AI assistant, Quick reference
AI cards (plan, stage, change, failure)	Chapter 16 — AI assistant, Glossary

Term	Where to look
AI chats, where they are stored	Chapter 16 — AI assistant, Project files
AI errors	Chapter 16 — AI assistant, Diagnostics
AI limits and request size	Chapter 16 — AI assistant
AI settings (Apply AI changes)	Chapter 16 — AI assistant
AI undo	Chapter 16 — AI assistant, Quick reference
Accessibility	Designing forms, workflow recipes
Arrays	Language toolkit, language reference
Async and Await	Files, network, and reports, workflow recipes
Backups	Project files and folder layout
Build	Build and publish, Glossary
Build cache	Project files and folder layout
Buttons	Components, Component cookbook
Chat (AI)	Chapter 16 — AI assistant
Check	IDE at a glance, Diagnostics
Check messages (codes)	Appendix — Check messages by code, Diagnostics
Classes	Code organization, Language reference
ComboBox	Component reference, Component cookbook
Codex	Chapter 16 — AI assistant
Components	Components, Component cookbook
Context chips	Chapter 16 — AI assistant
Controls	Designing forms, Component reference
Linux	Chapter 21 — Publish for Linux
WeatherDesk	Capstone project, Test workbook
DataGrid	SQLite and data, Component cookbook
Dates and times	Language toolkit, Component cookbook
Debugging	Check, test, and debug, Diagnostics
Diagnostics	Diagnostics and troubleshooting

Term	Where to look
Events	Writing event code, Language reference
Export	Files, network, and reports, Workflow recipes
Forms	Designing forms, Working with more than one form
Functions	Writing event code, Language reference
Glossary	Glossary
IDE	The IDE at a glance
Input validation	Handling input and validation, Workshop 1
JSON	Files, network, and reports, Language reference
Layout	Designing forms, Component reference
LM Studio	Chapter 16 — AI assistant
Lists and choices	Component cookbook
Loops	Language toolkit, Practice lab
Menus	Component reference, Component cookbook
Modules	Code organization, Project files
Multiple forms	Working with more than one form, Workflow recipes
OpenRouter	Chapter 16 — AI assistant
Packages	Reusable controls and offline packages
Plans and stages (AI)	Chapter 16 — AI assistant
Pauses in an AI plan	Chapter 16 — AI assistant, Diagnostics
Privacy (AI)	Chapter 16 — AI assistant
Project packages (`.mcbproj`)	Installation and projects, Project files
Properties	The IDE at a glance, Component reference
Publishing	Building and publishing, Publish for Windows, Publish for Linux, Project review
SQLite	SQLite and application data, WeatherDesk
Testing	Check, test, and debug, Test workbook

Term	Where to look
Timer	Component reference, Workflow recipes
Troubleshooting	Diagnostics and troubleshooting
Variables and types	Writing event code, Language reference
Windows	Chapter 20 — Publish for Windows

When you come back to a topic

When you return to the book, start with the goal rather than the chapter number. Read the short explanation, copy the smallest example, and run the matching test from the workbook. This keeps reference work connected to a visible result.