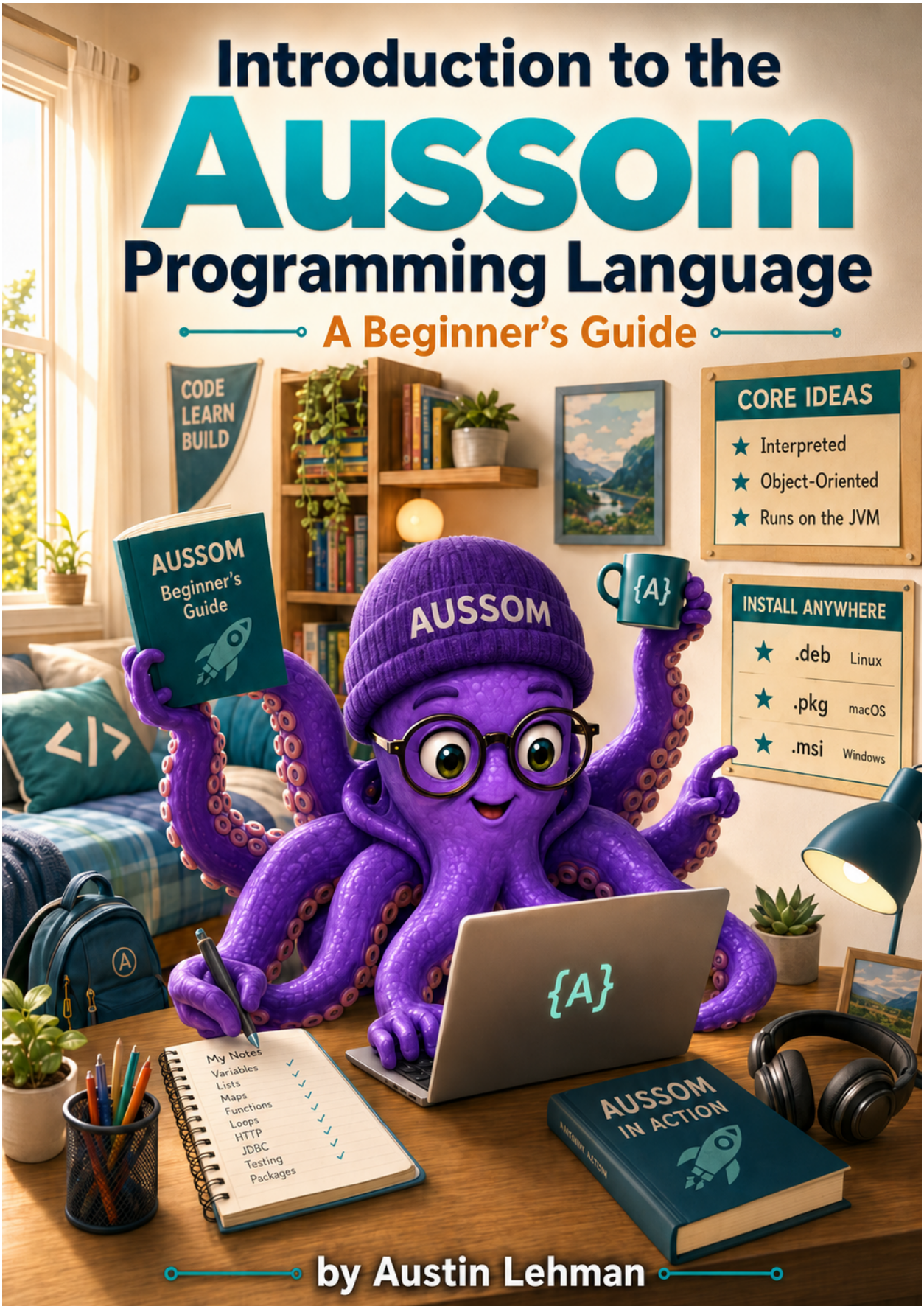
Introduction to the Aussom Programming Language

A Beginner's Guide



by Austin Lehman

Copyright

Revision 1 (2026)

Contents

Introduction

I started on this book in an effort to provide a true beginner's guide to the Aussom programming language, one that starts from the very beginning and builds up the core of the language piece by piece, with runnable examples and pointers to the API references along the way. This book is just that, a straightforward guide for getting you writing real Aussom programs, even if you have never written a line of code before. This book is one of a number of books for the Aussom Programming Language that I plan to write to provide anyone an easy path to getting started.

I want to say a quick thank you to all the people who helped to get Aussom to where it is today, and to those whose libraries we directly or indirectly use in the language and its standard library. There are too many to mention each by name here, but there are countless open-source libraries written by hardworking individuals on their own time that help to make this all possible. Many thanks to them and please keep them in mind when you read this book and write programs in Aussom.

Finally, I want to say thank you to you! You are the one who is making the time to learn Aussom, and it's you who make all of this worthwhile. Thank you for your patience and if you find anything incorrect or have any suggestions please reach out, I would love to hear from you.

Warm Regards,

Austin Lehman

austin@rosevillecode.com

Preface

Welcome! You are about to learn to write programs in the **Aussom** programming language, starting from the very beginning.

A programming language is how you give a computer instructions. With Aussom you will write those instructions in plain text, run them, and watch the computer do what you asked — print messages, do math, work with text and files, and much more. By the end of this book, you will be able to write real, useful programs of your own.

About This Book

This book is a hands-on guide. You will not just read about ideas, you will write real programs and watch them run. We start at the very beginning, with a one-line program that prints a greeting, and build up one step at a time. Each chapter adds a new piece of the language, so by the end you will understand how a complete program fits together.

You do not need to know Aussom. You do not need to have written a program before. We explain every new word the first time it shows up, and we walk through what each line of code does and why.

Who This Book Is For

This book is for **beginners**. It is written for people who are new to programming, including:

- Students and self-taught learners just getting started.
- People who are curious about coding but have never tried it.
- Developers who know another language and want a fast, friendly tour of Aussom.

If you are already an experienced developer who just needs to look up a type or a function, you may prefer the **API reference documentation** at aussom-lang.com, which lists everything built into the language in detail. This book takes the slower, friendlier path meant for learning.

What You Need

To follow along, you need three things, and the first chapters help you get them:

- **A computer** running Windows, macOS, or Linux.

- **The Aussom CLI** — the program that runs your Aussom code. Chapter 2 shows you how to install it.
- **A text editor** for writing code. Any editor works, but Chapter 3 sets up a free one with an Aussom plugin that makes writing code easier.

That's it. No prior setup and no paid tools are required.

How to Read This Book

Read the chapters **in order**. Each one builds on the chapter before it, so skipping around can leave gaps.

The best way to learn is to **type the examples yourself and run them**. Reading code is useful, but running it, breaking it, and fixing it is how the ideas stick. Do not worry if something does not fully click on the first read. Keep going, run the example, and it will make more sense in practice.

Conventions Used in This Book

A few simple conventions appear throughout the book:

- Code you write in Aussom looks like this:

```
c.log("Hello, Aussom!");
```

- Commands you type into a terminal look like this, and the program's output (if any) follows:

```
aussom hello.auss
```

- File names, type names, and short pieces of code inside a sentence are shown in `fixed-width` text, for example the `c.log` function or the `hello.auss` file.
- Screenshots show what you should see on your own screen when you run an example.

We speak directly to **you**, the reader, because you are the one writing these programs.

The Aussom Website and the Playground

Almost everything you need lives on the official website, aussom-lang.com. You will use its **Download** page to install Aussom, and its **Docs** page — including the searchable **API Reference** — to look up details as you grow.

The site also has a **Playground** at playground.aussom-lang.com, where you can try small snippets of Aussom right in your browser without installing anything. It's handy for quick experiments, though in this book we run everything locally with the CLI so you learn the real workflow.

Getting the Example Code

Every example in this book lives in a public code repository so you can read, run, and change it:

- **github.com/rsv-code/introduction-to-aussom-book-examples**

Each chapter has its own numbered directory in that repository. For example, Chapter 4's example is in `04-your-first-program`.

To keep the book readable, we usually show only the important parts of a program in the text and link to the full, runnable version in that repository. You are encouraged to download it and run it as you read.

Let's begin.

Part I

Getting Started

Chapter 1 - Meet Aussom

Welcome! You are about to learn a programming language called **Aussom**. By the end of this book you will be able to write real programs with it — programs that do math, work with text, read and write files, talk to the web, and more.

This first chapter does not ask you to write any code yet. Instead, it gives you the big picture: what Aussom is, the different forms it comes in, which form this book uses, and how Aussom takes the code you write and actually runs it. Getting this picture first will make everything that follows easier to understand.

What Aussom Is

Aussom is a **programming language**: a set of words and rules you use to write instructions for a computer. You type those instructions into a text file, and the computer carries them out.

Three words describe Aussom, and we will define each one because you will hear them a lot.

Interpreted. Some languages must be *compiled* first — translated all at once into a separate file the computer can run — before anything happens. Aussom is **interpreted** instead: a program called the *interpreter* reads your code and runs it directly, right away. There is no separate build step to wait for. You write a file, you run it, you see the result. This makes Aussom fast to experiment with, which is great when you are learning.

Object-oriented. Aussom is an **object-oriented** language, often shortened to *OOP* (object-oriented programming). This means Aussom lets you bundle your code into **objects** — self-contained units that hold some information and the actions that go with it — built from templates called **classes**. That's a powerful way to organize larger programs, and we build up to it slowly, starting in Part VI. Here's the reassuring part: **you don't need any of that to begin.** Aussom also lets you write short, direct **scripts** — just a few lines that run from top to bottom — and that's exactly where we'll start in Chapter 4. You'll add classes and objects later, once you have some code under your belt.

Runs on the JVM. Aussom runs on the **JVM**, short for *Java Virtual Machine*. The JVM is a widely used, well-tested foundation that knows how to run programs on Windows, macOS, and Linux. Aussom sits on top of it. The practical upside for you: Aussom behaves the same across those systems, and it can borrow from the huge world of existing Java tools when it needs to. You do **not** need to know Java to use Aussom — that is the whole point of this book.

One Language, Four Editions

Aussom comes in four **editions**. They are all the *same language* — the same words, the same rules, the same way of writing code — but each edition is packaged for a different job. Think of it like water: it is the same thing whether it comes in a bottle, a garden hose, or an ice cube tray; the container just fits the situation.

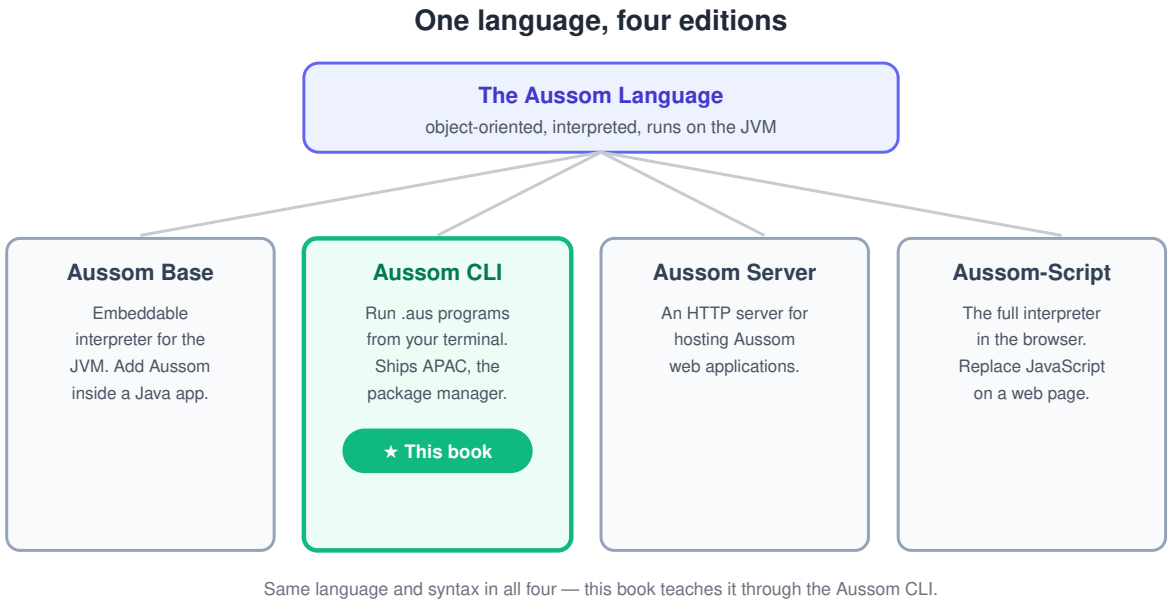


Figure 1: The four editions of Aussom and how they relate

Here is what each edition is for:

- **Aussom Base** — an *embeddable* interpreter for the JVM. "Embeddable" means other Java programs can tuck Aussom *inside* themselves and use it as a built-in scripting layer. This is a tool for people building larger Java applications.
- **Aussom CLI** — the *command-line* version. You run Aussom programs by typing a command in your terminal. It also comes with **APAC**, Aussom's package manager (a tool for installing add-on code others have written; more on that much later, in Part X). **This is the edition this book uses.**
- **Aussom Server** — an HTTP server for hosting Aussom *web applications*, so Aussom code can power a website.
- **Aussom-Script** — the full Aussom interpreter that runs *in a web browser*, as an alternative to JavaScript on a web page.

The term **CLI** stands for *command-line interface* — a way of using a program by typing commands as text, rather than clicking buttons in a window. If you have ever opened a "Terminal" or "Command Prompt" and typed a command, that is a command-line interface.

Why This Book Focuses on the Aussom CLI

Everything you learn about the language itself — types, math, text, loops, classes, files — is identical in all four editions. So we pick the one that is simplest to set up and best for learning: the **Aussom CLI**.

With the CLI you write a program in a plain text file, type one command to run it, and immediately see the output in your terminal. There is no web server to configure and no browser or Java project to wrap around your code. It is just you, your file, and the result — the clearest possible way to learn. Once you know the language through the CLI, moving to any other edition is easy, because the language is the same.

The Aussom Website: Your Home Base

Almost everything you need lives on the official website, **aussom-lang.com**. It is worth getting familiar with it now, because you will come back to it often. Two pages matter most while you are learning:

- **Download** — where you get the Aussom CLI installer for your computer, plus the editor plugins. You will use this page in Chapter 2 and Chapter 3.
- **Docs** — the documentation. This includes a Quick Start, a language guide, and the **API Reference**: a complete, searchable list of every built-in piece of Aussom. This book teaches you the language step by step; the API Reference is where you look up exact details once you know what you are looking for.

The site also has a **Playground** (at playground.aussom-lang.com) where you can try small snippets of Aussom in your browser without installing anything. It is handy for quick experiments, though in this book we will run everything locally with the CLI so you learn the real workflow.

How Aussom Runs Your Code

Let's trace what actually happens when you run an Aussom program, because it explains a lot about how you will work day to day.

You write your instructions in a plain text file. Aussom source files end in **`.aus`** or **`.auss`** — we'll start with `.auss` scripts in Chapter 4. You then hand your file to the Aussom interpreter — the `aussom` command you will install in the next chapter. The interpreter does two things: first it **reads** your file and checks that the code follows the language's rules; then it **runs** the code, one instruction at a time, doing whatever you asked. Anything your program prints shows up in your terminal.

Notice there is **no separate compile step**. You do not build your program into another file and then run that. You run the source file itself, directly. Change a line, run it again, see the new result — that quick loop is what makes an interpreted language so pleasant to learn with.

If your code breaks a rule — a missing symbol, a typo — the interpreter tells you during the "read" step and stops before running anything. If something goes wrong while the program is running, it reports that too. You will see real examples of both in Chapter 4, and you will learn that these messages are helpful, not scary.

Figure 2: Your source file goes into the interpreter, which reads then runs it, and the output appears in your terminal

Where We Go From Here

You now have the map. Aussom is an interpreted, object-oriented language that runs on the JVM; it comes in four editions; and we are using the CLI edition because it is the clearest way to learn. You write your instructions in a text file, and the `aussom` interpreter reads and runs it directly — starting with the simple `.auss` scripts we'll write next.

In **Chapter 2**, you will install the Aussom CLI on your computer and confirm it works. In **Chapter 3**, you will set up a code editor so writing Aussom is comfortable. Then, in **Chapter 4**, you will write and run your very first Aussom program. Let's get you set up.

Chapter 2 - Installing the Aussom CLI

To write and run Aussom programs on your own computer, you need the **Aussom CLI** installed. This chapter walks you through it from start to finish: finding the right download, installing it on your operating system, and confirming it works. It takes just a few minutes.

We will do everything through the official website, **aussom-lang.com**, because that is where the real, up-to-date installers live.

Getting Around the Aussom Website

Open a web browser and go to **aussom-lang.com**. Across the top you will see a few links. Two of them are all you need right now:

- **Download** — the installers and plugins. This is where we are headed in a moment.
- **Docs** — the documentation, including the **API Reference** you'll consult later to look up exact details of anything built into the language.

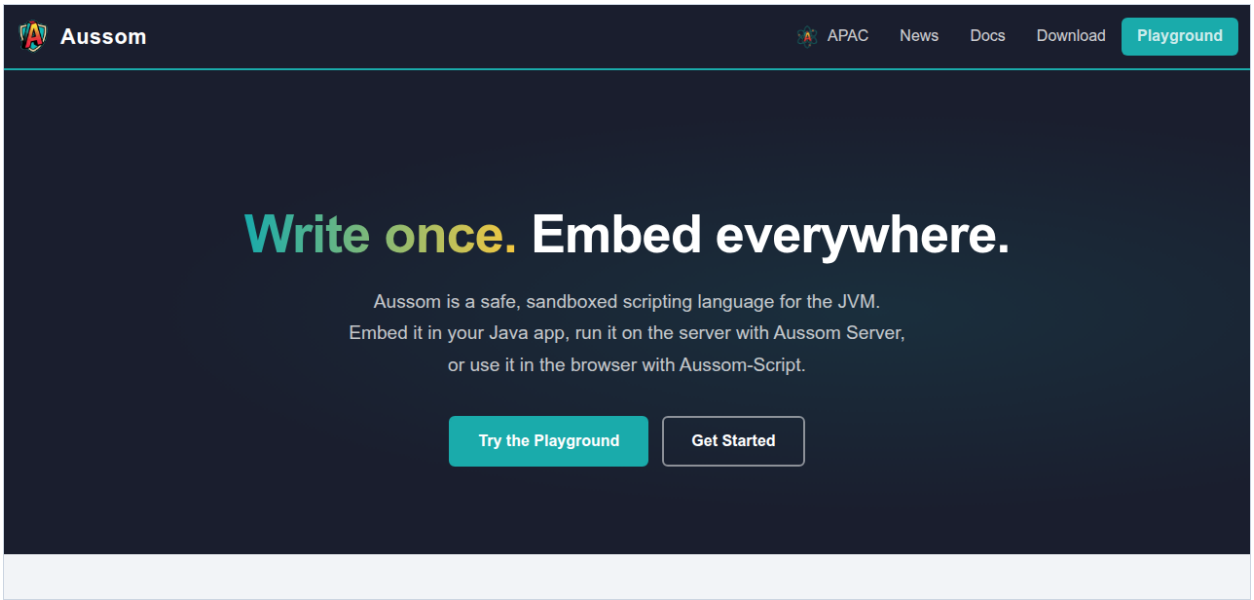


Figure 3: The Aussom home page, with Docs and Download in the top navigation bar

Click **Download**. The page is organized into sections. The first section is the **Aussom CLI** — that's the one we want. It offers a download button for each operating system. Below it you'll also find

the editor plugins (we install those in Chapter 3) and the other editions from Chapter 1.

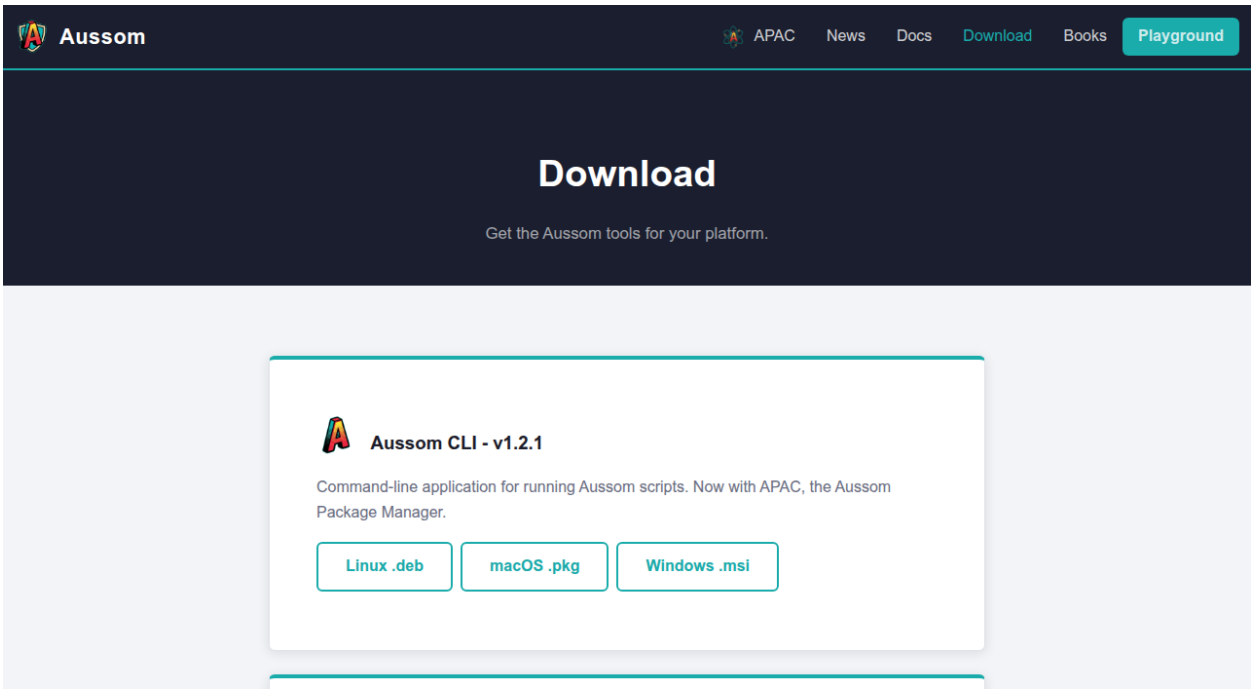


Figure 4: The Download page, showing the Aussom CLI section with Linux, macOS, and Windows installers

Downloading the Right Installer for Your System

The Aussom CLI section offers a separate installer for each operating system. Pick the one that matches your computer:

- **Linux** — a `.deb` file (for example, `aussom-1.2.1-amd64.deb`).
- **macOS** — a `.pkg` file (for example, `aussom-1.2.1.pkg`).
- **Windows** — an `.msi` file (for example, `aussom-1.2.1.msi`).

The number in the file name is the **version** (1.2.1 at the time of writing). Yours may be higher — that's fine; always grab the current one. Click your system's installer to download it, then follow the matching section below.

> The Aussom CLI installer also includes **APAC**, Aussom's package manager, and a set of documentation you can open from the command line. You get all of it in this one install; we'll use APAC much later, in Part X.

Installing on Linux (the `.deb` Installer)

The `.deb` file is a standard package for Debian-based Linux systems such as Ubuntu. Open a **terminal**, move to the folder where the file downloaded (usually your `Downloads` folder), and install it with `dpkg`. The `sudo` in front means "run this as an administrator," so your system may ask for your password.

```
cd ~/Downloads
sudo dpkg -i aussom-1.2.1-amd64.deb
```

Replace the file name with the exact one you downloaded. When it finishes, the `aussom` command is installed and ready. Skip ahead to *Checking Your Install*.

Installing on macOS (the `.pkg` Installer)

The `.pkg` file is a standard macOS installer that walks you through a few short steps. **Double-click** the downloaded file in Finder to start it.

The installer opens with a welcome screen. Click **Continue**.

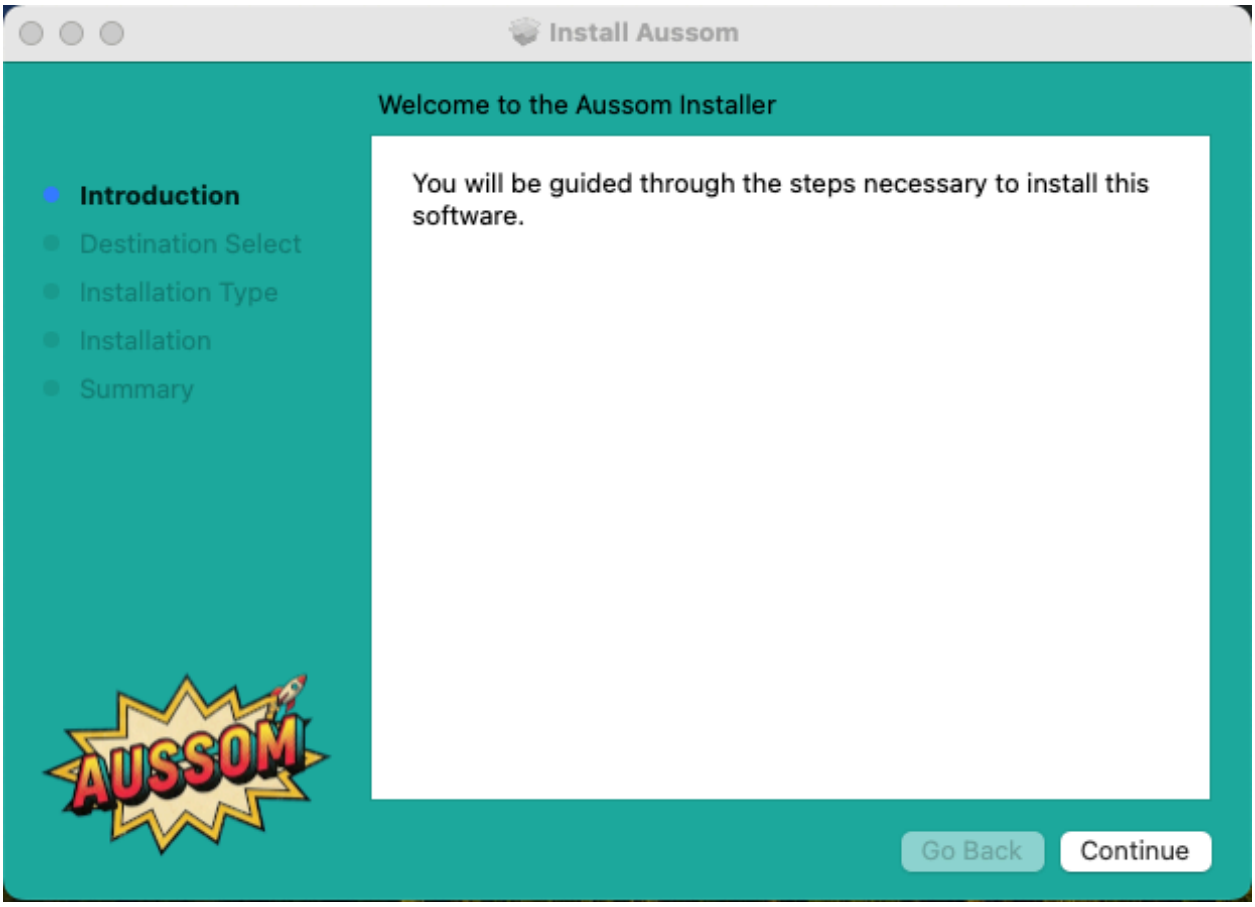


Figure 5: The macOS installer's welcome screen

Next it shows how much space the install needs and where it will go. Click **Install** to begin. macOS may ask for your password to approve the installation — enter it, just like installing any other Mac app.

The installer copies its files. This takes only a moment.

When you see "The installation was successful," you're done. Click **Close**.

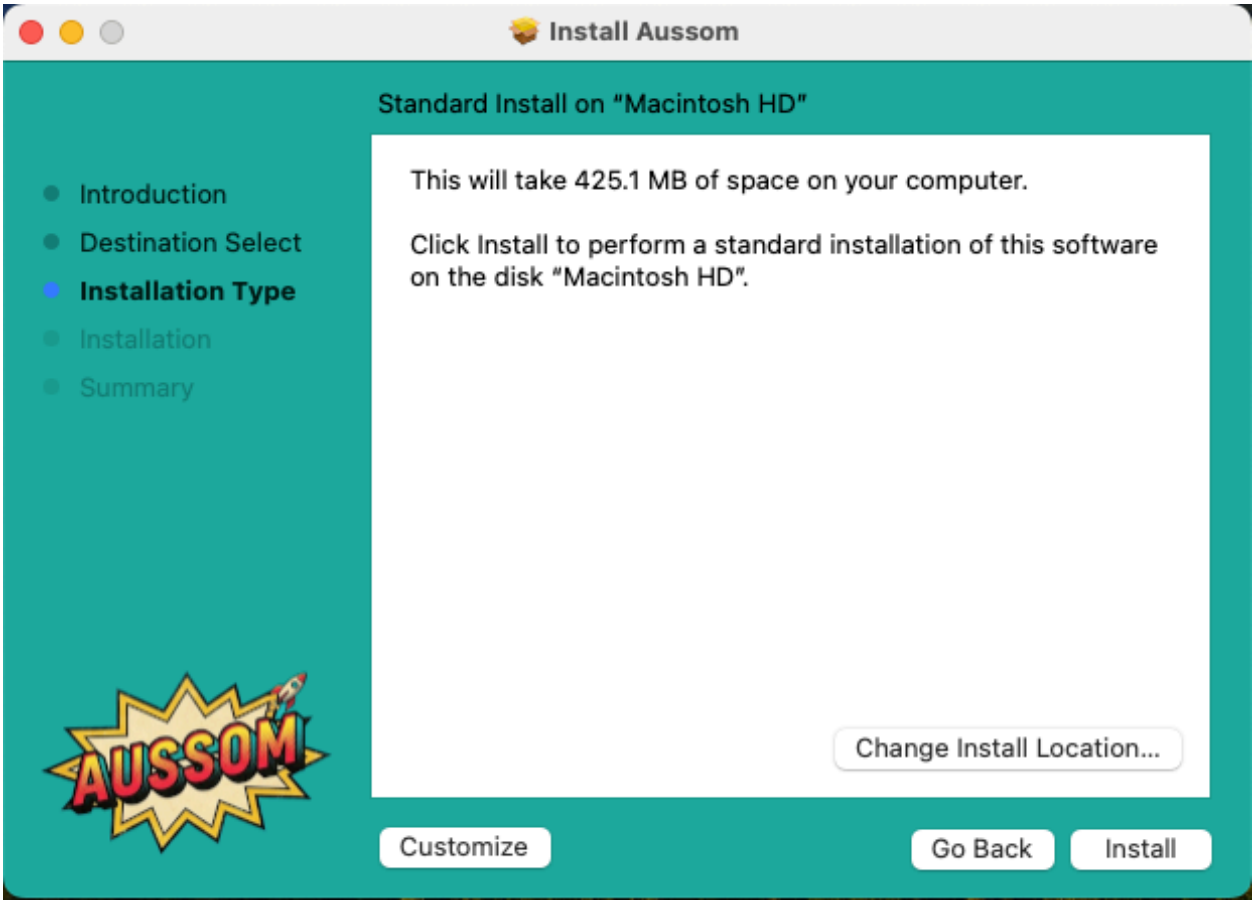


Figure 6: Confirming a standard installation

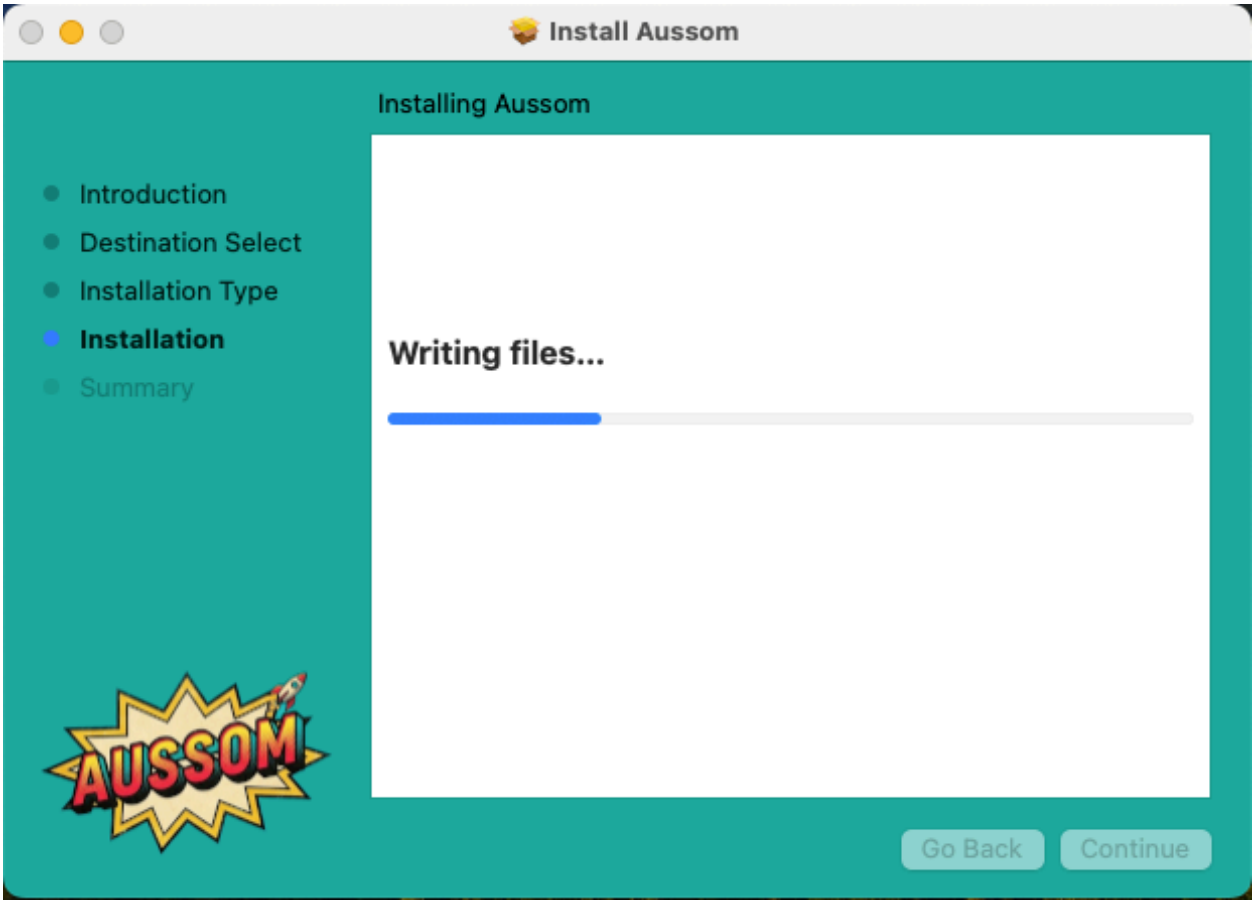


Figure 7: The installer writing files

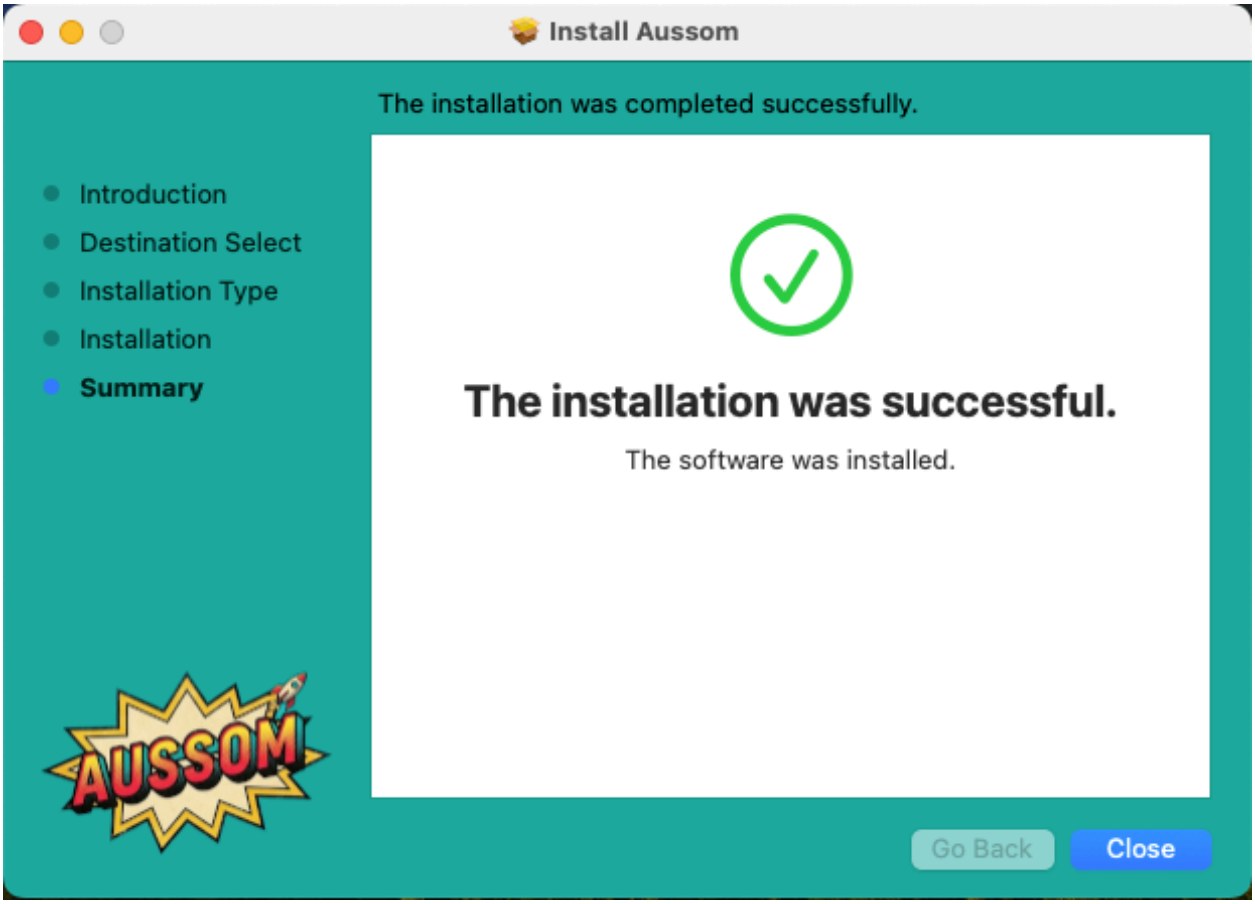


Figure 8: The successful installation screen

The `aussom` command is now installed. Open the **Terminal** app (find it in *Applications* → *Utilities*, or search for "Terminal" with Spotlight) and continue to *Checking Your Install*.

Installing on Windows (the `.msi` Installer)

The `.msi` file is a standard Windows installer. **Double-click** the downloaded file to start it. (If Windows shows a security prompt asking whether to allow the app to make changes, choose **Yes**.)

First, the installer shows the license agreement. Check **I accept the terms in the License Agreement**, then click **Install**.

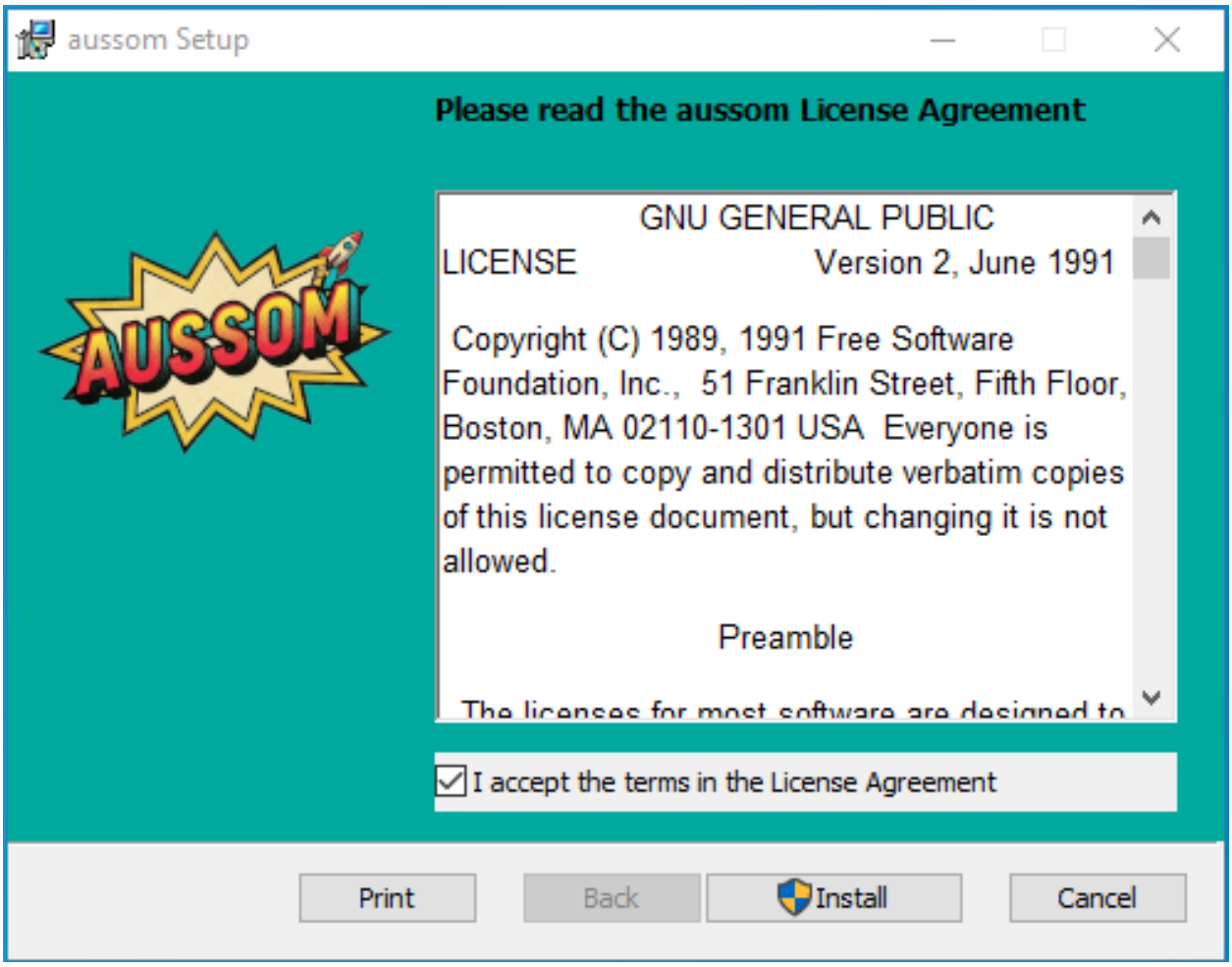


Figure 9: Accepting the license agreement

The Setup Wizard now installs Aussom, copying its files and registering the program. Let the progress bar run to the end — it takes only a moment.

When it's done, you'll see "Completed the aussom Setup Wizard." Click **Finish**.

The `aussom` command is now installed and available from any command window. Open a terminal — **Windows Terminal**, **PowerShell**, or **Command Prompt** all work — by searching for one in the Start menu, then continue to *Checking Your Install*.

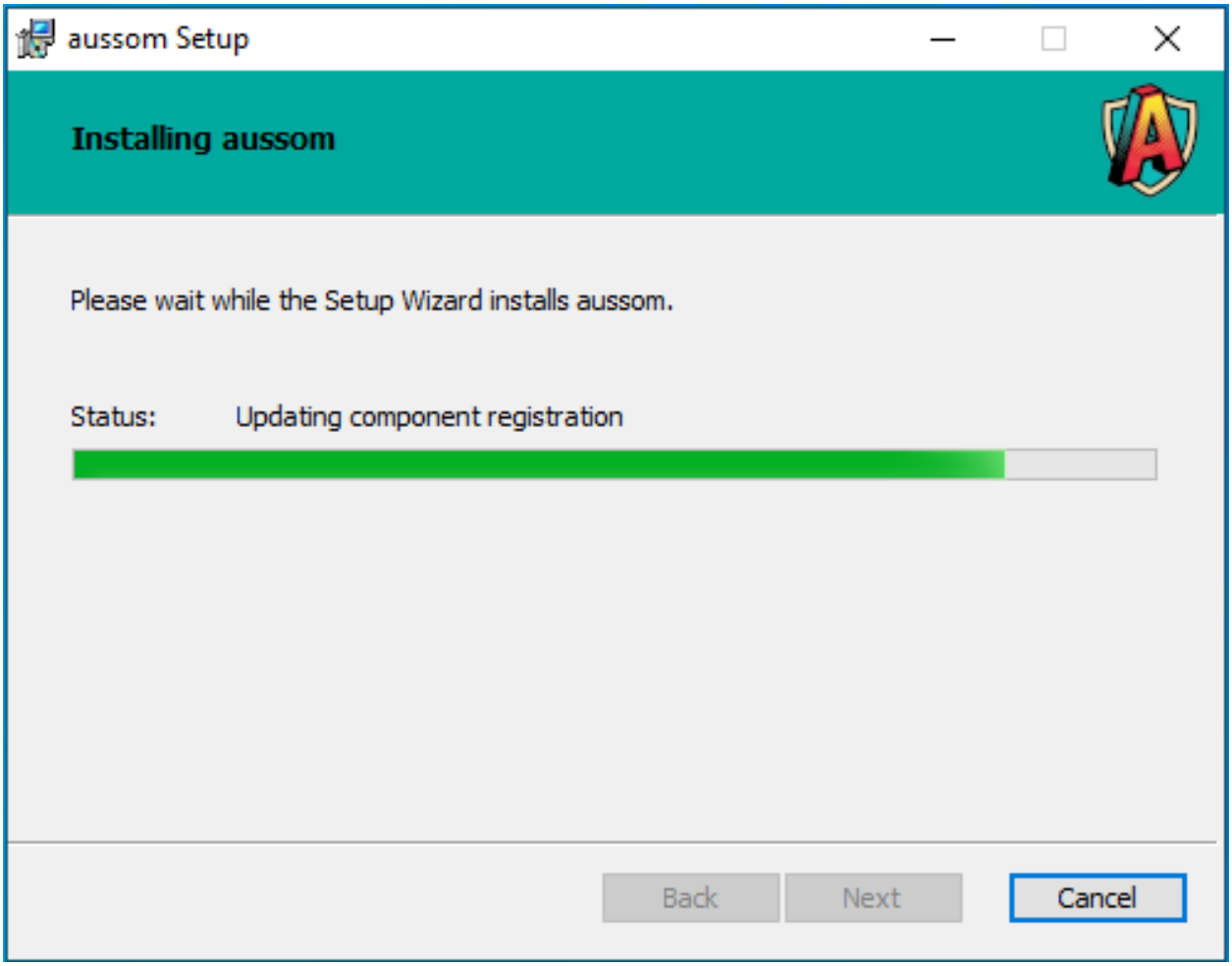


Figure 10: The Setup Wizard installing Aussom

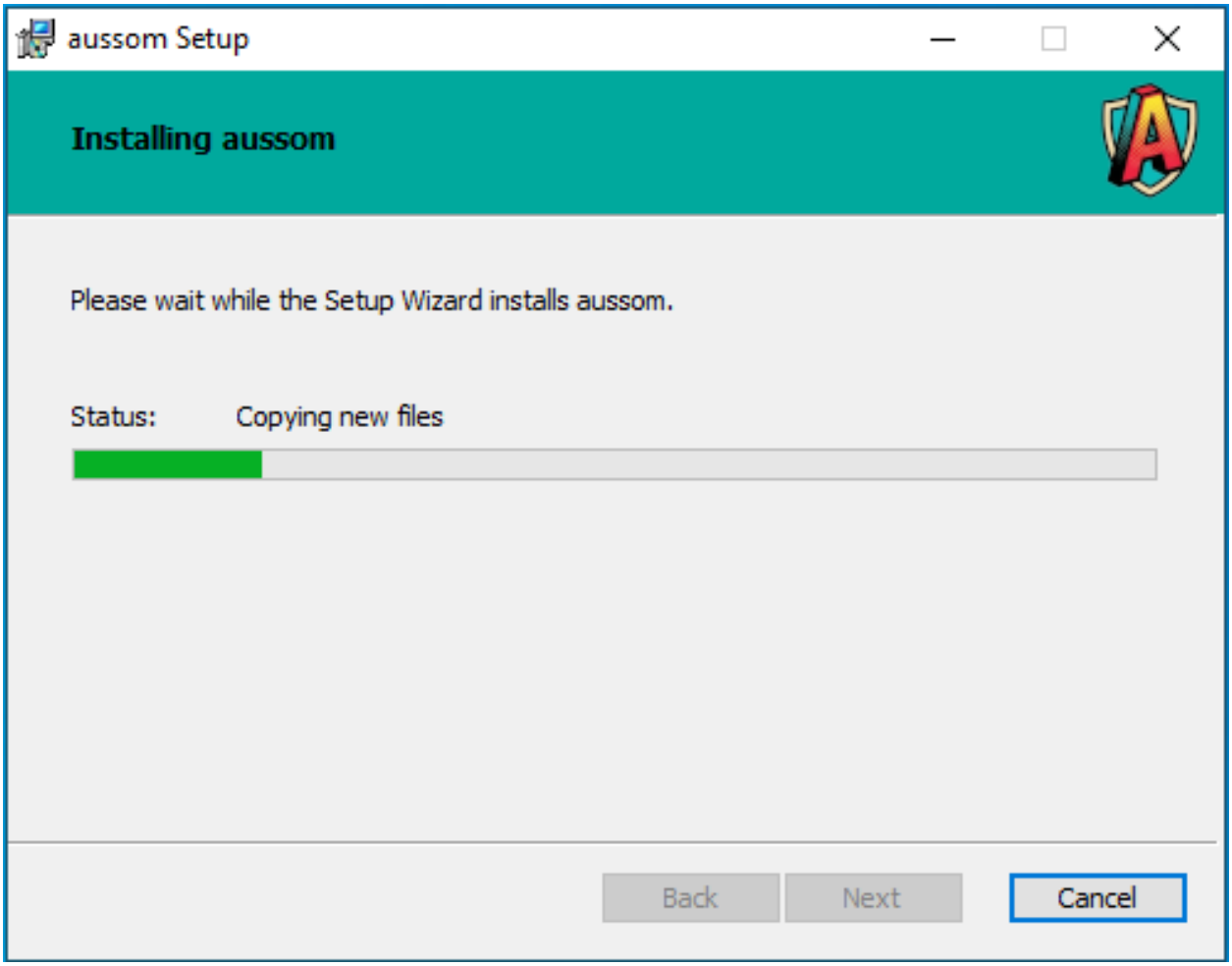


Figure 11: Installation continues until the progress bar is full

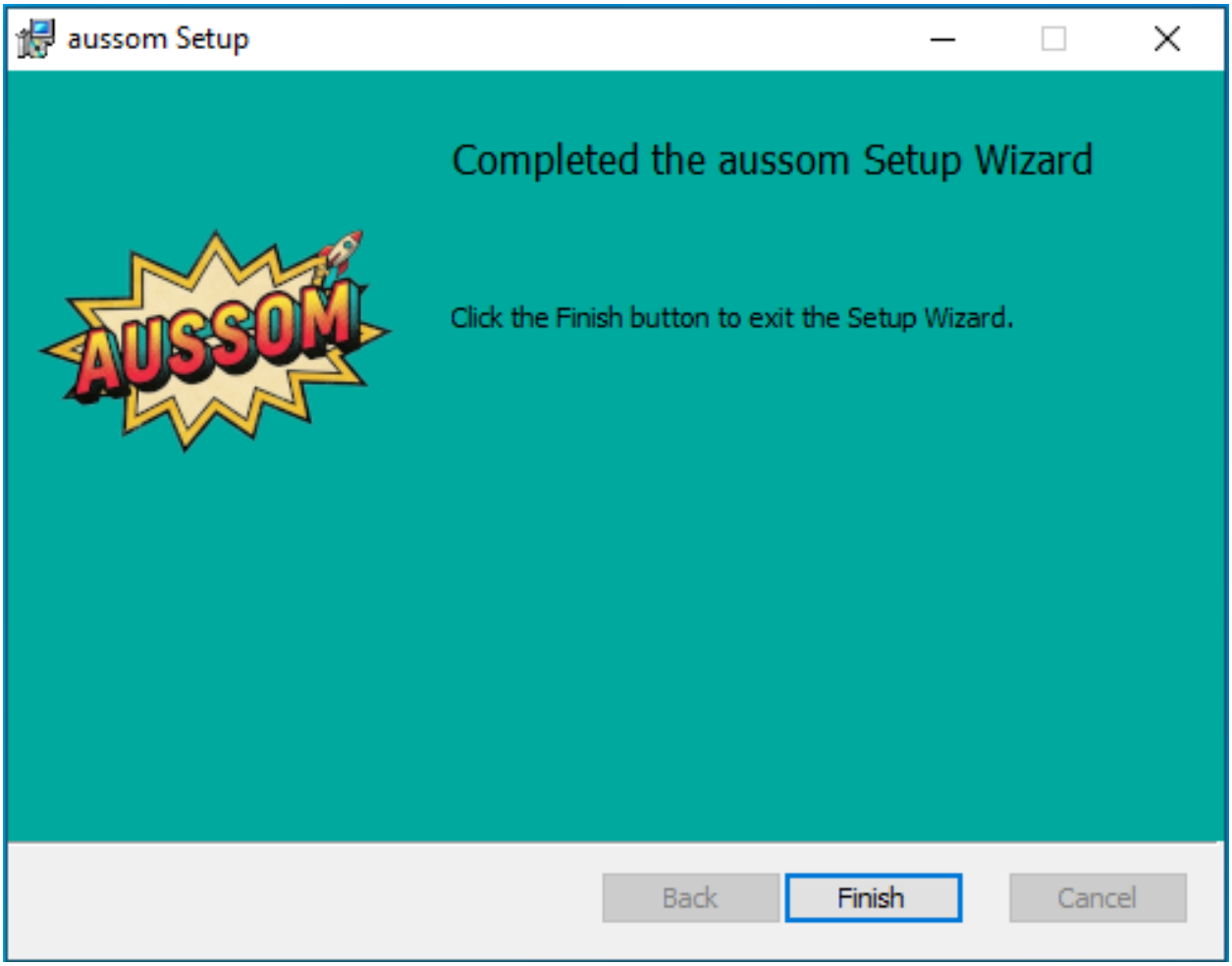


Figure 12: The completed Setup Wizard

Checking Your Install

No matter which system you're on, the test is the same. In your terminal, type this and press Enter:

```
aussom -v
```

The `-v` flag means "print the version." If Aussom is installed correctly, you'll see version and license information, something like this:

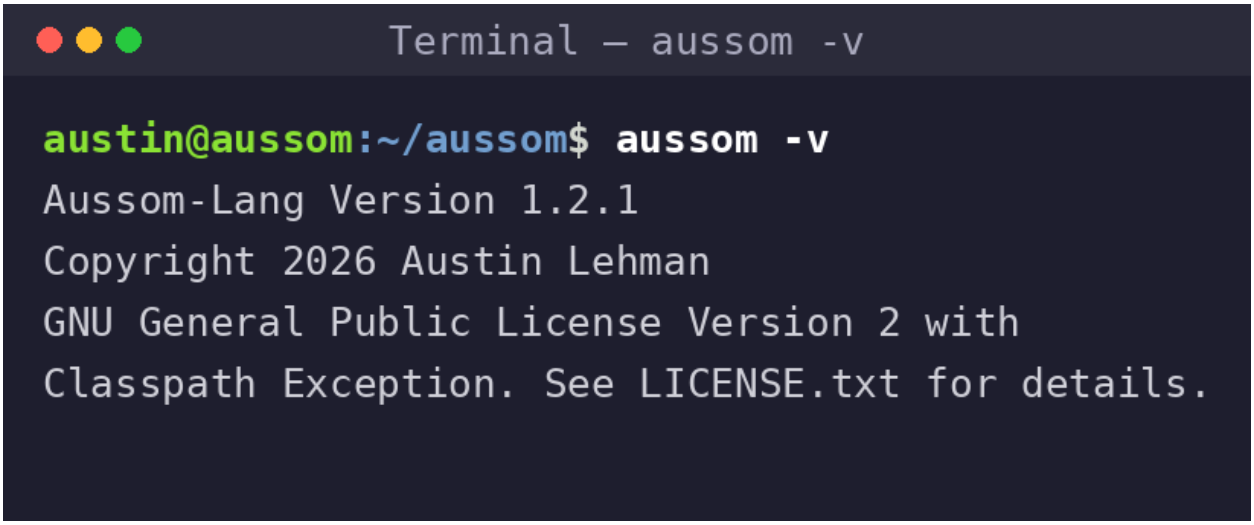


Figure 13: Running aussom -v prints the version and license information

If you see that, **congratulations — Aussom is installed and working.** If instead you get an error like "command not found," the terminal can't find the `aussom` command yet. The most common fix is to fully close the terminal and open a new one, so it picks up the freshly installed command. (On Windows, this is especially common right after installing.)

Seeing What the CLI Can Do

The Aussom command has a built-in help listing. Run:

```
aussom -h
```

The `-h` flag ("help") prints every option the `aussom` command accepts:

```
usage: aussom [options] <aussom-file>
 -d,--doc                       generate aussomdoc for file
 -db,--debug                    start the DAP server and run the script
                                under the debugger
    --debug-log <file>          log every DAP JSON-RPC message to this file
                                (for debugging the debugger)
    --debug-no-wait             do not wait for configurationDone before
                                running the script
    --debug-port <port>         DAP transport: listen on this TCP port
                                instead of using stdio
    --debug-suspend-on-entry    pause at the first executable line before
```

```
                                user code runs
   --exclude-tags <list>        comma-separated tag list; skip @Test
                                methods whose tags include at least one
                                match (overrides --tags)
 -h,--help                      print this message
 -jl,--jar-location             print the absolute path of this CLI's jar
                                and exit
 -l,--loglevel <arg>            set the log level (trace, debug, info,
                                warning, error)
 -o,--outdir <arg>              the output directory to place the generated
                                aussomdoc file
 -od,--opendoc                  opens the Aussom documentation in the
                                system browser
 -s,--scripting                 run the script file in script mode
                                (top-level statements allowed)
 -t,--test                      run tests for file
 -ta,--test-all                 run tests for all classes loaded in the
                                engine
   --tags <list>                comma-separated tag list; only run @Test
                                methods whose tags include at least one
                                match (untagged tests are excluded)
 -v,--version                   print the version information
```

That's a lot of options, and you do **not** need to understand them yet. The `<aussom-file>` at the top is the important part: it shows the normal way to use the command — `aussom` followed by the name of a file to run, which is exactly what you'll do in Chapter 4. A few flags you'll meet as the book goes on:

- **`-v`** — print the version (you just used it).
- **`-h`** — print this help message.
- **`-od`** — open the documentation in your browser (next section).
- **`-s`** — run a file in *script mode* (Chapter 28).
- **`-t`** — run tests in a file (Chapter 34).
- **`-ta`** — run tests in *all* loaded classes, across included files (Chapter 34).

The rest — debugging, doc generation, test tags — are for later or for advanced use. You can always run `aussom -h` to remind yourself what's available.

Opening the Built-in Docs

The install comes with a full copy of the documentation, and the CLI can open it in your web browser for you:

```
aussom -od
```

The `-od` flag ("open docs") launches the documentation page in your default browser. This is handy when you're offline or just want the reference close at hand. It's the same material you'll find under **Docs** on the website.

Updating and Uninstalling

Updating. When a newer version of Aussom comes out, the reliable way to move to it is to **uninstall the version you have first, then install the new one fresh** using the steps earlier in this chapter. Running a newer installer on top of an existing install isn't the supported path, so uninstall-then-reinstall is the recommended routine.

Uninstalling on Linux. Use your package manager:

```
sudo apt-get purge aussom
```

Uninstalling on macOS and Windows. Remove Aussom the same way you remove any other installed program (for example, through *Add or Remove Programs* on Windows).

One reassurance for later: uninstalling the CLI removes Aussom itself but **keeps any global libraries you've installed with APAC** (Aussom's package manager, covered in Part X). Those add-on packages are stored separately and are left in place, so when you reinstall, they're still there.

You're Ready

Aussom is installed, and `aussom -v` proves it. You could start writing programs right now with any plain text editor. But a little setup makes writing Aussom much more pleasant — colored code, instant error checking, and a run button. In **Chapter 3**, we'll set up an editor with the official Aussom plugin. If you'd rather jump straight to writing code, you can skip ahead to **Chapter 4** and come back for the editor setup anytime.

Chapter 3 - Setting Up Your Editor

You write Aussom programs in plain text, so technically any text editor works. But a good editor with the **Aussom plugin** installed makes the job far more pleasant. It colors your code so it's easy to read, underlines mistakes as you type, suggests what you might write next, and gives you a button to run your program. This chapter sets that up.

If you'd rather start coding immediately with a basic editor, you can skip to Chapter 4 and return here later. But most people find the plugin worth the five minutes it takes.

Choosing an Editor

Aussom has an official plugin for two popular, free editors:

- **IntelliJ IDEA** — a powerful development environment from JetBrains. Its free "Community Edition" is plenty for everything in this book.
- **Visual Studio Code (VS Code)** — a fast, lightweight, free editor from Microsoft that's popular with beginners and professionals alike.

Either one is a great choice. If you already use one of them, stick with it. If you're starting fresh and unsure, **VS Code** is a common first pick because it's light and simple. In this book the screenshots will be from IntelliJ because it's the editor I use but either one will work well. Install your chosen editor from its own website first (jetbrains.com for IntelliJ, code.visualstudio.com for VS Code), then come back here to add the Aussom plugin.

Downloading the Plugins from the Aussom Website

Both plugins come from the same **Download** page you used in Chapter 2 (on **aussom-lang.com**). Scroll past the Aussom CLI section to **Editor Plugins**. You'll see two downloads:

- **IntelliJ IDEA Plugin** — download the file for IntelliJ.
- **VS Code Plugin** — download the file for VS Code.

Download the one that matches your editor and remember where it saved (usually your `Downloads` folder). You install it *from that file*, using the steps below.

Installing the IntelliJ IDEA Plugin

IntelliJ can install a plugin straight from a file on disk:

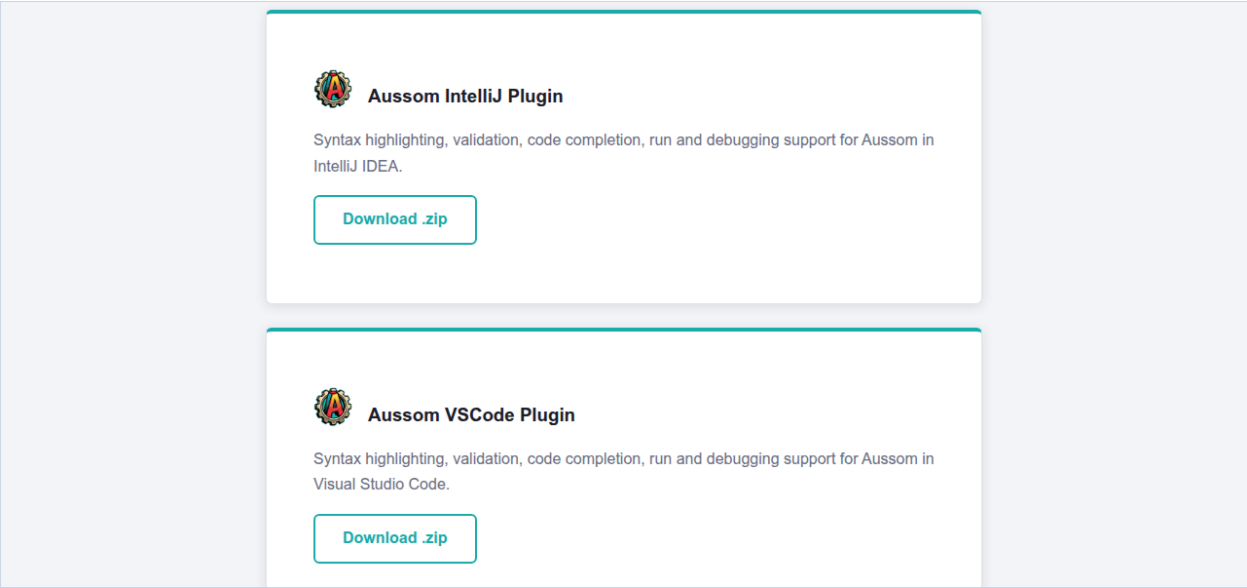


Figure 14: The Editor Plugins on the Download page: the IntelliJ IDEA and VS Code plugin downloads

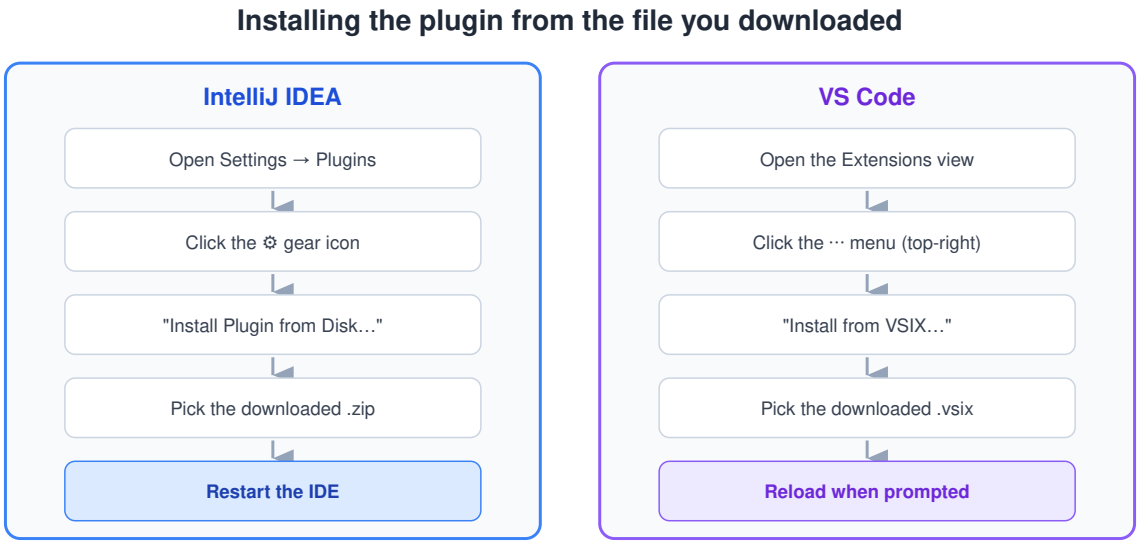


Figure 15: Installing the plugin from the file you downloaded, in IntelliJ or VS Code

1. Open IntelliJ and go to **Settings** (on macOS, **IntelliJ IDEA → Preferences**).
2. Select **Plugins** in the left-hand list.
3. Click the **gear icon** near the top of the Plugins panel.
4. Choose **Install Plugin from Disk…**
5. Find and select the plugin file you downloaded, then confirm.
6. When prompted, **restart the IDE** so the plugin loads.

After the restart, IntelliJ recognizes `.aus` files and the Aussom features turn on automatically.

Opening and Running a Script in IntelliJ

With the plugin installed, running an Aussom file takes one click. Open a project or folder that contains your scripts — below we've opened the book's example project — and double-click a script in the **Project** panel to open it. Here `hello.auss` is open and has just been run:

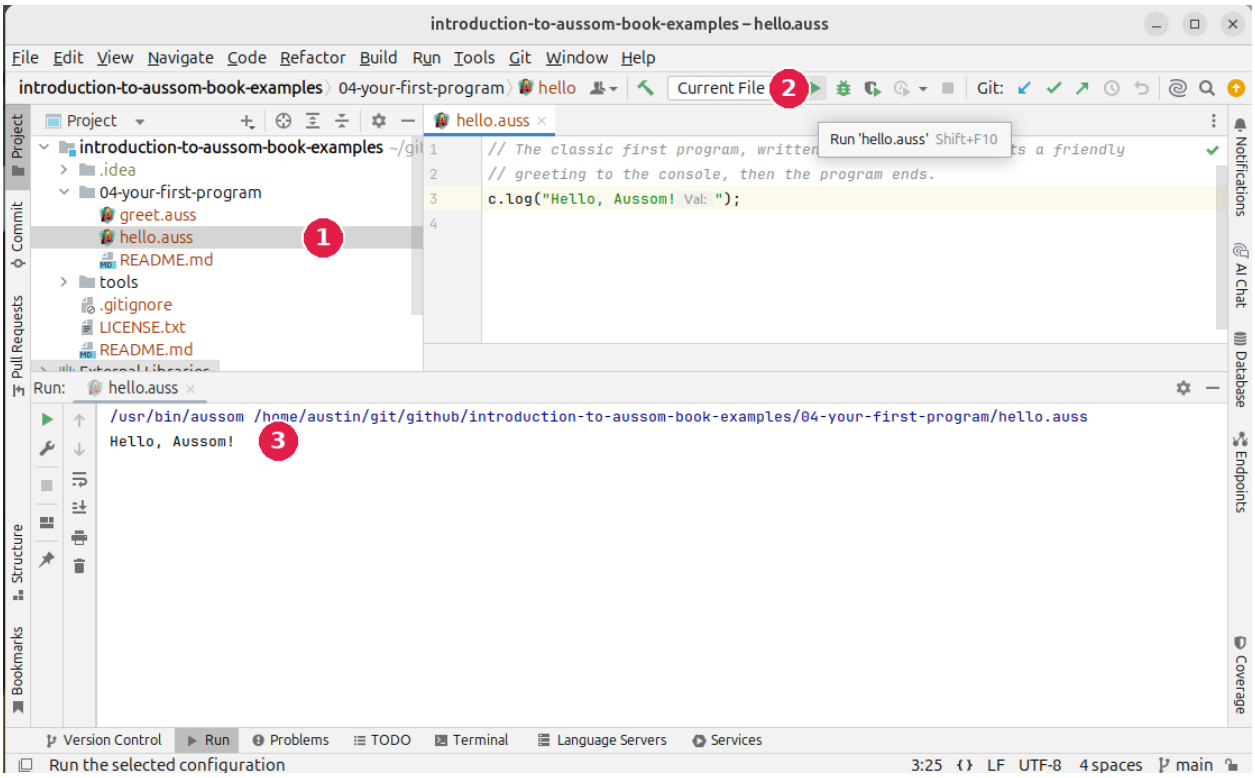


Figure 16: Opening and running hello.auss in IntelliJ IDEA

1. **The open script.** `hello.auss` is open in the editor. Any `.aus` or `.auss` file opens the same way — double-click it in the Project panel on the left.
2. **The Run button.** With a script open, click the green **Run** arrow in the toolbar. Hovering over it shows a tooltip — here, *Run 'hello.auss'* — so you know exactly what it will run.
3. **The output.** The program runs and its output appears in the **Run** panel at the bottom. Here you can see our greeting, `Hello, Aussom!`.

You can also **right-click the script file** — in the Project panel or in the editor — and choose **Run** from the context menu. It does the same thing as the Run button, and it's often the quickest way.

Installing the VS Code Plugin

VS Code installs a plugin from a file using its "VSIX" option (a VSIX is simply the file format VS Code plugins come in):

1. Open VS Code and click the **Extensions** icon in the left sidebar (it looks like four small squares).
2. Click the **`...` (More Actions)** menu at the top-right of the Extensions panel.
3. Choose **Install from VSIX...**
4. Find and select the plugin file you downloaded, then confirm.
5. If VS Code asks, **reload** the window.

After that, VS Code recognizes `.aus` files and the Aussom features turn on automatically.

Opening and Running a Script in VS Code

Running a script in VS Code works much the same way. Open a folder that contains your scripts, then click a script in the **Explorer** to open it. Here `hello.auss` is open and has just been run:

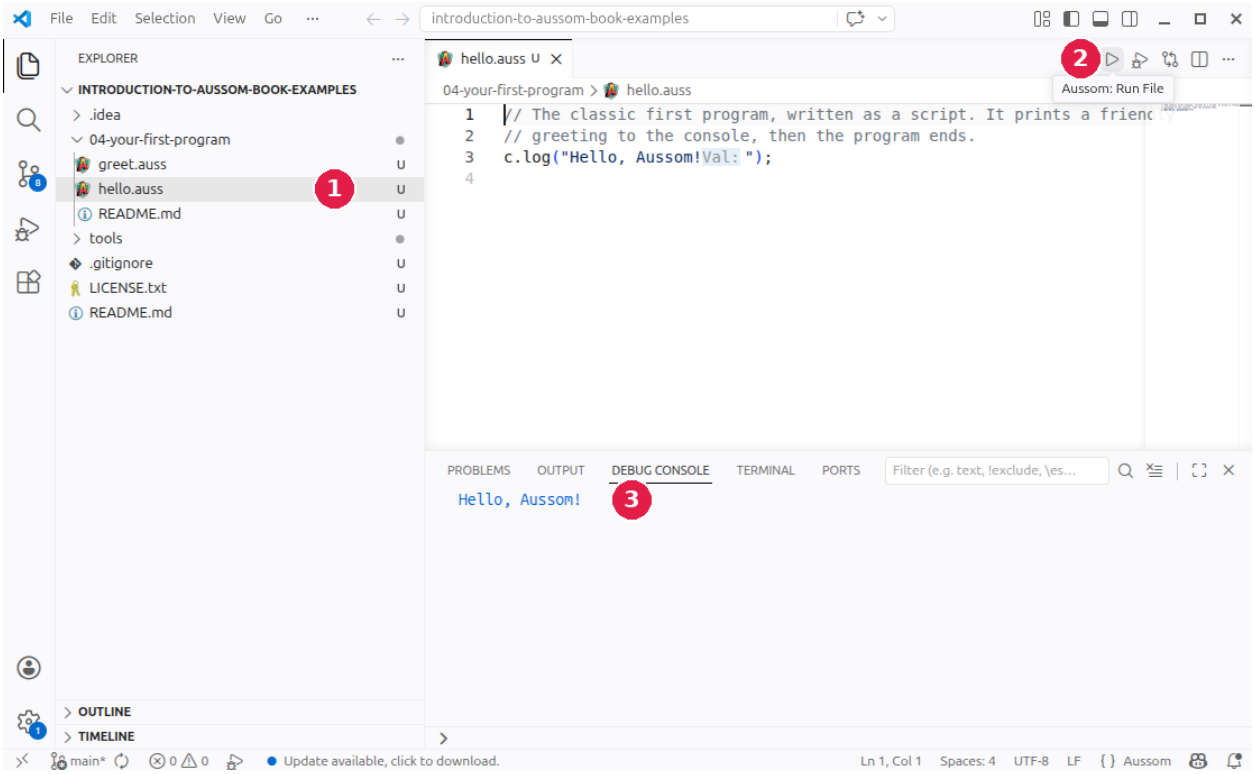


Figure 17: Opening and running hello.auss in VS Code

1. **The open script.** `hello.auss` is open in the editor. Open any `.aus` or `.auss` file the same way — click it in the Explorer on the left.
2. **The Run button.** With a script open, click the **Run** arrow at the top-right of the editor. Hovering over it shows a tooltip — *Aussom: Run File* — confirming what it does.
3. **The output.** The program runs and its output appears in the **Debug Console** panel at the bottom. Here you can see our greeting, `Hello, Aussom!`.

As in IntelliJ, you can also **right-click the script file** — in the Explorer or the editor — and pick **Run** from the context menu.

> **If your editor says it can't find Java:** the plugins include a helper (a "language server") that runs on Java, the same platform the Aussom CLI uses. Installing the Aussom CLI in Chapter 2 normally covers this. If the editor still reports it can't find Java, install a Java runtime (a JDK, version 17 or newer) and restart the editor.

What the Plugins Give You

Once the plugin is active, open or create a file ending in `.aus` and you'll notice the difference right away. Both plugins provide the same core features:

- **Syntax highlighting** — different parts of your code (keywords, text, names) appear in different colors, which makes code far easier to read and helps you spot mistakes.
- **Validation (diagnostics)** — the editor checks your code as you type and underlines problems, so you catch a typo or a missing symbol *before* you even run the program.
- **Code completion** — as you type, the editor suggests names and finishes them for you, so you type less and misspell less.
- **Run** — a run control lets you launch the current file without switching to a terminal.
- **Debugging** — for later, when your programs grow: you can pause a running program, step through it line by line, and inspect values. We don't need this yet, but it's there when you do.

The plugins also help with **doc comments** (the `/** ... */` comments you'll see in the examples): typing `/**` expands into a comment skeleton, and pressing Enter continues it. You'll learn what doc comments are for in Chapter 20.

Ready to Write Code

Your editor now understands Aussom, and the CLI from Chapter 2 can run it. That's the whole toolkit. In **Chapter 4**, you'll finally write and run your first Aussom program — and understand every line of it.

Chapter 4 - Your First Program

It's time to write and run your first Aussom program. It's a small one — it prints a greeting — but you'll understand *every single character* of it by the time you finish. That's a real milestone.

You'll need the Aussom CLI from Chapter 2. An editor with the plugin from Chapter 3 is nice but not required; a plain text editor is fine.

Writing `hello.auss`

Create a new file named **`hello.auss`** and type in this one line:

```
c.log("Hello, Aussom!");
```

That's the whole program. Save the file.

Notice the file's extension: **`.auss`** (with a double "s"). This tells Aussom to run the file as a **script** — a plain list of instructions that Aussom carries out from top to bottom, one after another. Scripts are the simplest way to write Aussom, which makes them the perfect place to start.

> **A note on the two ways to write Aussom.** Aussom has a second style, using files that end in `.aus`, where you organize your code into *classes* (remember from Chapter 1 that Aussom is object-oriented). That style is great for larger programs, and you'll learn it later once you have some code under your belt. For now, and for a good while, we'll stick with simple `.auss` scripts.

Before we run it, let's understand what we wrote.

Understanding the Line

Here is our one line with every part labeled:

Let's take it piece by piece:

- **`c`** is the **console** — Aussom's connection to your terminal. It's always available in any program; you don't have to set it up or import anything first.
- **`.log(...)`** calls the console's `log` action, which prints whatever you put in the parentheses, followed by a new line.
- **`"Hello, Aussom!"`** is a **string** — a piece of text. In Aussom, text goes inside double quotes. The quotes mark where the text starts and stops; they aren't printed themselves.

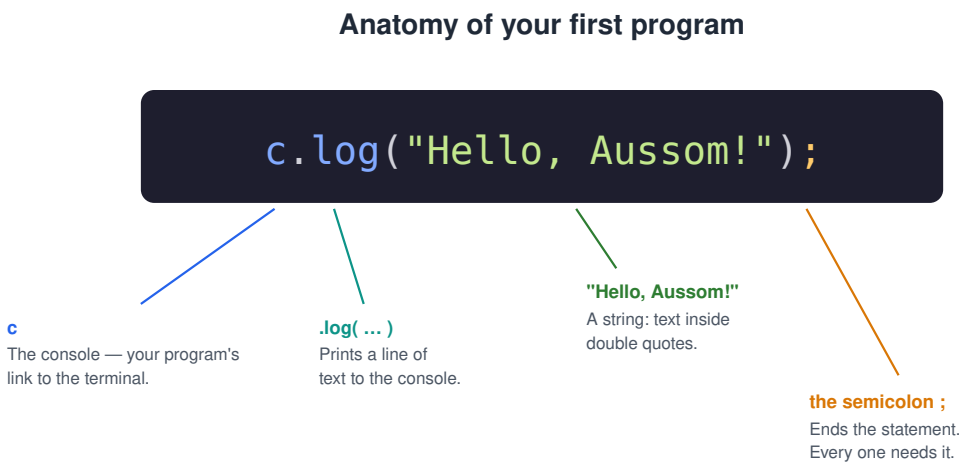


Figure 18: A labeled breakdown of c.log("Hello, Aussom!"); — the console, the log call, the string, and the semicolon

- **;** is a **semicolon**. In Aussom, every *statement* — every single instruction — ends with one. It's how you tell Aussom "this instruction is finished." Forgetting it is the most common beginner slip, and we'll see exactly what that looks like at the end of the chapter.

Put together, the line means: *"Console, print the text `Hello, Aussom!`."*

Running Your Program with `aussom`

Open your terminal and move to the folder where you saved `hello.auss`. For example, if it's in a folder called `aussom` in your home directory:

```
cd ~/aussom
```

Then run it by typing `aussom` followed by the file name:

```
aussom hello.auss
```

Aussom reads the file and runs it. You'll see:

Figure 19: Running hello.auss prints "Hello, Aussom!"

That's it — you just wrote and ran a real program. Change the text between the quotes, save, and

run it again to see your new message. That quick write–run–see loop is how you'll work throughout this book.

> **Running from the terminal vs. the IDE.** Typing `aussom <file>` in the terminal is how you run a program from the command line, and it's the convention we'll use throughout the rest of the book — it works the same on every system and makes the commands easy to show. If you'd rather run your programs *inside* your editor, that works just as well: use the Run button or the right-click **Run** option described in Chapter 3. Pick whichever you prefer; the result is the same.

You can find this ready-to-run example, and the next one, here: `04-your-first-program`.

Printing to the Console with `c.log`

You just used `c.log`, and it's the tool you'll reach for most while learning: it prints a line of text. But the console can do more than `log`. When you want them, these related actions are available:

- **`c.log`** — print a line of text (what we're using).
- **`c.println`** — also prints a line; works just like `log`.
- **`c.print`** — prints *without* moving to a new line, so the next output continues on the same line.
- **`c.info`** — prints a line tagged with an `[info]` label.
- **`c.warn`** — prints a line tagged with a `[warning]` label.
- **`c.err`** — prints a line tagged with an `[error]` label.

The tagged versions (`info`, `warn`, `err`) help you tell different kinds of messages apart. For example, this script:

```
c.log("a plain line");
c.info("some information");
c.warn("a warning");
c.err("an error");
```

prints:

```
a plain line
[info] some information
[warning] a warning
[error] an error
```

For now, `c.log` is all you need.

Reacting to Input with `args`

A program is more useful when it can respond to what the person running it types. When you run a script, Aussom hands it a built-in list called **`args`** — the words you type on the command line *after* the file name. Let's use it.

Create a second file, **`greet.auss`**:

```
if (#args > 0) {
    c.log("Hello, " + args[0] + "!");
} else {
    c.log("Hello there! Tip: type your name after the file name.");
}
```

There are a few new pieces here. Each gets a full chapter later, but here's enough to follow along:

- **`#args`** — the # symbol gives the **length** of a list, so `#args` is how many words were typed. (Lists are Chapter 10.)
- **`if (...) { ... } else { ... }`** — a **decision**: run the first block if the condition is true, otherwise run the `else` block. Here the condition is `#args > 0`, meaning "at least one word was typed." (Decisions are Chapter 12.)
- **`args[0]`** — reads the **first** item from the list. Aussom counts from zero, so item `0` is the first word. (More in Chapter 10.)
- **`"Hello, " + args[0] + "!"`** — the + here **joins** strings together, so if `args[0]` is `Ada`, this builds `"Hello, Ada!"`. (Strings are Chapter 8.)

Run it with your name, then again with nothing, to see both paths:

```
aussom greet.auss Ada
aussom greet.auss
```

The output:

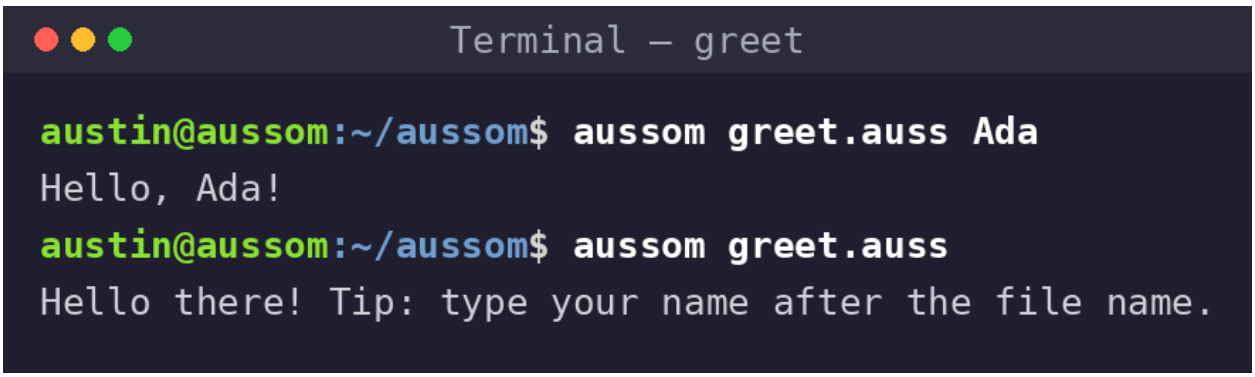


Figure 20: greet.auss greets Ada when given a name, and gives a tip when not

Your program now reacts to what the user typed. That's a big step up from a fixed message.

Reading Error Messages

Everyone makes mistakes while coding — constantly. The good news is that Aussom tells you when something is wrong and points you toward the problem. Learning to read those messages early will save you a lot of frustration.

Let's cause one on purpose. Open `hello.auss` and **remove the semicolon** at the end of the line, then add a second line, so the file looks like this:

```
c.log("Hello, Aussom!")
c.log("Goodbye!");
```

Save and run it. Instead of the greeting, you get an error:

```
[error] hello.auss [2]: PARSE_ERROR: Unknown symbol found at line 2 column 1.

instead expected token classes are [SEMI, LBRACE, EQ, PLEQ, MIEQ, ...]
[error] Engine.parseScriptLine: parse error.
```

That looks like a lot, but you only need a few parts of it. Read it piece by piece:

- **`hello.auss`** — the file with the problem.
- **`PARSE_ERROR`** — the *kind* of problem. "Parse" is the reading step from Chapter 1, where Aussom checks that your code follows the rules. A parse error means the code broke a rule, so Aussom stopped before running anything.
- **`line 2 column 1`** — *where* Aussom noticed the problem. Here's a key lesson: the reported spot is where Aussom got confused, which is often *just after* the real mistake. Line 2 is the second `c.log`. Aussom expected the *previous* line to be finished before it got there — finished with a semicolon.
- **`instead expected token classes are [SEMI, ...]`** — what Aussom *wanted* to see. **`SEMI`** means *semicolon*. That's the strong hint: a semicolon is missing.

So the fix is on **line 1** (add the `;` back), even though the message pointed at line 2. Put the semicolon back, save, and run again — you're greeted once more.

When you hit an error, don't panic and don't guess randomly. Read the message, find the line number, and remember it often points at the line just *after* the mistake. With a little practice you'll read these at a glance.

What You Learned

You wrote, ran, and understood your first Aussom programs. Along the way you met the pieces you'll use constantly:

- A **`.auss` script** is a list of instructions Aussom runs from top to bottom — the simplest way to write Aussom. (The class-based `.aus` style comes later.)
- **`c.log("...")`** prints a line to the console, and `c` also offers `println`, `print`, `info`, `warn`, and `err`.
- A **string** is text in double quotes, and **`+`** joins strings together.
- Every **statement ends with a semicolon**.
- **`args`** hands your program the words typed after the file name.
- When something's wrong, Aussom prints an **error** with a kind, a location, and a hint — and the location often points just past the real mistake.

That's a solid foundation. In **Chapter 5**, we start filling it in, beginning with the most basic building block of all: **variables**, the way programs remember values.

Part II

Language Fundamentals

Chapter 5 - Variables, Values, and Comments

In Chapter 4 your program printed a fixed message. Real programs need to *remember* things — a name, a score, a total — and use them again later. The tool for that is the **variable**. This chapter shows how to create variables, how they can hold any kind of value, where they can be used, and how to leave notes in your code with comments.

From here on, every example is a `.auss` script, just like Chapter 4. Write the code in a file and run it with `aussom <file>`.

Creating and Assigning Variables

A **variable** is a named box that holds a value. You put a value in the box, and later you can read it back or replace it — all by using the box's name.

You create a variable simply by giving it a value, using the **=** sign:

```
name = "Ada";
age = 36;
```

That's it — no special keyword is needed. The first time you assign to a name, Aussom creates the variable for you.

Read the line `age = 36;` from the middle out: the name **age** is on the left, the value **36** is on the right, and the **=** in the middle means *"store the value on the right into the box on the left."* This is important: in programming, = means **"put this value here,"** not "is equal to" in the math sense. (Checking whether two things are equal is a different operator, ==, which you'll meet in Chapter 9.)

Once a variable exists, use it anywhere you'd use its value — just write its name:

```
c.log(name);                // prints: Ada
c.log("Age: " + age);       // prints: Age: 36
```

Naming your variables. A variable name can contain letters, digits, and underscores (_), but it can't start with a digit. Names are **case-sensitive**, so `age` and `Age` are two different variables. Beyond those rules, pick names that describe what the value is — `firstName` is far clearer than `x`. The common style in Aussom is to start with a lowercase letter and capitalize each following word, like `firstName` or `totalScore` (this is called *camelCase*).

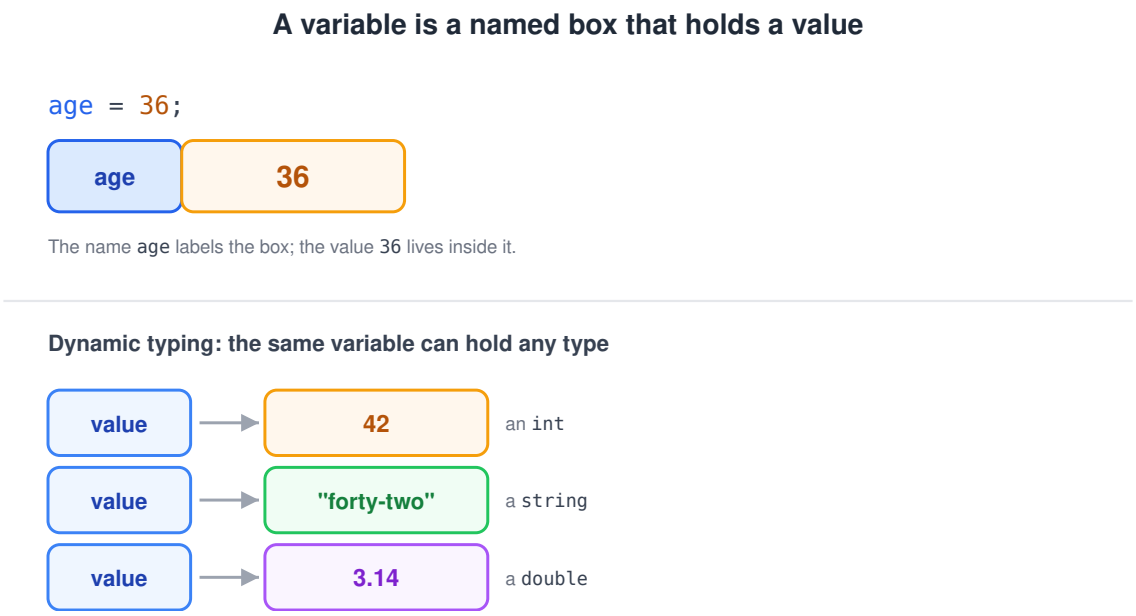


Figure 21: A variable is a named box; the same box can hold different types over time

Changing a variable. Assigning again replaces what's in the box:

```
age = 36;
age = 37;                    // the box now holds 37; the old 36 is gone
c.log("Next year Ada will be " + age + ".");
```

Dynamic Typing: Variables Hold Any Type

Every value has a **type** — 42 is an `int` (a whole number), `"Ada"` is a `string` (text), `3.14` is a `double` (a decimal number). You'll meet all the types in Chapter 6.

Aussom uses **dynamic typing**. That means a variable is not locked to one type. The same variable can hold a number now and text a moment later; Aussom figures out the type from whatever value you put in. To see this, we'll use **`lang.type(value)`**, a built-in helper that returns the name of a value's type as a string:

```
value = 42;
c.log("value holds a " + lang.type(value) + " (" + value + ")");
value = "forty-two";
c.log("value holds a " + lang.type(value) + " (" + value + ")");
value = 3.14;
c.log("value holds a " + lang.type(value) + " (" + value + ")");
```

Running this prints:

```
value holds a int (42)
value holds a string (forty-two)
value holds a double (3.14)
```

The one variable `value` happily held an `int`, then a `string`, then a `double`. That flexibility is convenient, but a word of advice: even though you *can* change a variable's type, it usually makes code clearer to let each variable hold one kind of thing. If `age` is a number, keep it a number.

Variable Scope

Scope is a fancy word for a simple idea: *where in your program a variable can be seen and used.*

In a script, a variable is available from the line where you create it onward, for the rest of the script. That includes inside blocks like `if` statements and loops (which we cover in Part IV), **and after them, too.** A variable you create inside a block does not disappear when the block ends:

```
if (true) {
    greeting = "hello";      // created inside the if block
}
c.log(greeting);             // still works out here: prints hello
```

This is worth remembering: Aussom does **not** give each block its own private scope. A variable created anywhere in your script sticks around for the rest of it. (Later, when you write functions inside classes in Part V and Part VI, you'll see that each function *does* get its own scope — variables created inside a function stay inside that function. But that's a story for later.)

Writing Comments

A **comment** is a note written for humans. Aussom completely ignores comments when it runs your program, so they never change what the code does. You use them to explain *why* your code does something, or to jot a reminder.

Aussom has two kinds of comment you'll use:

- **Line comment** — starts with `//` and runs to the end of the line:

  ```
  count = 0;        // start the counter at zero
  // This whole line is a comment.
  ```

- **Block comment** — starts with `/*` and ends with `*/`, and can span as many lines as you want:

  ```
  /* This is a block comment.
     It can go on for several lines.
     Aussom ignores all of it. */
  ```

There's a third kind, the **documentation comment**, written `/** ... */`. It's a special comment used to describe classes and functions, and you'll meet it in Chapter 20. Don't use it at the top of a script, though — it's reserved for describing declarations, and a script will report an error if one leads it. For everyday notes in a script, stick with `//` and `/* */`.

Good comments explain the *why*, not the obvious *what*. A comment like `// add one to count` next to `count++` just repeats the code; a comment like `// skip the header row` tells the reader something the code alone doesn't.

Try It Yourself

Here is a short program that pulls together everything in this chapter. Save it as `variables.auss` and run it:

```
// Create a variable by assigning a value with =.
name = "Ada";
age = 36;
c.log("Name: " + name);
c.log("Age: " + age);

// Reassign a variable. The box now holds a new value.
age = 37;
c.log("Next year Ada will be " + age + ".");

// Dynamic typing: one variable can hold any type of value.
value = 42;
c.log("value holds a " + lang.type(value) + " (" + value + ")");
value = "forty-two";
c.log("value holds a " + lang.type(value) + " (" + value + ")");
value = 3.14;
c.log("value holds a " + lang.type(value) + " (" + value + ")");

/* This is a block comment. Aussom ignores everything
   between the slash-star markers, across as many lines as you like. */
c.log("Done.");
```

Running it prints:

You can find this example here: `05-variables`.

What You Learned

- A **variable** is a named box for a value; create one by assigning with **=**, which means "store this value here" (not "equals").
- Aussom uses **dynamic typing** — a variable can hold any type of value, and you can check a value's type with **`lang.type(value)`**.
- Names are case-sensitive, can't start with a digit, and are best written in descriptive `camelCase`.
- A variable's **scope** is the rest of the script; Aussom has no separate block scope, so variables made inside `if`/loop blocks live on afterward.
- **Comments** (`//` for a line, `/* */` for a block) are notes for humans that Aussom ignores.

Next, in **Chapter 6**, we'll look closely at all nine of Aussom's data types — the different kinds of value a variable can hold.

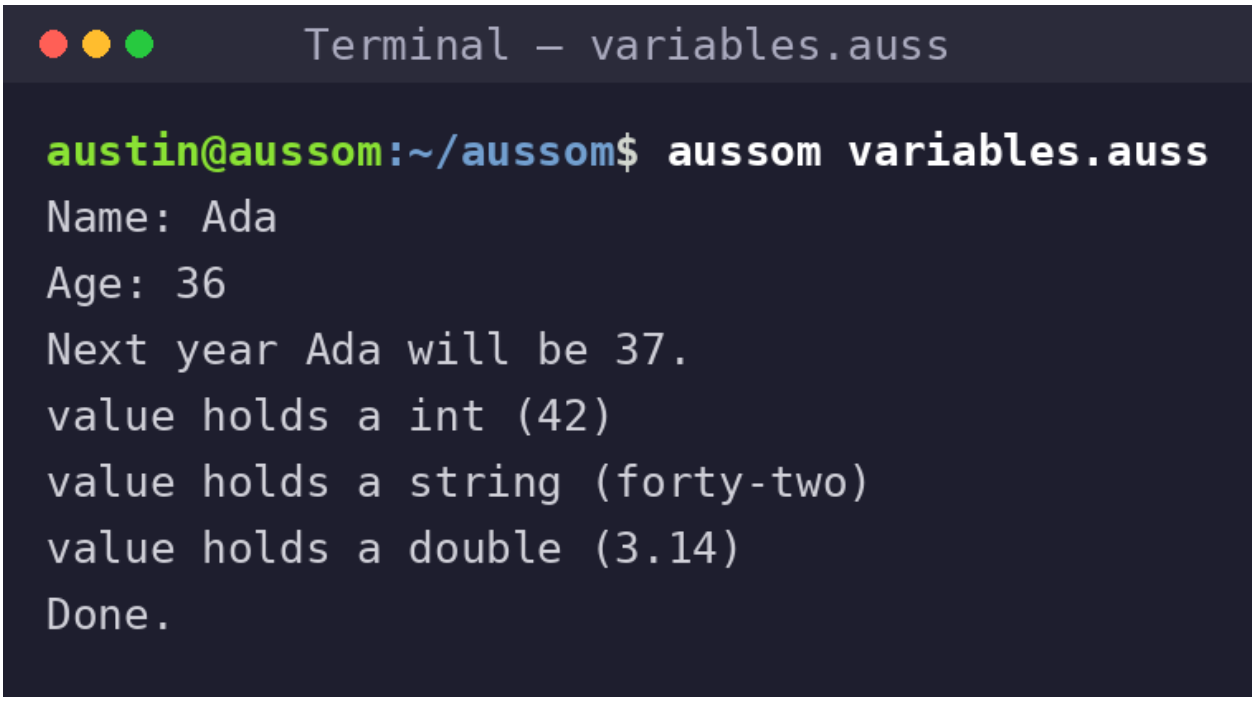


Figure 22: Running variables.auss

Chapter 6 - The Nine Data Types

Every value in Aussom has a **type** — a kind. `42` is a whole number, `"Ada"` is text, `true` is a yes/no value. Aussom has exactly **nine** types, and once you know them, a lot of the language falls into place: you'll understand what a value can do and which operations make sense for it.

This chapter gives you a quick tour of all nine, explains the handy idea of "truthiness," and shows how to ask Aussom what type a value is.

A Tour of the Types

Here are the nine types, grouped so they're easier to remember. Some get a whole chapter of their own later; for now, just meet them and learn their look.

Figure 23: The nine data types, grouped into simple values, collections, and other

Simple values each hold a single thing:

- **`null`** — the absence of a value; "nothing." A variable set to `null` is empty on purpose. Written as `null`.

- **`bool`** — a *boolean*: one of just two values, `true` or `false`. Booleans are what conditions (Chapter 12) test.
- **`int`** — an *integer*, a whole number with no decimal point: `42`, `-7`, `0`.
- **`double`** — a number with a decimal point: `3.14`, `-0.5`, `2.0`. (Chapter 7 covers `int` and `double` in depth.)
- **`string`** — text, written inside quotes: `"hello"`. (Chapter 8.)

Collections hold *other* values inside them:

- **`list`** — an ordered series of values, written in square brackets: `[1, 2, 3]`. You reach items by position. (Chapter 10.)
- **`map`** — a set of values you look up by a *key*, written in curly braces: `{"name": "Ada", "age": 36}`. (Chapter 11.)

Two more you'll use once your programs grow:

- **`callback`** — a reference to a function, so you can pass a function around and call it later. Written with `::`, like `::greet`. (Chapter 15.)
- **`object`** — an instance of a class, a bundle of data and the actions that go with it, created with `new`, like `new Dog()`. (Chapter 16.)

You've already used several of these without thinking about it: the strings you printed, the `int` ages, and the `args` list from Chapter 4.

Truthiness and `null`

Let's start with **`null`**. It means "no value." You'll get `null` when something is missing — for example, looking up a key that isn't in a map. You can set a variable to `null` yourself, and you can check for it with == (equals), which you'll meet fully in Chapter 9:

```
result = null;
if (result == null) {
    c.log("There's nothing here yet.");
}
```

Now, **truthiness**. In a condition — like the test inside an `if` — Aussom needs to decide whether a value counts as "yes" (true) or "no" (false). For an actual `bool` that's obvious. But *any* value can be used in a condition, so Aussom has a simple rule for the rest.

These values are **falsy** (they count as false):

- **`false`** — the boolean false.
- **`null`** — nothing.
- **`0`** and **`0.0`** — zero, whether whole or decimal.
- **`""`** — an empty string.
- **`[]`** and **`{}`** — an empty list or an empty map.

Everything else is truthy — any non-zero number, any non-empty string, any list or map with something in it, and `true`.

This is genuinely useful. Because an empty or missing value is falsy, you can write a check like "does this have a value?" very naturally:

```
name = "";
if (name) {
    c.log("Hi, " + name);
} else {
    c.log("I don't know your name yet.");   // this runs: "" is falsy
}
```

We'll cover `if/else` properly in Chapter 12; the point here is just how values turn into true or false.

Checking a Value's Type

Sometimes you want to know what type a value actually is — while learning, while debugging, or to decide what to do with it. The built-in helper **`lang.type(value)`** returns the type's name as a string:

```
c.log(lang.type(42));            // int
c.log(lang.type(3.14));          // double
c.log(lang.type("hello"));       // string
c.log(lang.type(true));          // bool
c.log(lang.type(null));          // null
c.log(lang.type([1, 2, 3]));     // list
c.log(lang.type({"a": 1}));      // map
```

For the simple types and collections you get exactly the names from this chapter: `int`, `double`, `string`, `bool`, `null`, `list`, `map`. For a `callback` you get `"callback"`, and for an **object** you get the *class name* of the object — so `lang.type(new Dog())` returns `"Dog"`. That's handy once you start writing classes in Chapter 16.

Try It Yourself

This program names one value of each simple type, adds a list and a map, reports each type with `lang.type`, and then shows truthiness in action. Save it as `datatypes.auss` and run it:

```
nothing = null;
flag    = true;
whole   = 42;
decimal = 3.14;
text    = "hello";
numbers = [1, 2, 3];
person  = {"name": "Ada", "age": 36};

c.log("null    -> " + lang.type(nothing));
c.log("bool    -> " + lang.type(flag));
c.log("int     -> " + lang.type(whole));
c.log("double  -> " + lang.type(decimal));
c.log("string  -> " + lang.type(text));
c.log("list    -> " + lang.type(numbers));
```

```
c.log("map     -> " + lang.type(person));

// Truthiness: which values count as false inside a condition?
if (0)    { c.log("0 is truthy"); }     else { c.log("0 is falsy"); }
if ("")   { c.log("'' is truthy"); }    else { c.log("'' is falsy"); }
if (null) { c.log("null is truthy"); }  else { c.log("null is falsy"); }
if ("hi") { c.log("'hi' is truthy"); }  else { c.log("'hi' is falsy"); }
```

Running it prints:

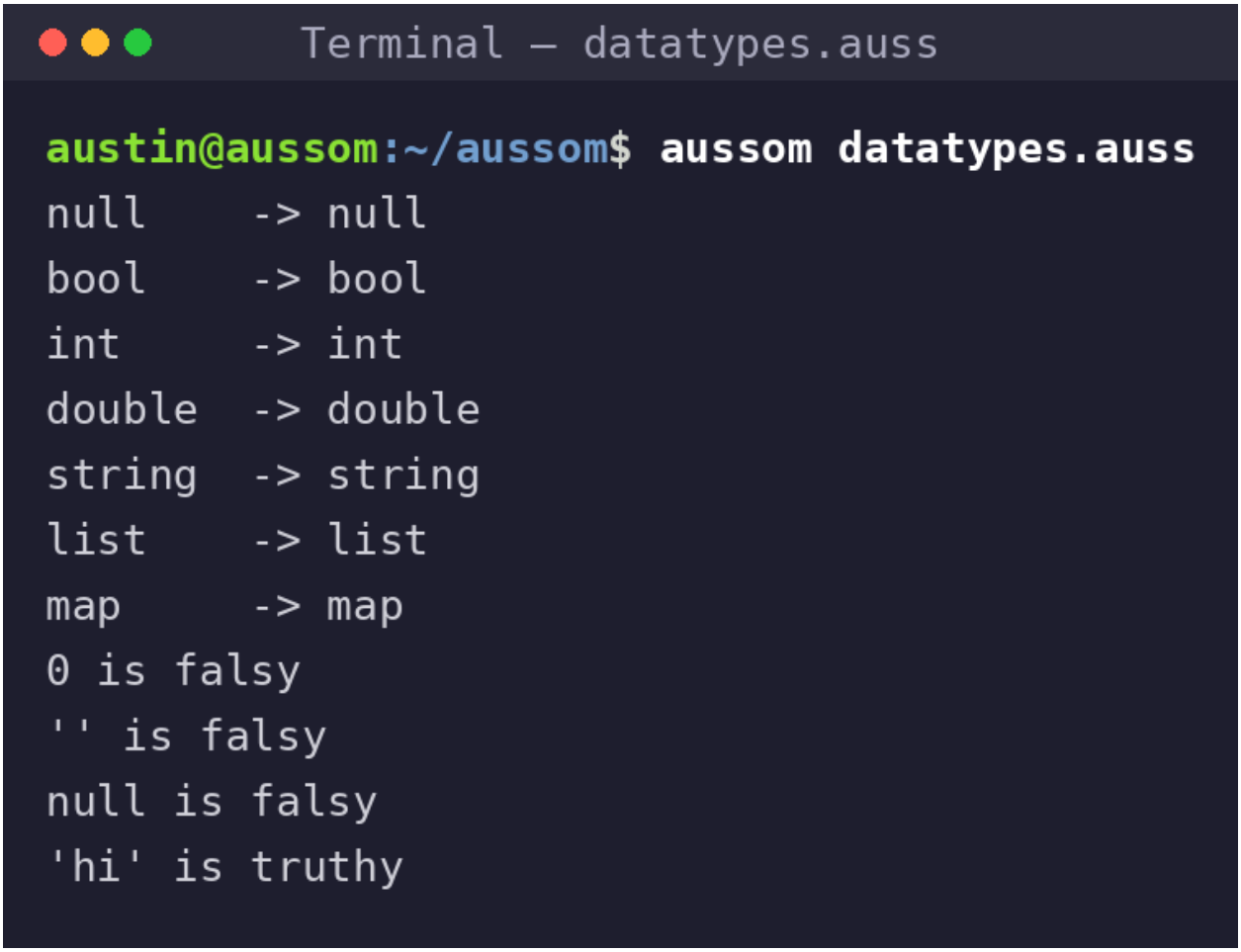


Figure 24: Running datatypes.auss

You can find this example here: `06-data-types`.

What You Learned

- Aussom has **nine data types**: `null`, `bool`, `int`, `double`, `string`, `list`, `map`, `callback`, and `object`.
- **`null`** means "no value," and you test for it with `== null`.

- **Truthiness**: `false`, `null`, `0`, `0.0`, `""`, `[]`, and `{}` are *falsy*; everything else is *truthy*. This lets you write checks like `if (name)`.
- **`lang.type(value)`** tells you a value's type — the type name for simple values, or the class name for an object.

Next, in **Chapter 7**, we'll dig into the two number types, `int` and `double`, and everything you can do with them.

Chapter 7 - Working with Numbers

Numbers show up in almost every program — counting things, adding up totals, measuring, scoring. In Chapter 6 you met Aussom's two number types, `int` and `double`. This chapter puts them to work: the math operators, the methods a number carries, how to convert between numbers and other types, and the handy `Int`, `Double`, and `Bool` helper classes.

Integers and Decimals

Aussom has two number types:

- **`int`** — an *integer*, a whole number with no fractional part: `42`, `-7`, `0`, `1000`. Internally it's a 64-bit number, so it can hold very large values.
- **`double`** — a number with a decimal point: `3.14`, `-0.5`, `2.0`.

Aussom decides which one you mean from how you write the number. A plain number is an `int`; a number with a decimal point is a `double`:

```
count = 10;      // an int
price = 3.99;    // a double
c.log(lang.type(count));   // int
c.log(lang.type(price));   // double
```

Arithmetic and Number Methods

You do math with the usual operators. Most behave exactly as you'd expect:

```
c.log(10 + 3);     // 13   addition
c.log(10 - 3);     // 7    subtraction
c.log(10 * 3);     // 30   multiplication
c.log(10 % 3);     // 1    remainder (what's left after dividing)
```

Division has **one surprise worth knowing up front**: the `/` operator *always* gives a `double`, even when the numbers divide evenly.

```
c.log(10 / 3);     // 3.3333333333333335   a double, as you'd expect
c.log(10 / 5);     // 2.0                  still a double, not 2
```

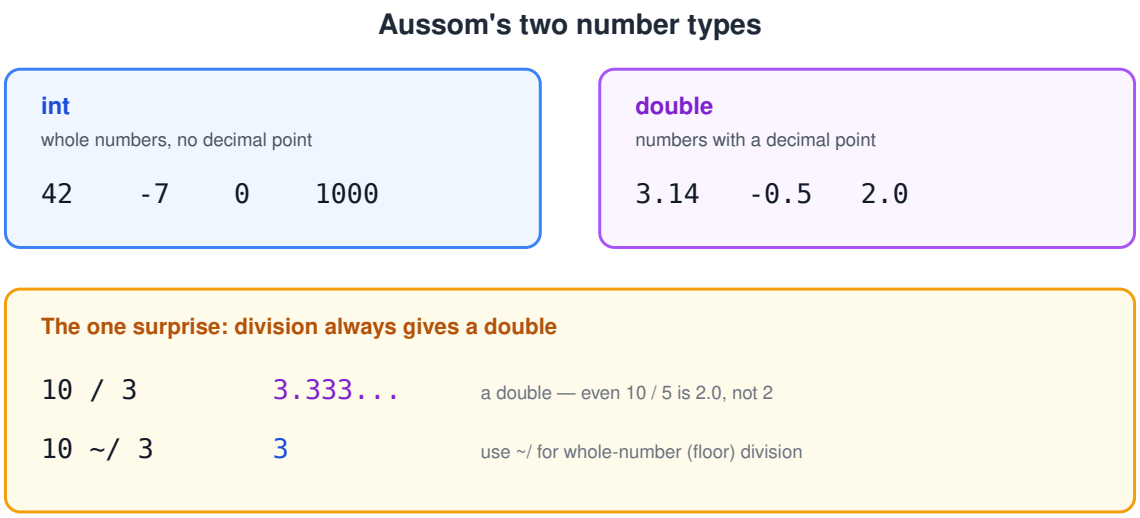


Figure 25: Aussom's two number types, and how division behaves

When you want whole-number division instead, use `~/` (floor division). It divides and throws away the fractional part, giving you an `int`:

```
c.log(10 ~/ 3);    // 3    whole-number (floor) division
```

If you mix an `int` and a `double` in the same expression, the result is a `double` (`2 + 3.0` is `5.0`).

Changing a number in place. These shortcuts update a variable using its own current value — handy in loops and running totals:

- **`+=`** — add and store: `total += 5;` is short for `total = total + 5;`.
- **`-=`**, **`*=`**, **`/=`** — the same idea for subtract, multiply, divide.
- **`++`** — add one: `count++;`.
- **`--`** — subtract one: `count--;`.

Number methods. A number value also carries methods you call with a dot. The common ones on an **`int`**:

- **`toString`** — turn the number into text: `n.toString()` gives `"42"`.
- **`toDouble`** — convert to a `double`: `(42).toDouble()` gives `42.0`.
- **`toBool`** — convert to a boolean (`0` is false, anything else true).
- **`compare`** — compare with another int; returns a negative number, `0`, or a positive number depending on order.

An `int` also has rarely-needed helpers for working with its binary form — `toBinary`, `toHex`, `toOctal`, and `signum` (the sign, as -1, 0, or 1) — which you can look up in the API reference if you ever need them.

The common methods on a **`double`**:

- **`toInt`** — convert to an `int` by **dropping** the decimal part (it truncates, it doesn't round): `(3.99).toInt()` gives 3.
- **`toString`** — turn it into text.

- **`isNan`** — true if the value is "not a number" (the result of an undefined operation).
- **`isInfinite`** — true if the value is infinity.

  For more advanced math — rounding, powers, square roots, absolute value, random numbers — Aussom has a `math` module, covered in Chapter 23. This chapter sticks to what the number types do on their own.

Converting Between Numbers, Strings, and Booleans

Programs constantly move values between types — reading a number that arrived as text, or building a message out of numbers. Here's the toolkit.

Number to string. Call `toString()`, or just use the number in a string with +, which converts it automatically (you've been doing this since Chapter 4):

```
n = 42;
c.log(n.toString());        // "42"
c.log("The answer is " + n);   // "The answer is 42"  (auto-converted)
```

String to number. Text like `"100"` is a `string`, not a number — you can't do math on it until you convert it. Use the helper classes `Int` and `Double`:

```
c.log(Int.parse("100"));       // 100    a real int
c.log(Double.parse("3.5"));    // 3.5    a real double
```

Between int and double. Use `toDouble()` to add a decimal part, and `toInt()` to drop it:

```
c.log((7).toDouble());     // 7.0
c.log((3.99).toInt());     // 3   (truncates, so 3.99 becomes 3)
```

Booleans. Convert a `bool` to a number with `toInt()` (true is `1`, false is `0`), and parse text into a bool with `Bool.parse`:

```
c.log((true).toInt());       // 1
c.log(Bool.parse("true"));   // true
```

If you hand `Int.parse` or `Double.parse` text that isn't a valid number, it raises an error rather than guessing. You'll learn to handle errors like that in Chapter 19.

The `Int`, `Double`, and `Bool` Helpers

You may have noticed two spellings: lowercase `int` and capitalized `Int`. They are different, and the difference is worth understanding:

- The **lowercase** `int`, `double`, and `bool` are the **types**. When you have a value, you call *its* methods on it — `n.toString()`, `price.toInt()`.
- The **capitalized** `Int`, `Double`, and `Bool` are **static helper classes**. They're not values; they're a home for functions you call *by name* for things that don't belong to any one value — like turning text into a number, or asking for the largest possible integer.

The most useful helper functions:

- **`Int.parse`** — text to an `int`: `Int.parse("100")`.
- **`Int.maxVal`** / **`Int.minVal`** — the largest and smallest `int` Aussom can hold.
- **`Double.parse`** — text to a `double`: `Double.parse("3.5")`.
- **`Double.maxVal`** / **`Double.minVal`** — the largest and smallest `double`.
- **`Double.posInfinity`** / **`Double.negInfinity`** / **`Double.nanVal`** — the special "infinity" and "not a number" values.
- **`Bool.parse`** — text to a `bool`: `Bool.parse("true")`.

For example:

```
c.log(Int.maxVal());     // 9223372036854775807  — the biggest int
```

Try It Yourself

This program exercises the arithmetic operators, the division rule, conversions, and a helper. Save it as `numbers.auss` and run it:

```
// int is whole; double has a decimal point.
count = 10;
price = 3.99;
c.log("count is a " + lang.type(count) + ", price is a " + lang.type(price));

// Arithmetic.
c.log("10 + 3  = " + (10 + 3));
c.log("10 - 3  = " + (10 - 3));
c.log("10 * 3  = " + (10 * 3));
c.log("10 / 3  = " + (10 / 3));    // division ALWAYS gives a double
c.log("10 / 5  = " + (10 / 5));    // even when it divides evenly
c.log("10 ~/ 3 = " + (10 ~/ 3));   // ~/ is whole-number (floor)
                                   // division
c.log("10 % 3  = " + (10 % 3));    // % is the remainder

// Conversions.
c.log("(3.99).toInt()      = " + (3.99).toInt());   // drops the decimal
                                                    // part
c.log("(42).toString()     = " + (42).toString());
c.log("Int.parse('100')    = " + Int.parse("100"));
c.log("Double.parse('3.5') = " + Double.parse("3.5"));
c.log("(true).toInt()      = " + (true).toInt());

// The Int / Double helper classes hold handy functions.
c.log("Int.maxVal()        = " + Int.maxVal());
```

Running it prints:

You can find this example here: `07-numbers`.

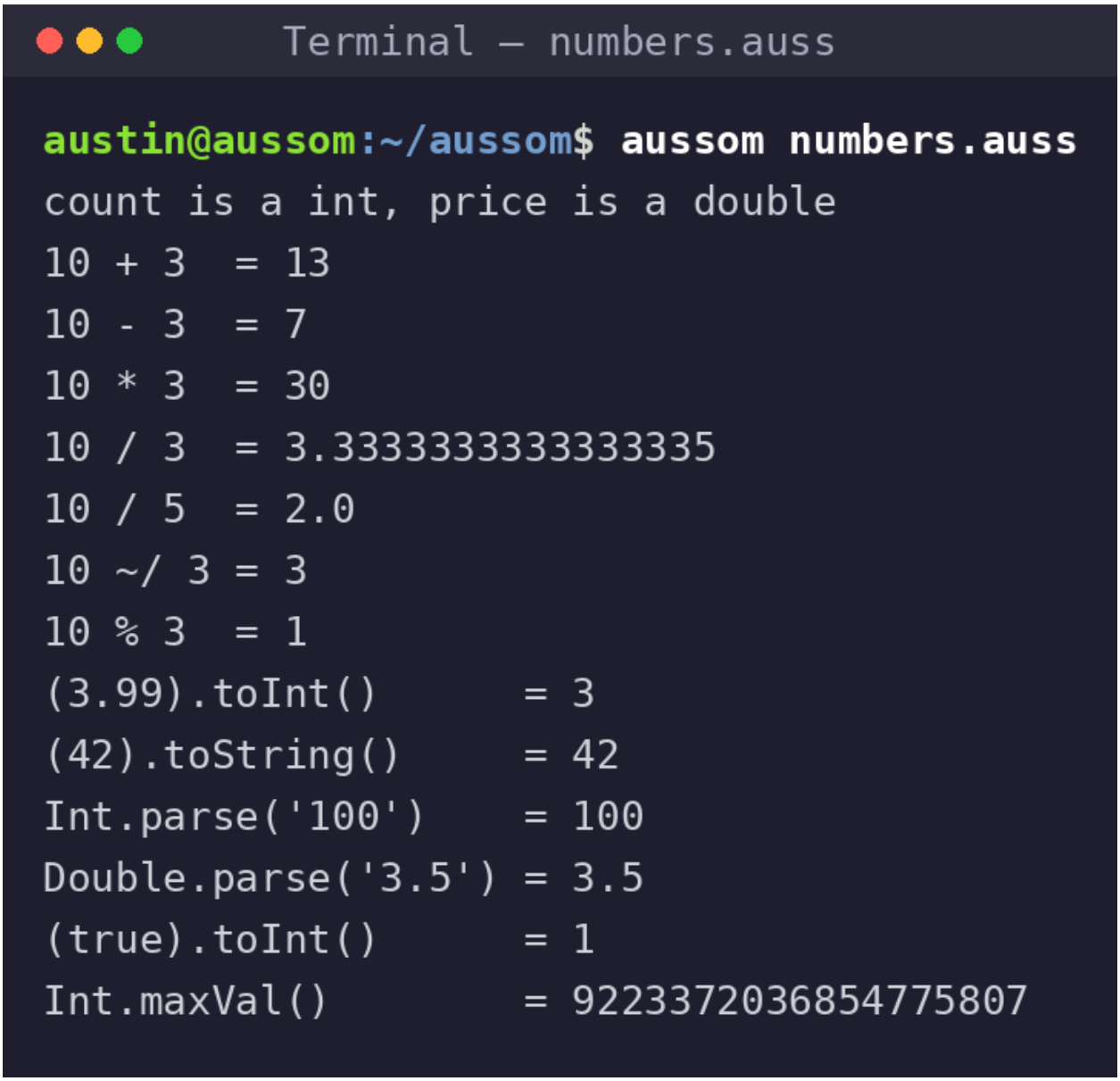


Figure 26: Running numbers.auss

What You Learned

- Aussom has two number types: **`int`** (whole) and **`double`** (decimal); a decimal point in the literal decides which.
- The math operators are `+`, `-`, `*`, `/`, `~/`, and `%`. **`/` always produces a `double`**; use **`~/`** for whole-number division. Compound forms (`+=`, `++`, …) update a variable in place.
- Numbers carry methods like **`toString`**, **`toInt`** (truncates), and **`toDouble`**. Convert text to numbers with **`Int.parse`** / **`Double.parse`**.
- Lowercase **`int`/`double`/`bool`** are the *types* (methods on a value); capitalized **`Int`/`Double`/`Bool`** are *helper classes* (functions like `parse` and `maxVal`).

That wraps up numbers. Next, in **Chapter 8**, we turn to the other everyday value you'll use constantly: the **string**, and all the ways Aussom lets you work with text.

Chapter 8 - Working with Strings

Text is everywhere in programs — names, messages, file contents, answers typed by a user. In Aussom, a piece of text is a **string**, and you've been using strings since your very first program. This chapter gives you the full toolkit: how to write strings (including the tricky business of quotes), the methods a string carries for searching and reshaping text, and how to build a bigger string out of other values.

Creating Strings and Escaping Quotes

A **string** is text wrapped in quotes. Aussom lets you use **double quotes** or **single quotes**, and they work the same way:

```
a = "Hello, Aussom!";
b = 'Hello, Aussom!';
```

Having both is handy when your text *contains* a quote. Put the text in the other kind of quote and you don't have to do anything special:

```
c.log("It's a fine day.");                  // apostrophe inside double
                                            // quotes
c.log('She said "hello" and waved.');       // double quotes inside
                                            // single quotes
```

But what if you need the *same* kind of quote that surrounds the string? Then you **escape** it with a backslash (\) — the backslash tells Aussom "this next character is part of the text, not the end of the string":

```
c.log("She said \"hello\" and waved.");     // \" is a literal double
                                            // quote
```

The backslash starts a few other useful **escape sequences**:

- **\n** — a new line (starts the next line).
- **\t** — a tab.
- **\"** — a double quote.
- **** — a single backslash.

```
c.log("Line one\nLine two");
c.log("Name:\tAda");
```

prints:

```
Line one
Line two
Name:   Ada
```

One more thing to know up front: string methods **never change the original string** — they return a *new* string with the change. So `s.toUpper()` gives you an upper-case copy; `s` itself is untouched. (If you want to keep the result, assign it to a variable.)

Common String Methods

A string carries a rich set of methods you call with a dot. Here are the ones you'll reach for most, grouped by what they do. In the examples, `s` is `"Hello, Aussom!"`.

Length — how many characters:

```
s.length()      // 14   — also available as the # operator: #s
```

Case — upper and lower:

- **`toUpper`** — an all-caps copy: `s.toUpper()` gives `"HELLO, AUSSOM!"`.
- **`toLower`** — an all-lowercase copy: `s.toLower()` gives `"hello, aussom!"`.

Searching — is something in there, and where:

```
s.indexOf("Aussom")     // 7 — position of the first match (-1 if not
                        // found)
s.contains("Aussom")    // true — is that text anywhere inside?
s.startsWith("Hello")   // true
s.endsWith("!")         // true
```

- Related: **`lastIndexOf`** (search from the end) and **`indexOfStart`** (start searching from a given index).

Slicing — pull out part of a string. Characters are numbered from **0** (the first character is at index 0):

```
s.charAt(0)       // "H"        — the single character at an index
s.substr(7)       // "Aussom!"  — from index 7 to the end
s.substr(0, 5)    // "Hello"    — from index 0 up to (not including) 5
```

Notice the second number in `substr(0, 5)` is the **end index**, and it's *not* included — you get characters 0, 1, 2, 3, and 4, which is `"Hello"`.

Cleaning up — whitespace and emptiness:

- **`trim`** — a copy with spaces removed from both ends: `"  hi  ".trim()` gives `"hi"`.
- **`isEmpty`** — true if the string has no characters (`""`).
- **`isBlank`** — true if the string is empty or only whitespace.

Replacing:

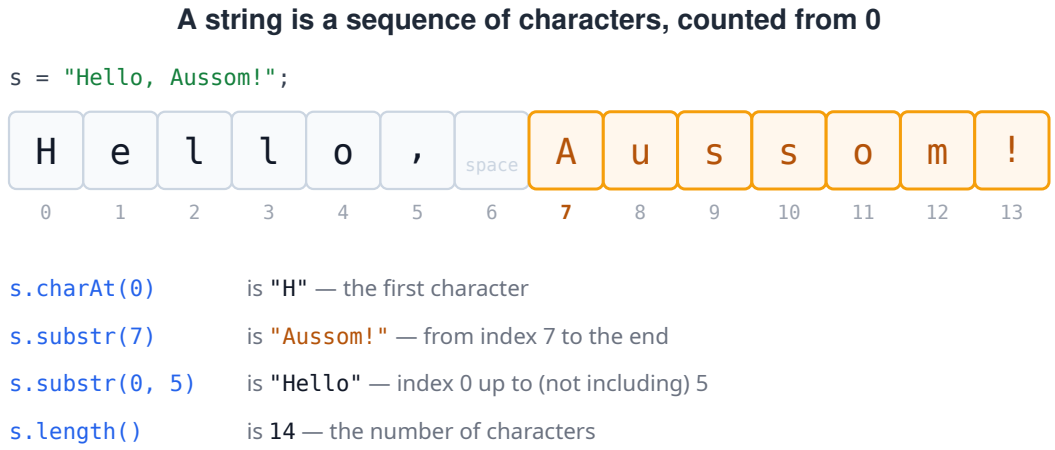


Figure 27: How a string is indexed, and what charAt, substr, and length return

```
s.replace("Aussom", "World")    // "Hello, World!" — swaps every match
```

- Related: **`replaceRegex`** and **`replaceFirstRegex`** for pattern-based replacement (you'll meet regular expressions in Chapter 24).

Splitting and joining — move between a string and a list of pieces:

```
"red,green,blue".split(",")     // ["red", "green", "blue"] — string to
                                // list
["red", "green", "blue"].join(", ")   // "red, green, blue" — list back
                                      // to string
```

`split` is a string method; `join` is a *list* method (lists are Chapter 10). The two are a matched pair for chopping text apart and putting it back together.

Comparing — the operator == checks whether two strings are equal, and string methods add case-insensitive options:

- **`equals`** — same as ==, true if the strings match exactly.
- **`equalsICase`** — true if they match ignoring upper/lower case.
- **`compare`** / **`compareICase`** — return a negative number, `0`, or a positive number to say which comes first alphabetically (useful for sorting).

That's the everyday set. There are a few more (like `matches` for a regular expression); the full list is in the API reference at aussom-lang.com.

Building Strings from Values

You build a bigger string by joining pieces with the **+** operator. When one side of a + is a string, Aussom converts the other side to text automatically — so you can drop numbers and other values right in:

```
name = "Ada";
age = 36;
c.log("Message: " + name + " is " + age + " years old.");
```

prints:

```
Message: Ada is 36 years old.
```

This is the same string concatenation you've used since Chapter 4, now with a name. For assembling a message from a list of items, `join` is often cleaner than a long chain of +.

Try It Yourself

This program exercises the common string methods and builds a message from values. Save it as `strings.auss` and run it:

```
s = "Hello, Aussom!";
c.log("length:      " + s.length());
c.log("upper:       " + s.toUpper());
c.log("lower:       " + s.toLower());
c.log("charAt(0):   " + s.charAt(0));
c.log("substr(7):   " + s.substr(7));      // index 7 to the end
c.log("substr(0,5): " + s.substr(0, 5));   // index 0 up to (not incl.)
                                           // 5
c.log("indexOf:     " + s.indexOf("Aussom"));
c.log("contains:    " + s.contains("Aussom"));
c.log("startsWith:  " + s.startsWith("Hello"));
c.log("replace:     " + s.replace("Aussom", "World"));
c.log("trim:        '" + "   spaced   ".trim() + "'");

// split() breaks a string into a list of pieces.
parts = "red,green,blue".split(",");
c.log("split count: " + #parts + ", first: " + parts[0]);

// Build a longer string from values with + (non-strings convert
↪   automatically).
name = "Ada";
age = 36;
c.log("Message:     " + name + " is " + age + " years old.");
```

Running it prints:

You can find this example here: `08-strings`.

What You Learned

- A **string** is text in single or double quotes; use the other quote or an **escape** (`\"`, `\n`, `\t`, `\\`) when your text contains a quote or special character.

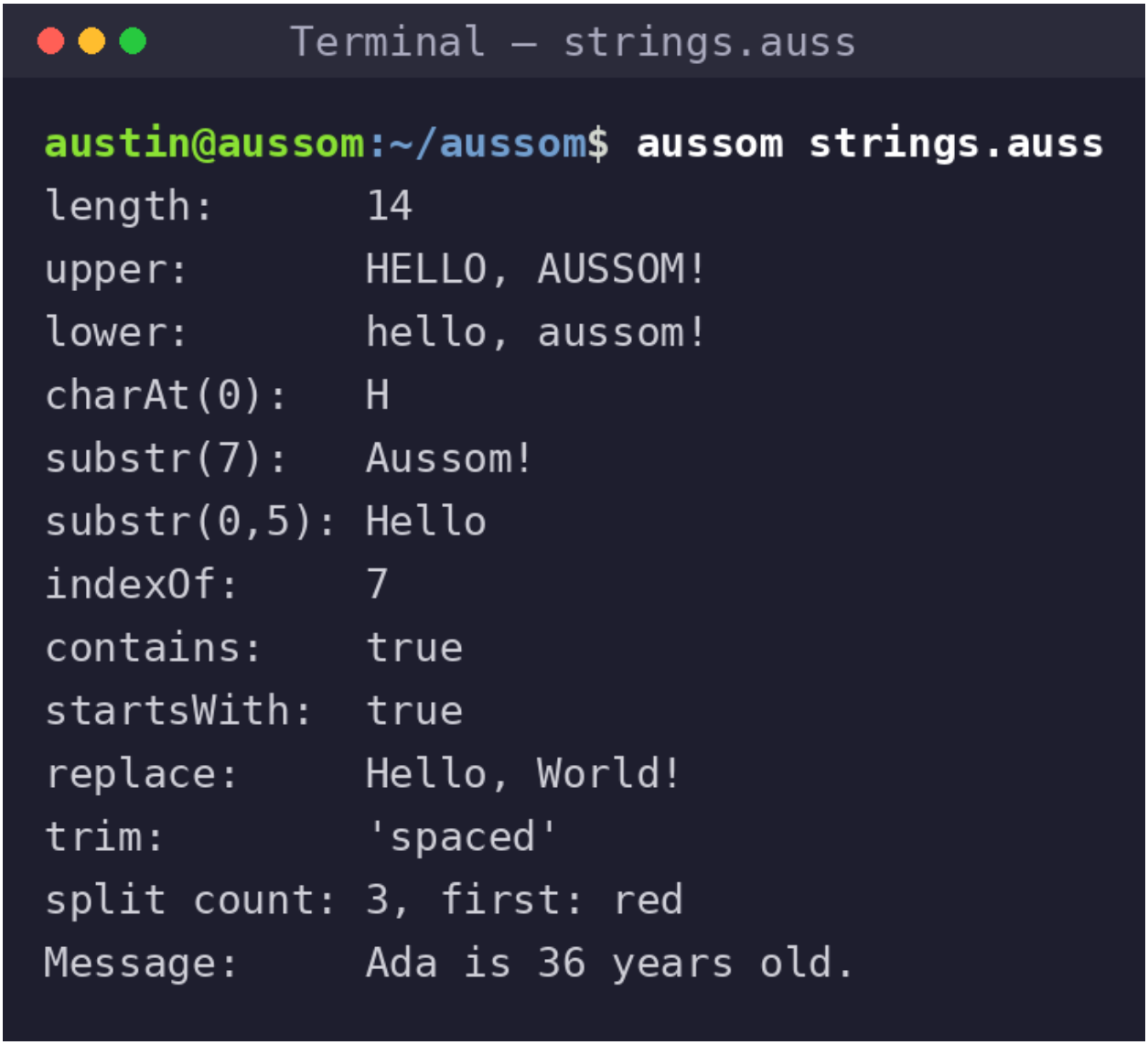


Figure 28: Running strings.auss

- String methods return a **new** string; the original is never changed.
- Common methods: **`length`**, **`toUpper/toLower`**, **`indexOf/contains/ startsWith/endsWith`**, **`charAt/substr`** (end index is exclusive), **`trim`**, **`replace`**, and **`split`** (with the list's **`join`** as its partner).
- Build strings with **+**, which converts non-string values to text automatically.

Next, in **Chapter 9**, we'll pull together all the **operators** you've been using — arithmetic, comparison, logical, and a few special ones — and see the rules for how they combine.

Chapter 9 - Operators and Expressions

You've already used **operators** in every chapter so far — + to add and to join strings, = to assign, == to compare. An operator is a symbol that combines values to produce a new value. A bit of code that produces a value is called an **expression** — `2 + 3`, `age >= 18`, and `"Hi, " + name` are all expressions.

This chapter gathers the operators in one place, adds the few you haven't met yet, and explains **precedence** — the rules for which operator acts first when you don't use parentheses.

Arithmetic Operators

You met these in Chapter 7, so here's a quick recap. They work on numbers:

```
c.log(10 + 3);     // 13   addition
c.log(10 - 3);     // 7    subtraction
c.log(10 * 3);     // 30   multiplication
c.log(10 / 3);     // 3.3333333333333335 division ALWAYS gives a double
c.log(10 ~/ 3);    // 3    ~/ is whole-number (floor) division
c.log(10 % 3);     // 1    % is the remainder
```

The one to remember is that `/` always produces a `double` — see Chapter 7 for the details and the `~/` alternative.

Comparison and Logical Operators

Comparison operators ask a yes/no question about two values and produce a `bool` (`true` or `false`):

```
c.log(5 == 5);     // true   equal to
c.log(5 != 3);     // true   not equal to
c.log(5 < 3);      // false  less than
c.log(5 > 3);      // true   greater than
c.log(5 <= 5);     // true   less than or equal to
c.log(5 >= 6);     // false  greater than or equal to
```

Two notes worth keeping in mind:

- **== compares values, = assigns them.** `age = 18` *stores* 18 in `age`; `age == 18` *asks whether* `age` is 18. Mixing these up is a classic beginner bug, so read == as "is equal to."

- == and != work on strings too (`"hi" == "hi"` is `true`), and numbers compare across types (`5 == 5.0` is `true`).

Logical operators combine or flip `bool` values. They're what you use to build bigger yes/no questions:

- **&&** (*and*) — true only if **both** sides are true.
- **||** (*or*) — true if **either** side is true.
- **!** (*not*) — flips true to false and false to true.

```
c.log(true && false);   // false  — both must be true
c.log(true || false);   // true   — at least one is true
c.log(!true);           // false  — flipped
```

You'll use comparison and logical operators constantly once we reach conditions in Chapter 12, where they decide which code runs.

Assignment and the Append Operator

You create and update variables with **assignment**:

- **=** — store a value: `count = 10;`.
- **+=**, **-=**, ***=**, **/=** — update a variable using its current value: `count += 5;` is short for `count = count + 5;`.
- **++** and **--** — add or subtract one: `count++;`, `count--;`.

There's also a special operator for lists, the **append operator @=**, which adds an item to the end of a list:

```
items = [];              // an empty list
items @= "first";        // items is now ["first"]
items @= "second";       // items is now ["first", "second"]
```

You'll work with lists fully in Chapter 10; @= is the quick way to grow one.

The Length Operator and the Callback Operator

Two more symbols round out the set:

- **#** — the **length** (or count) operator. Placed before a string, list, or map, it gives the number of items:

  ```
  c.log(#"hello");    // 5   — characters in the string
  c.log(#items);      // 2   — items in the list
  ```

  It does the same job as the `.length()` and `.size()` methods; # is just shorter.

- **::** — the **callback** operator. Placed before a function's name, it creates a *callback* — a reference to that function that you can store and pass around, to be called later. You met the `callback` type in Chapter 6, and you'll use callbacks fully in Chapter 15:

```
handler = ::onClick;    // a reference to the onClick function
```

The Safe Access Operator

Reaching for something that isn't there — a map key that was never set, a member an object doesn't have, or a list index past the end — is an **error** that stops your program (you'll learn to catch errors with `try/catch` in Chapter 19):

```
user = {"name": "Ada"};
c.log(user["email"]);       // ERROR — there is no "email" key
```

The **safe access operator `?`** avoids that. Put it in front of the access, and a missing value simply comes back as **`null`** instead of raising an error:

```
c.log(?user["email"]);      // null — no error, just "nothing there"
c.log(?user["name"]);       // Ada — a value that IS there comes back
                            // normally
```

It works the same for object members and list indexes:

```
nums = [10, 20, 30];
c.log(?nums[99]);           // null — past the end, but no crash
```

This turns a "check first, then look" dance into a single expression. A common use is a condition that reacts to something being absent:

```
if (?user["email"] == null) {
    c.log("No email on file.");
}
```

Without `?` you'd need a `try/catch` or a separate existence check; with it, one line does the job.

Operator Precedence

When an expression has several operators and no parentheses, **precedence** decides which one acts first. It follows the math you already know: multiplication before addition, and so on.

```
c.log(2 + 3 * 4);      // 14  — the 3 * 4 happens first, then + 2
c.log((2 + 3) * 4);    // 20  — parentheses force the + to happen first
```

From first (highest) to last, the order is:

1. **`( )`** — parentheses always win.
2. **`!`** and **`?`** — logical *not* and safe access (each applies to the single value right after it).
3. **`* / ~/ %`** — multiply, divide, floor-divide, remainder.
4. **`+ -`** — add, subtract.
5. **`< > <= >= == !=`** — comparisons.
6. **`&&`** — logical *and*.

7. `||` — logical *or*.

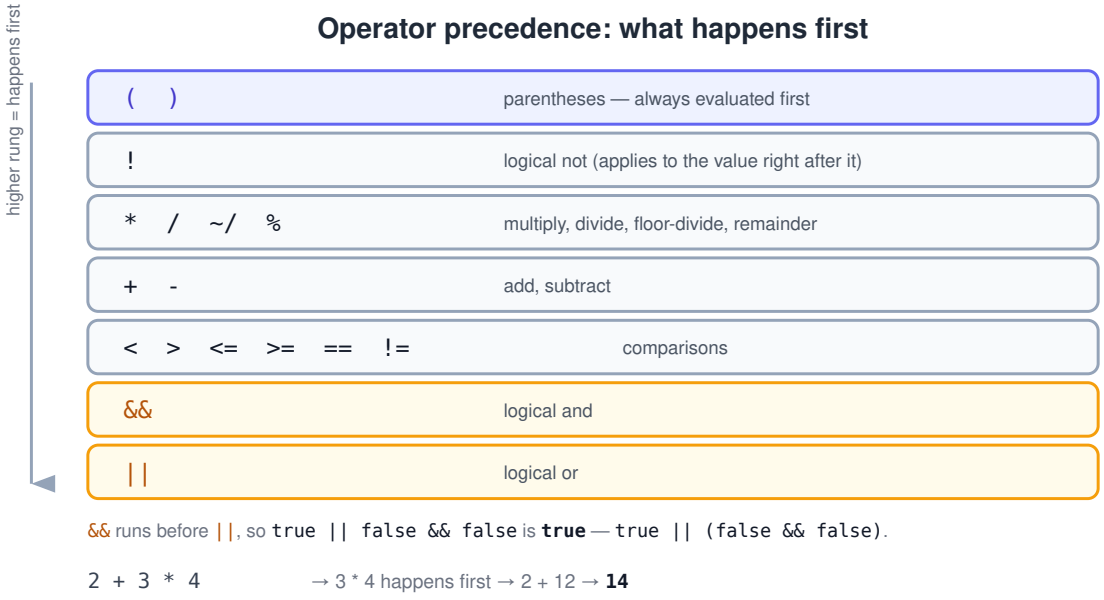


Figure 29: The precedence ladder, from parentheses at the top down to logical or at the bottom

Two of these are worth a closer look. First, the logical operators aren't all equal: **`&&` runs before `||`**. So a combined condition groups its `&&` part first:

```
c.log(true || false && false);   // true
```

Aussom reads that as `true || (false && false)`: it works out `false && false` (which is `false`), then `true || false`, which gives `true`.

Second, when a condition gets hard to read at a glance, don't rely on memorizing the whole ladder — **add parentheses** to say exactly what you mean. `true || (false && false)` and `(true || false) && false` are different expressions, and the parentheses make which one you meant obvious to both Aussom and the next person reading your code.

Try It Yourself

This program runs through comparison, logical, append, length, and precedence. Save it as `operators.auss` and run it:

```
// Comparison operators produce a bool.
c.log("5 == 5:   " + (5 == 5));
c.log("5 != 3:   " + (5 != 3));
c.log("5 < 3:    " + (5 < 3));
c.log("5 >= 5:   " + (5 >= 5));
c.log("'a'=='a': " + ("a" == "a"));

// Logical operators combine or flip booleans.
```

```
c.log("true && false: " + (true && false));
c.log("true || false: " + (true || false));
c.log("!true:         " + (!true));

// @= appends to a list; # counts a string, list, or map.
items = [];
items @= "first";
items @= "second";
c.log("items: " + #items + " long, first = " + items[0]);
c.log("length of \"hello\": " + #"hello");

// ? safely reads a missing key/member/index as null instead of erroring.
user = {"name": "Ada"};
c.log("?user[\"name\"]:  " + ?user["name"]);
c.log("?user[\"email\"]: " + ?user["email"]);
c.log("?items[99]:      " + ?items[99]);

// Precedence: * before +, and parentheses win.
c.log("2 + 3 * 4    = " + (2 + 3 * 4));
c.log("(2 + 3) * 4  = " + ((2 + 3) * 4));
// && runs before ||, so this is true || (false && false) = true.
c.log("true || false && false = " + (true || false && false));
```

Running it prints:

You can find this example here: `09-operators`.

What You Learned

- An **expression** is any code that produces a value; **operators** are what combine values.
- **Comparison** operators (==, !=, <, >, <=, >=) produce a `bool`; **logical** operators (&&, ||, !) combine booleans. Remember == (compare) versus = (assign).
- **Assignment**: =, the compound forms (+=, ++, ...), and the list append operator **@=**. The **#** operator gives length; **::** makes a callback; the **safe access ?** returns `null` instead of erroring on a missing key, member, or index.
- **Precedence** runs math-style (parentheses → ! → * / → + - → comparisons → && → ||), with **&& running before ||**. When a combined condition is hard to read, add parentheses to make the order clear.

That completes the core building blocks of Part II. Next, in **Part III**, we start working with the two collection types you've been glimpsing — beginning with **Chapter 10, Lists**.

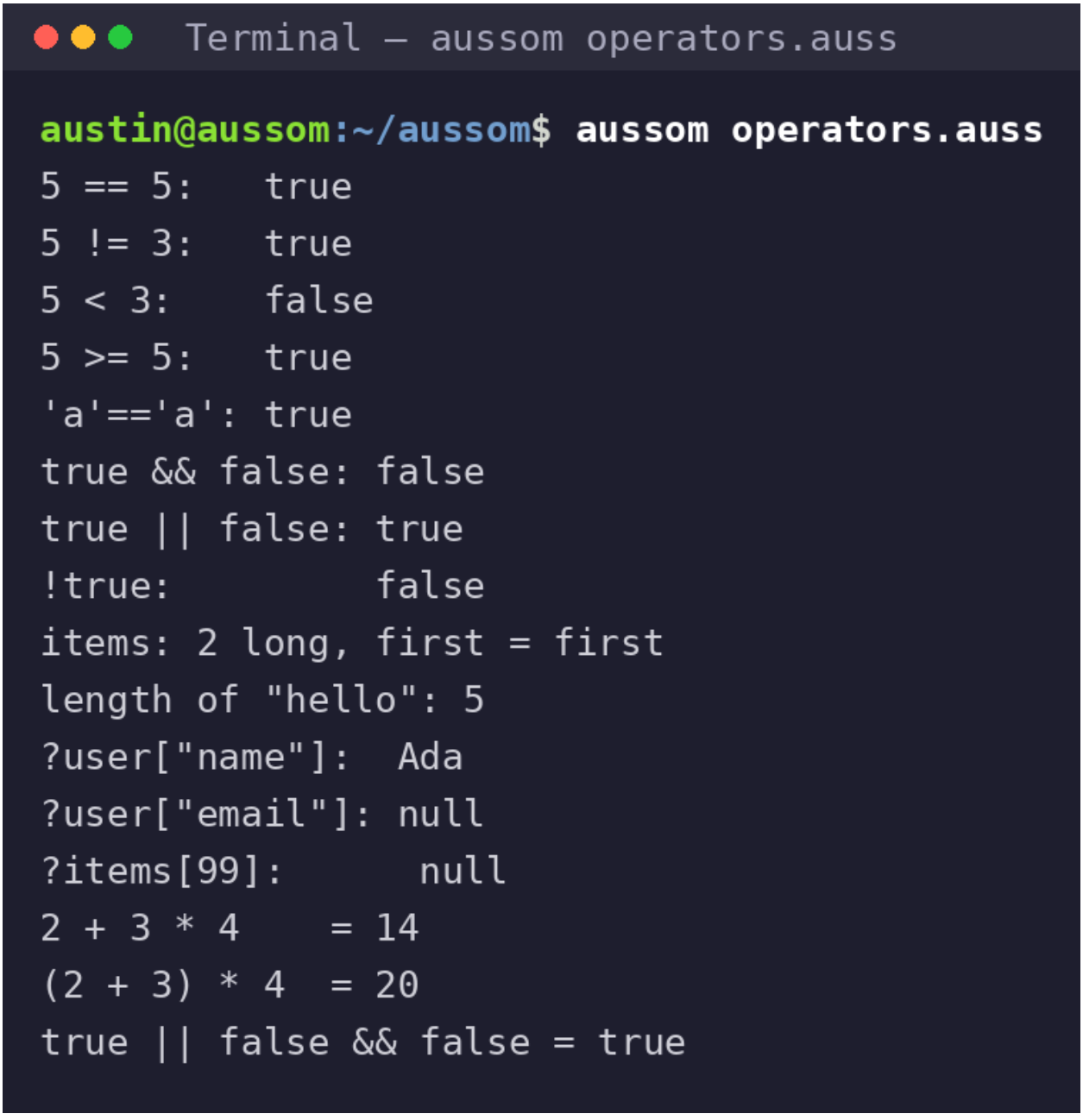


Figure 30: Running operators.auss

Part III

Collections

Chapter 10 - Lists

So far each variable has held a single value. But programs constantly deal with *groups* of values — a shopping list, the words in a sentence, the scores in a game. Aussom's first tool for groups is the **list**: an ordered series of values you can grow, read by position, loop over, and reshape.

You've already seen lists in passing — the `args` from Chapter 4 is a list, and `split` handed you one in Chapter 8. Now we'll use them properly.

Creating Lists and Adding Items

Write a list with square brackets, its items separated by commas. An empty list is just `[]`:

```
fruits = ["apple", "banana", "cherry"];
empty  = [];
```

A list can hold values of **any** type — not just numbers and strings, but booleans, other lists, maps, and even the objects you'll build in Chapter 16. The items don't have to be the same type as each other, either:

```
mixed = [42, "hello", 3.14, true, ["a", "b"], {"name": "Ada"}];
```

That single list holds an `int`, a `string`, a `double`, a `bool`, another **list**, and a **map**. Most lists you write will hold one kind of thing, but this flexibility has a very useful payoff: because a list can hold other lists, you can build a **list of lists** — a natural way to represent a grid or table.

```
grid = [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]];
grid[1]        // [4, 5, 6]   — the second row, which is itself a list
grid[1][2]     // 6           — index into the row, then into the item
```

Each item of `grid` is a three-item list, so `grid[1]` gives you the whole second row, and a *second* index, `grid[1][2]`, reaches the item inside it. (We cover indexing in detail next.)

To add an item to the end of a list, use the **append operator `@=`** from Chapter 9, or the **add** method — they do the same thing:

```
fruits @= "date";           // fruits is now [..., "cherry", "date"]
fruits.add("elderberry");   // and now ends with "elderberry"
```

To add every item from another list at once, use **`addAll`**:

```
fruits.addAll(["fig", "grape"]);
```

Indexing, Length, and Slicing

Each item in a list has a position called its **index**, and Aussom counts from **0** — so the first item is at index 0, the second at index 1, and so on. Read an item by putting its index in square brackets:

```
fruits[0]        // "apple"   — the first item
fruits[2]        // "cherry"  — the third item
```

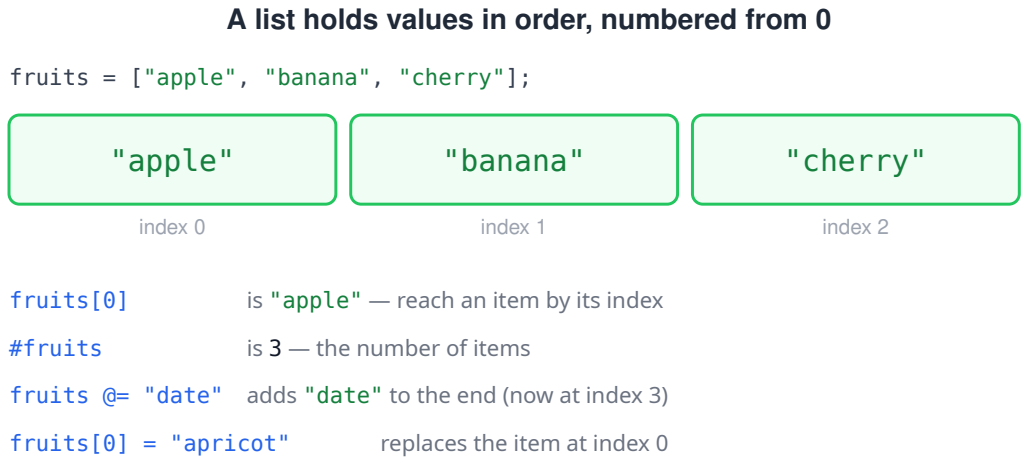


Figure 31: How a list is indexed, and what indexing, length, and append do

You can also *change* an item by assigning to its index:

```
fruits[0] = "apricot";     // replaces the first item
```

The **#** operator gives the number of items (the same as the `size` method):

```
#fruits          // 3   (or fruits.size())
```

To pull out a run of items into a new list, use **`subList(start, end)`**. Like `substr` for strings, the end index is **not** included:

```
[10, 20, 30, 40].subList(0, 2)     // [10, 20]  — indexes 0 and 1
```

Reaching for an index that doesn't exist is an error. Valid indexes run from **0** up to **`#list - 1`** — so for our three-item `fruits`, the only valid indexes are 0, 1, and 2. Ask for anything outside that range — a number too big, or a negative one — and Aussom stops the program with an error:

```
fruits = ["apple", "banana", "cherry"];   // valid indexes: 0, 1, 2
c.log(fruits[99]);                          // there is no index 99
```

Running that prints:

```
[error] astObj.evalObj(): Index out of bounds.
```

This "index out of bounds" error is one of the most common you'll run into, and it usually comes from an *off-by-one* slip — for instance, using `#fruits` (which is 3) as an index when the last valid index is `#fruits - 1` (which is 2). If you see it, check that your index is really within range for the list you're reading.

Looping Over a List

Most of the time you don't want one item — you want to do something with *every* item. The **`for`** loop walks through a list, handing you one item at a time. (We cover loops fully in Chapter 13; here's the shape you need for lists.)

```
for (fruit : fruits) {
    c.log("- " + fruit);
}
```

Read the `for (fruit : fruits)` as *"for each `fruit` in `fruits`."* The variable `fruit` takes the value of each item in turn, and the block runs once per item.

Common List Methods

Beyond adding and indexing, a list carries methods for the everyday jobs. Here are the ones you'll use most:

- **`add`** — add an item to the end (same as `@=`): `fruits.add("kiwi")`.
- **`addAll`** — add all items from another list: `fruits.addAll(more)`.
- **`contains`** — is a value in the list? `fruits.contains("cherry")` returns `true` or `false`.
- **`indexOf`** — the index of the first item equal to a value, or `-1` if it isn't in the list: `fruits.indexOf("cherry")`.
- **`remove`** — remove the first item equal to a **value**: `fruits.remove("cherry")`.
- **`removeAt`** — remove the item at a given **index**: `fruits.removeAt(1)` removes the second item.
- **`sortAsc`** — sort the items into **ascending** order, smallest to largest or alphabetical (this is the everyday "sort" you'll usually want): `nums.sortAsc()`.
- **`sort`** — sort into **descending** order, largest to smallest: `nums.sort()`.
- **`reverse`** — flip the current order of the items end-to-end: `nums.reverse()`.
- **`isEmpty`** — `true` if the list has no items.
- **`clear`** — remove every item, leaving an empty list.
- **`join`** — combine the items into a single string with a separator (from Chapter 8): `fruits.join(", ")`.

Note the two ways to remove an item: **`remove`** takes a *value* (the item you want gone), while **`removeAt`** takes an *index* (the position). For custom ordering there's also **`sortCustom`**, which takes a callback that decides the order (callbacks are Chapter 15). The full list of methods is in the API reference at aussom-lang.com.

Try It Yourself

This program creates a list, adds and changes items, checks and removes, loops, and sorts. Save it as `lists.auss` and run it:

```
fruits = ["apple", "banana", "cherry"];
c.log("list:    " + fruits.join(", "));
c.log("first:   " + fruits[0]);          // items are numbered from 0
c.log("count:   " + #fruits);

fruits @= "date";                         // append with @=
fruits.add("elderberry");                 // or with .add()
c.log("after adds: " + fruits.join(", ") + " (" + #fruits + " items)");

fruits[0] = "apricot";                    // replace an item by index
c.log("changed:   " + fruits.join(", "));

c.log("contains 'cherry': " + fruits.contains("cherry"));
c.log("indexOf 'cherry':  " + fruits.indexOf("cherry"));

fruits.remove("cherry");                  // remove by value
c.log("after remove('cherry'): " + fruits.join(", "));

fruits.removeAt(0);                       // remove by index (position
                                          // 0)
c.log("after removeAt(0):      " + fruits.join(", "));

c.log("each fruit:");
for (f : fruits) {
    c.log("  - " + f);
}

nums = [3, 1, 4, 1, 5];
nums.sortAsc();                           // ascending: smallest first
c.log("sortAsc: " + nums.join(", "));
nums.sort();                              // descending: largest first
c.log("sort:    " + nums.join(", "));
nums.reverse();                           // flip the current order
c.log("reverse: " + nums.join(", "));
```

Running it prints:

You can find this example here: `10-lists`.

What You Learned

- A **list** is an ordered group of values, written with `[ ]`; items can be any type.
- Add with **`@=`** or **`add`**; read and change items by **index** (counting from 0) with `list[i]`; count

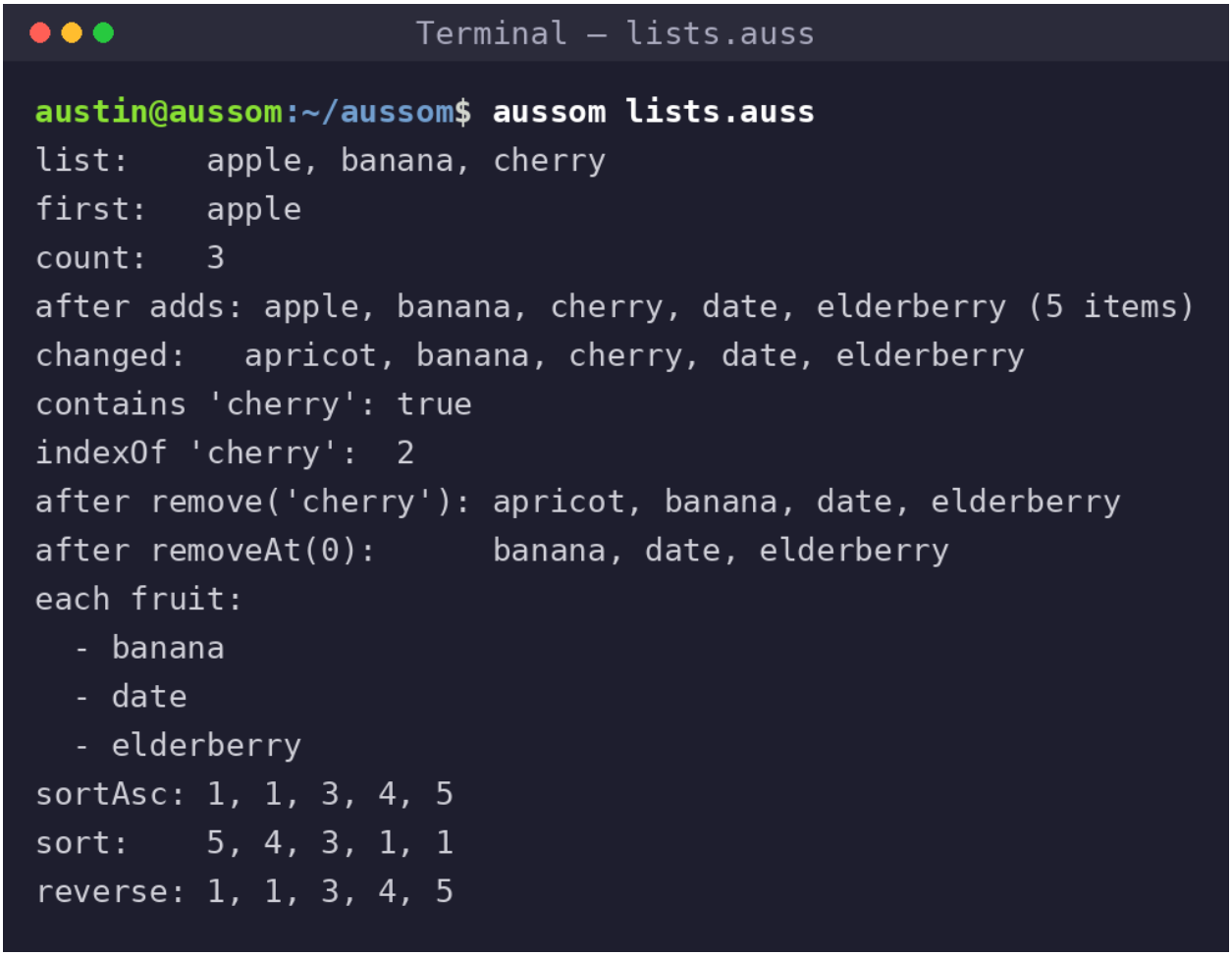


Figure 32: Running lists.auss

with **#**; slice with **`subList`** (end exclusive).
- **Loop** over a list with `for (item : list)`.
- Handy methods: **`contains`** and **`indexOf`** (find a value), **`remove`** (delete by value) and **`removeAt`** (delete by position), **`sortAsc`** (ascending), **`sort`** (descending), **`reverse`**, **`isEmpty`**, **`clear`**, and **`join`**.

Next, in **Chapter 11**, we meet the other collection type — the **map** — which looks up values by a name instead of a position.

Chapter 11 - Maps

A list is perfect when order and position matter. But often you want to look something up by a *name* instead — a person's age by "age," a price by a product code, a setting by its label. For that, Aussom gives you the **map**: a collection of values, each stored under a **key** you choose.

You've already met maps in Chapter 6. This chapter shows how to build them, read from them safely, and work through their contents.

Creating Maps and Setting Keys

Write a map with curly braces. Inside, each entry is a **key**, then a colon, then a **value**; entries are separated by commas. Keys are always **strings**; values can be any type. An empty map is `{}`:

```
person = {"name": "Ada", "age": 36};
empty  = {};
```

That reads as: under the key `"name"` is the value `"Ada"`, and under `"age"` is 36.

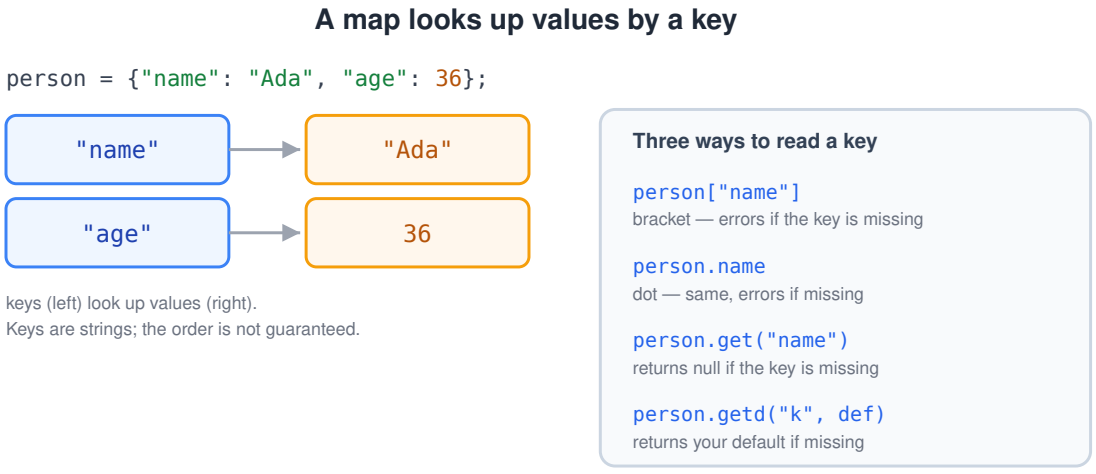


Figure 33: A map stores values under string keys, with several ways to read them

To add a new entry or change an existing one, assign to a key with square brackets, or use the **`put`** method:

```
person["city"] = "Paris";      // add a new key
person.put("job", "coder");    // same thing, method style
person["age"] = 37;            // assigning to an existing key updates
                               // it
```

Related helpers you can reach for:

- **`putIfAbsent`** — set a key only if it isn't already there.
- **`putAll`** — copy every entry from another map in at once.

Values Can Be Any Type

A key is always a string, but a **value** can be any Aussom type — numbers, strings, booleans, lists, and even other maps. That's what lets a map model real-world, structured data. For example, a map whose values are **lists**:

```
profile = {
    "name":   "Ada",
    "tags":   ["coder", "writer"],
    "scores": [90, 85, 88]
};
```

or a **map of maps**, where each value is itself a map:

```
people = {
    "alice": {"age": 30, "city": "Paris"},
    "bob":   {"age": 25, "city": "Rome"}
};
```

You reach into a nested structure one step at a time, chaining keys — and indexes for any lists along the way. Bracket and dot access both work at each level:

```
people["alice"]["age"]     // 30
people.bob.city            // "Rome"
profile.tags[0]            // "coder"  — first item of the tags list
profile["scores"][1]       // 85
```

Nesting can go as deep as you need — a list of maps, a map of maps of lists, and so on. This is exactly how data formats like JSON (Chapter 26) represent structured information: maps and lists nested inside one another. (We cover reading keys in detail in the next section.)

Reading Values: Bracket and Dot Access

There are a few ways to read a value out of a map, and the difference between them matters — especially when a key might be missing.

Bracket access uses the key in square brackets. **Dot access** works when the key is a simple name, just like reading a field:

```
person["name"]     // "Ada"
person.age         // 36   — dot access, same result
```

Both are quick, but they share one catch: **if the key isn't in the map, they raise an error** rather than giving you an empty answer. So use them when you're sure the key is there.

When a key *might* be missing, use one of the safe readers instead:

- **`get`** — returns the value, or **`null`** if the key isn't there:

  ```
  person.get("zip")             // null  (no "zip" key)
  ```

- **`getd`** — returns the value, or a **default you provide** if the key isn't there:

  ```
  person.getd("zip", "n/a")     // "n/a"
  ```

`getd` (think "get with default") is especially handy — it lets you read a possibly-missing setting and fall back to a sensible value in one step.

Checking, Removing, and Listing Keys

To ask whether a key exists before you use it, call **`containsKey`**:

```
person.containsKey("city")     // true
person.containsKey("zip")      // false
```

There's also **`containsVal`** to check whether a given *value* appears anywhere in the map.

Remove an entry by its key with **`remove`**:

```
person.remove("age");          // the "age" entry is gone
```

To see everything in a map, two methods hand you lists you can loop over or join:

- **`keySet`** — a list of all the keys.
- **`values`** — a list of all the values.

```
person.keySet()      // ["name", "city", "job"]
person.values()      // ["Ada", "Paris", "coder"]
```

And the number of entries is the **#** operator or the `size` method:

```
#person              // 3   (or person.size())
```

Looping Over a Map

A `for` loop over a map walks through its **keys**, one at a time. From each key you can reach its value with bracket access:

```
scores = {"alice": 90, "bob": 85};
for (name : scores) {
    c.log(name + " scored " + scores[name]);
}
```

One thing to keep in mind: **a map does not keep its entries in any particular order.** When you loop or call `keySet`, the keys can come back in a different order than you added them. If order matters to you, a list is the better fit.

Try It Yourself

This program builds a map, reads it two ways, uses the safe readers, checks and removes a key, lists keys and values, and loops. Save it as `maps.auss` and run it:

```
person = {"name": "Ada", "age": 36};
c.log("name (bracket): " + person["name"]);
c.log("age (dot):      " + person.age);
c.log("size:           " + #person);

person["city"] = "Paris";                // add or update a key with
                                         // brackets
person.put("job", "coder");              // or with .put()
c.log("after adds: size " + #person);

c.log("has 'city': " + person.containsKey("city"));
c.log("has 'zip':  " + person.containsKey("zip"));

// Safe ways to read a key that might not be there:
// get() returns null if the key is missing...
c.log("get('zip'):         " + person.get("zip"));
// ...and getd() returns a default you supply instead.
c.log("getd('zip', 'n/a'): " + person.getd("zip", "n/a"));

person.remove("age");                    // remove a key
c.log("after remove 'age', has age: " + person.containsKey("age"));

c.log("keys:   " + person.keySet().join(", "));
c.log("values: " + person.values().join(", "));

c.log("entries:");
for (key : person) {
    c.log("  " + key + " = " + person[key]);
}
```

Running it prints:

(Notice the keys came back as `city, name, job`, not the order they were added — that's the "no

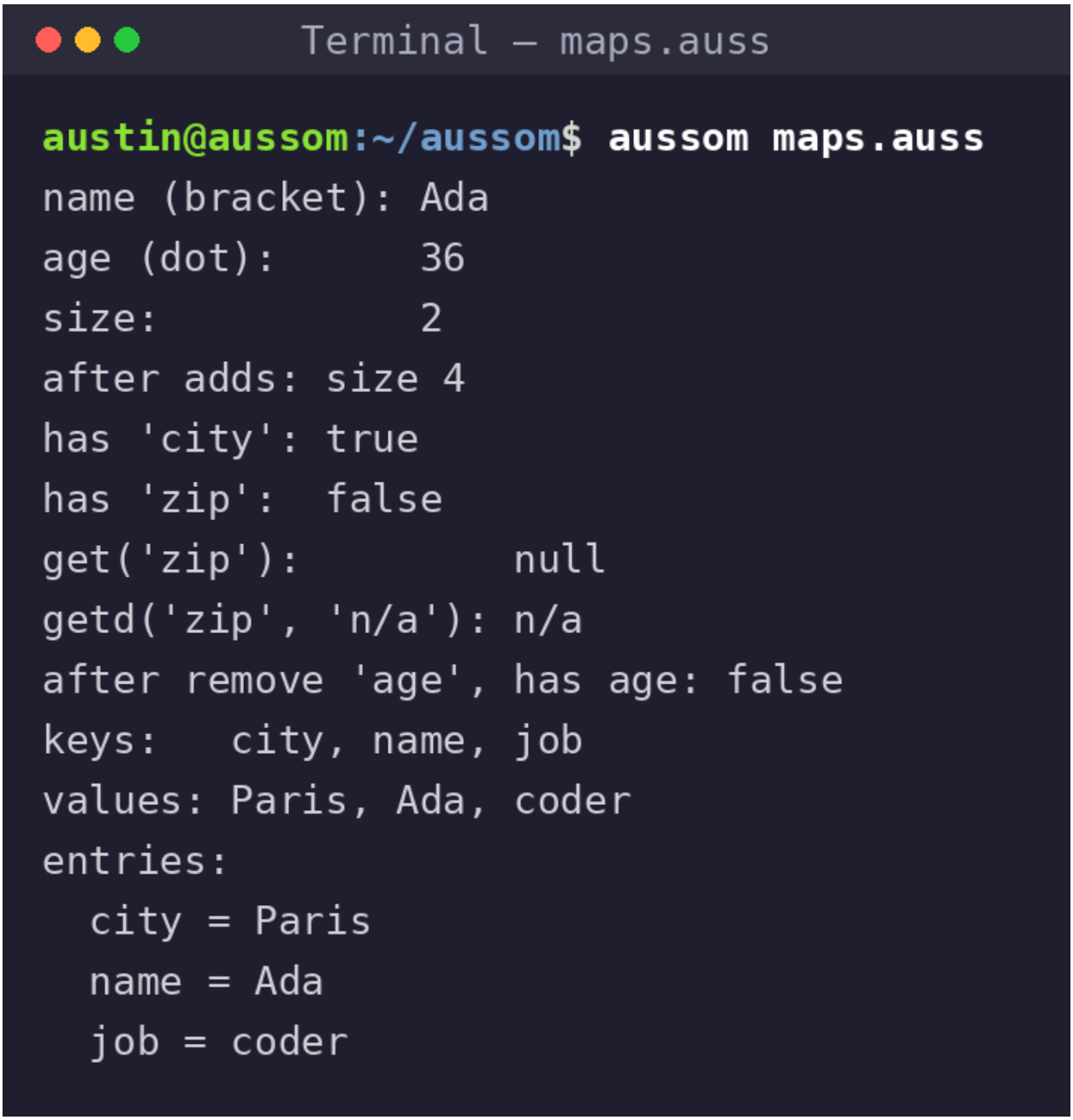


Figure 34: Running maps.auss

guaranteed order" rule in action.)

You can find this example here: `11-maps`.

What You Learned

- A **map** stores values under string **keys**, written with `{ }`.
- Add or update a key with `map["key"] = value` or **`put`**.
- Read with **bracket** or **dot** access (which **error on a missing key**), or safely with **`get`** (null if missing) or **`getd`** (a default if missing).
- Check with **`containsKey`**, remove with **`remove`**, and list contents with **`keySet`** and **`values`**; count with **`#`**.
- Loop with `for (key : map)` — but remember a map's order is **not guaranteed**.

That wraps up Part III. You now have both of Aussom's collection types: **lists** for ordered groups and **maps** for named lookups. Next, in **Part IV**, we put your values to work by making decisions and repeating actions — starting with **Chapter 12, Making Decisions**.

Part IV

Control Flow

Chapter 12 - Making Decisions

So far your programs have run straight through, top to bottom, doing the same thing every time. Real programs need to *choose* — show a welcome only if someone is signed in, charge shipping only for heavy orders, print a warning only when something's wrong. Making choices is called **control flow**, and its most basic tool is the **`if` statement**.

You've seen `if` in passing since Chapter 4. This chapter covers it properly.

`if`, `else if`, and `else`

An **`if` statement** runs a block of code *only when a condition is true*. The condition goes in parentheses, and the block to run goes in curly braces:

```
temp = 30;
if (temp > 25) {
    c.log("It's warm out.");
}
```

If `temp > 25` is true, the block runs; if not, Aussom skips it and moves on. (`temp > 25` is a comparison expression from Chapter 9 — it produces `true` or `false`.)

To do something *else* when the condition is false, add an **`else`** block:

```
if (temp > 25) {
    c.log("It's warm out.");
} else {
    c.log("Bring a jacket.");
}
```

Exactly one of the two blocks runs. And when you have several possibilities to check in order, chain them with **`else if`**:

```
if (score >= 90) {
    c.log("Grade: A");
} else if (score >= 80) {
    c.log("Grade: B");
} else if (score >= 70) {
    c.log("Grade: C");
} else {
```

```
    c.log("Grade: F");
}
```

Aussom checks each condition **from the top down** and runs the block for the **first** one that's true — then skips all the rest. So with `score = 82`, the first test (`>= 90`) is false, the second (`>= 80`) is true, and you get `Grade: B`; the `>= 70` and `else` are never even checked.

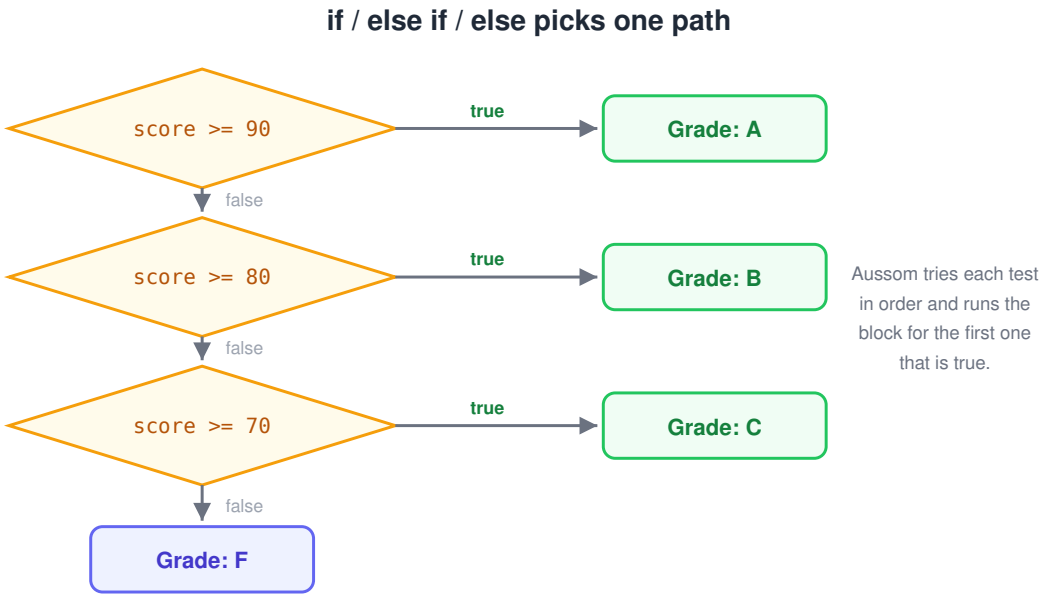


Figure 35: How if / else if / else picks exactly one path

The `else` at the end is optional. Use it as a catch-all for "none of the above."

Truthiness in Conditions

The condition in an `if` doesn't have to be a comparison — it can be *any* value. Aussom decides whether that value counts as true or false using the **truthiness** rule from Chapter 6. As a reminder, these values are **falsy**: `false`, `null`, `0`, `0.0`, `""` (empty string), `[]` (empty list), and `{}` (empty map). Everything else is **truthy**.

That makes some checks read very naturally. To do something only when a variable actually has a value, just test the variable:

```
name = "";
if (name) {
    c.log("Hello, " + name);
} else {
    c.log("No name was given.");   // this runs: "" is falsy
}
```

Here `if (name)` means "if `name` has a non-empty value." You don't have to write `if (name != "")` — the truthiness rule handles it.

Nesting and Combining Conditions

Sometimes one decision depends on another. You can put an `if` **inside** another `if` — this is called **nesting**:

```
if (temp > 25) {
    if (temp > 35) {
        c.log("It's very hot.");
    } else {
        c.log("It's warm.");
    }
}
```

The inner `if` only runs when the outer condition is already true.

Often, though, it's clearer to **combine** conditions into one test using the logical operators from Chapter 9 — **`&&`** (and), **`||`** (or), and **`!`** (not):

```
age = 20;
hasTicket = true;
if (age >= 18 && hasTicket) {
    c.log("Welcome to the show!");
} else {
    c.log("Sorry, you can't come in.");
}
```

`age >= 18 && hasTicket` is true only when **both** parts are true. Use `||` when **either** part is enough, and `!` to flip a condition. When you combine && and `||` in one condition, wrapping the parts in parentheses keeps the order clear.

Choosing with `switch`

When you're checking one value against several fixed options, a long chain of `else ifs` works but gets repetitive. The **`switch`** statement is a tidier way to say "look at this one value, and run the block that matches it."

Write `switch` with the value in parentheses, then a set of **`case`** blocks — one for each value you want to handle. Each case lists its value, a colon, and its own `{ }` block. An optional **`default`** block handles "none of the above":

```
day = "TUE";
switch (day) {
    case "MON": { c.log("Start of the work week."); }
    case "TUE": { c.log("Taco Tuesday!"); }
    case "SAT": { c.log("Weekend!"); }
    default:    { c.log("Just another day."); }
}
```

Aussom compares `day` to each case's value and runs the one that matches — here `case "TUE"`,

which prints `Taco Tuesday!`. If nothing matched, the `default` block would run instead.

Two things to know, especially if you've used `switch` in another language:

- **Only the matching block runs.** Aussom does not "fall through" from one case into the next, so you do **not** write `break` after each case — each `case` block stands on its own.
- The value can be a string, a number, or any comparable value, and `default` is optional. If nothing matches and there's no `default`, the `switch` simply does nothing.

Reach for `switch` when you're matching one value against a list of specific options. Stick with `if` / `else if` when your tests are ranges or more complex conditions, like `score >= 80` — those don't fit the "one value, fixed options" shape that `switch` is built for.

Try It Yourself

Save this as `decisions.auss` and run it:

```
score = 82;
if (score >= 90) {
    c.log("Grade: A");
} else if (score >= 80) {
    c.log("Grade: B");
} else if (score >= 70) {
    c.log("Grade: C");
} else {
    c.log("Grade: F");
}

// Truthiness: an empty string is falsy.
name = "";
if (name) {
    c.log("Hello, " + name);
} else {
    c.log("No name was given.");
}

// Combining conditions with && (and) and || (or).
age = 20;
hasTicket = true;
if (age >= 18 && hasTicket) {
    c.log("Welcome to the show!");
} else {
    c.log("Sorry, you can't come in.");
}

// Nesting: a decision inside a decision.
temp = 30;
if (temp > 25) {
    if (temp > 35) {
```

```
        c.log("It's very hot.");
    } else {
        c.log("It's warm.");
    }
}
```

Running it prints:

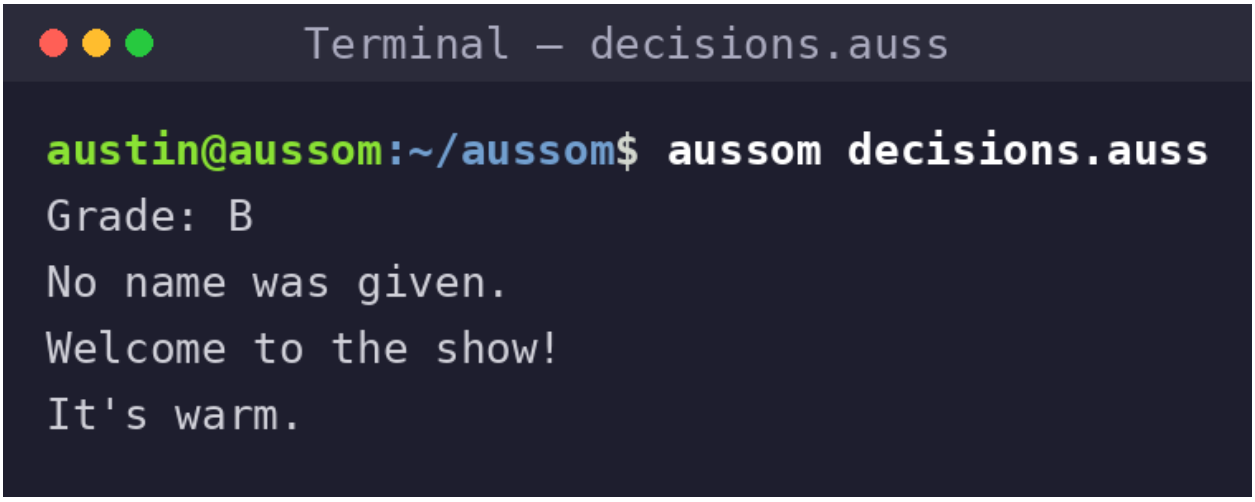


Figure 36: Running decisions.auss

You can find this example here: `12-decisions`.

What You Learned

- **`if`** runs a block only when its condition is true; **`else`** handles the false case; **`else if`** chains several checks, and the **first** true one wins.
- A condition can be any value, judged by the **truthiness** rule — so `if (name)` means "if `name` has a value."
- **Nest** an `if` inside another for dependent decisions, or **combine** conditions with **`&&`**, **`||`**, and **`!`**.

Next, in **Chapter 13**, we make programs *repeat* work with **loops**.

Chapter 13 - Loops

Computers are brilliant at doing the same thing over and over without getting bored. A **loop** is how you ask for that: run a block of code repeatedly, either a set number of times or once for each item in a collection. Loops are what turn a list of three fruits — or three thousand — into a single, short piece of code.

Every loop works the same way underneath: check a condition, and if it's true, run the body and check again; when the condition becomes false, stop.

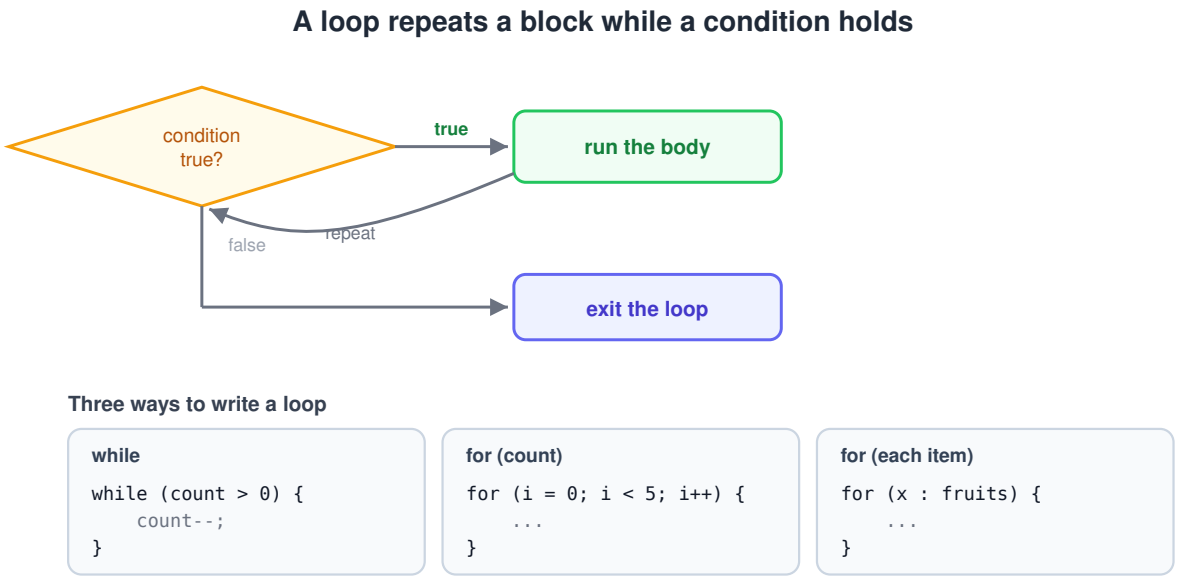


Figure 37: A loop checks a condition, runs its body, and repeats until the condition is false

The `while` Loop

The **`while` loop** is the simplest. It repeats its block *as long as* a condition stays true:

```
count = 3;
while (count > 0) {
    c.log("T-minus " + count);
    count--;                        // count down toward 0
```

```
}
c.log("Liftoff!");
```

Aussom checks `count > 0`, runs the block, then checks again — over and over — until `count` reaches 0 and the condition is false. That prints `T-minus 3`, 2, 1, then `Liftoff!`.

The body **must** change something that affects the condition (here, `count--`). If it never does, the condition stays true forever and the loop never stops — that's an *infinite loop*. If a program ever seems stuck, a runaway loop is the usual suspect; press `Ctrl+C` in the terminal to stop it.

The `for` Loop

The **`for` loop** packs the three parts of a counting loop onto one line: a **setup**, a **condition**, and a **step**, separated by semicolons.

```
for (i = 1; i <= 5; i++) {
    c.log("i is " + i);
}
```

Read it as: start with `i = 1`; keep going *while* `i <= 5`; and after each pass, do `i++` (add one). It's the tidy way to repeat something a known number of times, or to add up a series:

```
total = 0;
for (i = 1; i <= 5; i++) {
    total += i;
}
c.log("Sum of 1..5 = " + total);     // 15
```

The Three Parts, and Which You Can Skip

A `for` loop always has **two semicolons**, dividing it into three slots:

`for ( setup ; condition ; step ) { ... }`

- The **setup** runs once, before the loop starts — usually to create the counter.
- The **condition** is checked before every pass; the loop keeps going while it's true.
- The **step** runs after every pass — usually to move the counter along.

Each of the three slots is **optional**, but the two semicolons are not — they always mark the three positions. You might leave a slot empty when its work happens elsewhere:

```
i = 0;
for (; i < 3; i++) { c.log(i); }     // setup done before the loop

for (i = 0; i < 3; ) { c.log(i); i++; }  // step done inside the body
```

Both behave exactly like the full three-part version.

The **condition** behaves specially when you leave it out: an empty condition counts as **always true**, so the loop keeps running until something stops it. That makes `for (;;)` — all three slots empty —

an endless loop, which you exit with a **`break`** inside the body. It's the same idea as `while (true)` from the previous section, just written as a `for`.

Common Variations

The counter doesn't have to start at 1, go up by 1, or count up at all. Adjust the setup, condition, and step to fit the job:

```
for (i = 10; i > 0; i -= 2) {        // count DOWN by 2: 10, 8, 6, 4, 2
    c.log(i);
}

for (i = 0; i < 100; i += 10) {      // step by 10: 0, 10, 20, ... 90
    c.log(i);
}
```

Counting down with `i--`, skipping by any amount with `i += n` or `i -= n`, or starting from any value — it's all the same loop, just with different pieces in the three slots.

Looping Over Lists and Maps

Most often you don't want to count — you want to visit every item in a collection. The **for-each** form of the `for` loop does exactly that, handing you one item at a time. You saw it in Chapters 10 and 11; here it is together.

Over a **list**, each item goes into your variable in turn:

```
fruits = ["apple", "banana", "cherry"];
for (fruit : fruits) {
    c.log("- " + fruit);
}
```

Read `for (fruit : fruits)` as *"for each `fruit` in `fruits`."*

Over a **map**, the loop walks the **keys**; reach each value with the key:

```
prices = {"apple": 2, "banana": 1};
for (item : prices) {
    c.log(item + " costs " + prices[item]);
}
```

Remember from Chapter 11 that a map has no guaranteed order, so the entries may come out in a different order than you wrote them.

Breaking Out of a Loop

Sometimes you want to stop a loop early — the moment you've found what you were looking for, there's no reason to keep going. The **`break`** keyword ends the loop immediately:

```
nums = [20, 60, 130, 40];
for (n : nums) {
    if (n > 100) {
        c.log("First value over 100: " + n);
        break;                      // stop looping right away
    }
}
```

As soon as `break` runs, the loop is over and the program continues after it.

> **A note for readers coming from other languages:** some languages have a `continue` keyword that skips to the next pass of a loop. Aussom does **not** have `continue`. To skip work on some passes, wrap the body in an `if` instead — for example, `for (n : nums) { if (n > 0) { ... } }` processes only the positive numbers, which is the same effect.

Try It Yourself

Save this as `loops.auss` and run it:

```
// while: repeat as long as a condition is true.
count = 3;
while (count > 0) {
    c.log("T-minus " + count);
    count--;
}
c.log("Liftoff!");

// classic for: count with a variable.
total = 0;
for (i = 1; i <= 5; i++) {
    total += i;
}
c.log("Sum of 1..5 = " + total);

// for-each over a list.
fruits = ["apple", "banana", "cherry"];
for (fruit : fruits) {
    c.log("- " + fruit);
}

// for-each over a map walks its keys.
prices = {"apple": 2, "banana": 1};
for (item : prices) {
    c.log(item + " costs " + prices[item]);
}
```

```
// break: stop the loop early.
nums = [20, 60, 130, 40];
for (n : nums) {
    if (n > 100) {
        c.log("First value over 100: " + n);
        break;
    }
}
```

Running it prints:

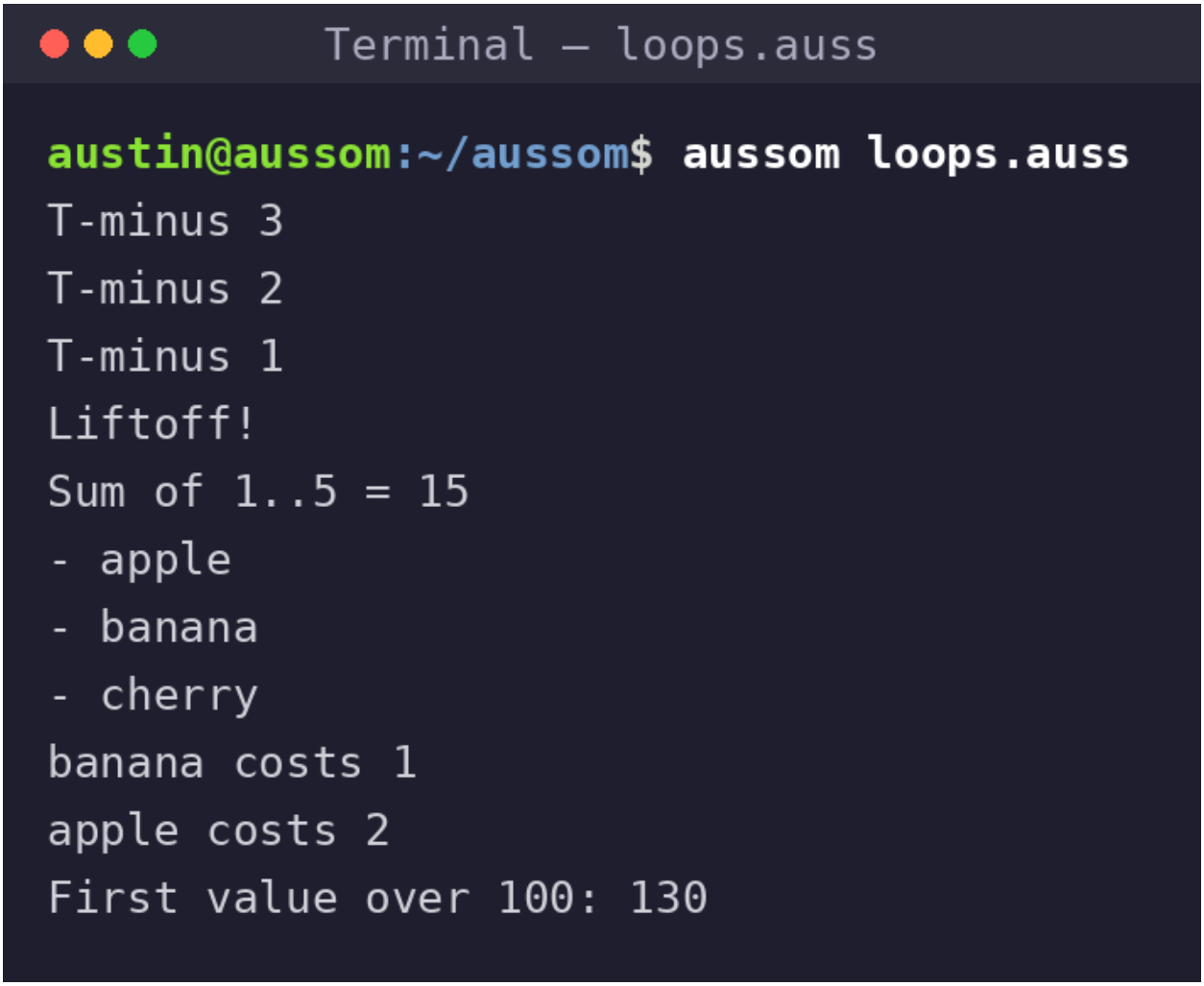


Figure 38: Running loops.auss

(The map entries came out `banana` then `apple` — a reminder that maps don't keep a set order.)

You can find this example here: `13-loops`.

What You Learned

- A **`while`** loop repeats while a condition is true — be sure the body changes that condition, or the loop runs forever.
- A **`for`** loop counts with a setup, condition, and step; the **for-each** form, `for (item : collection)`, visits every item in a list or map.
- **`break`** ends a loop early. Aussom has no `continue`; wrap the body in an `if` to skip work on some passes.

That wraps up Part IV. Next, in **Part V**, **Chapter 14** shows how to bundle code into reusable **functions**.

Part V

Functions and Callbacks

Chapter 14 - Functions

As programs grow, you find yourself writing the same few lines again and again, or wishing you could give a chunk of logic a name. A **function** solves both. A function is a named, reusable block of code that can take **inputs**, do some work, and hand back a **result**. Write it once, then call it by name whenever you need it.

You've been *calling* functions since Chapter 4 — `c.log(...)`, `s.toUpper()`, `nums.sort()` are all function calls. In this chapter you'll write your own.

Functions Live Inside a Class

Here's the one new idea to get comfortable with first: in Aussom, **a function lives inside a class.** A class is a container that groups related functions (and data) together. We'll explore classes fully in Chapter 16; for now you only need the small recipe below, and you can treat the class as a folder that holds your functions.

A function defined inside a class is also called a **method** — the two words mean the same thing here. Here's the smallest useful example:

```
class Calc {
    public add(A, B) {
        return A + B;
    }
}

calc = new Calc();              // make a Calc object
c.log(calc.add(2, 3));          // call its add function -> 5
```

Three moves make this work, and they'll be the same every time:

1. **Define** the class with your functions inside it (`class Calc { ... }`).
2. **Create an object** from the class with **new**: `calc = new Calc();`. An object is a working copy of the class you can call.
3. **Call** a function on the object with a dot: `calc.add(2, 3)`.

That's the scaffolding. The rest of this chapter is about the functions themselves.

Defining and Calling Functions

Look again at the definition:

```
public add(A, B) {
    return A + B;
}
```

Every function definition has the same shape: an access word (`public`), the **name** (`add`), a list of **parameters** in parentheses (`A, B`), and a **body** in braces. You *call* it by name, passing values for the parameters.

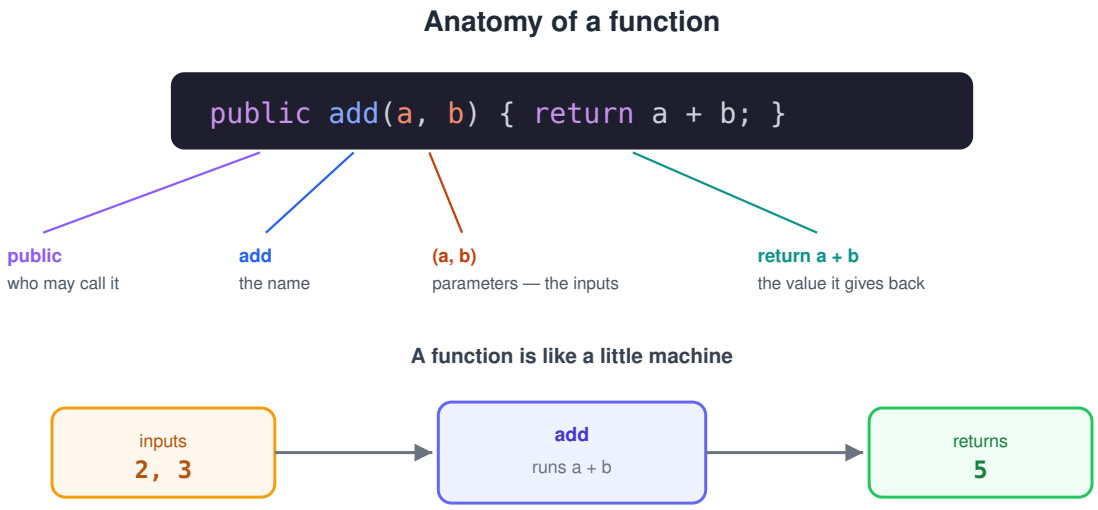


Figure 39: The parts of a function, and how inputs flow to a return value

Arguments and Return Values

Parameters are the named inputs a function expects; the actual values you pass when calling are the **arguments**. In `add(A, B)`, the parameters are `A` and `B`; in `calc.add(2, 3)`, the arguments are 2 and 3, so inside the function `A` is 2 and `B` is 3.

A function hands a value back with the **`return`** keyword. `return` does two things: it sends the value back to whoever called the function, and it *ends the function right there*.

```
public add(A, B) {
    return A + B;        // hands back the sum, and stops
}
```

The returned value takes the place of the call, so you can use it directly:

```
c.log("The total is " + calc.add(10, 5));    // "The total is 15"
```

Not every function needs to return something — some just *do* a thing, like print. A function with no `return` gives back **`null`** (the "no value" type from Chapter 6).

Default parameter values. You can give a parameter a default, used when the caller leaves that argument out. Put an = and a value after the parameter name:

```
public increment(N, Step = 1) {      // Step defaults to 1
    return N + Step;
}
```

Now `increment(10)` returns 11 (using the default `Step` of 1), while `increment(10, 5)` returns 15. Defaults are handy for the "usually this, but sometimes that" case.

Access Modifiers: `public` and `private`

The word in front of a function is its **access modifier**. It controls who is allowed to call the function:

- **`public`** — the function can be called from **outside** the class (like `calc.add(...)`). Use this for the functions you want others to use.
- **`private`** — the function can only be called from **inside** the same class. Use it for internal helpers that aren't meant for the outside world.

```
class Calc {
    private square(N) {              // helper, hidden from outside
        return N * N;
    }
    public sumOfSquares(A, B) {      // the public entry point
        return this.square(A) + this.square(B);
    }
}
```

Here `sumOfSquares` is `public`, so callers can use it, but `square` is `private` — trying to call `calc.square(3)` from outside is an error. Keeping helpers private lets you change them later without affecting anyone who uses your class.

Calling Other Functions with `this`

Inside a class, one function often needs to call another. To do that, put **`this.`** in front of the name. **`this`** means "the current object" — the very object the function is running on:

```
public addThenDouble(A, B) {
    sum = this.add(A, B);        // call add on this same object
    return sum * 2;
}
```

Without `this.`, Aussom wouldn't know you mean a function on the object. You'll see `this` again in Chapter 16, where it also reaches the object's stored data.

Function Overloading

Aussom lets you define **several functions with the same name**, as long as their parameter lists differ. When you call the function, Aussom looks at the arguments you passed and picks the matching version. This is called **overloading**, and it's handy when one idea has a few natural forms:

```
class Greeter {
    public greet() { return "Hello there!"; }
    public greet(Name) { return "Hello, " + Name + "!"; }
    public greet(First, Last) { return "Hello, " + First + " " + Last + "!"; }
}
```

Each call goes to the version whose **number of parameters** matches:

```
g = new Greeter();
c.log(g.greet());                    // Hello there!
c.log(g.greet("Ada"));               // Hello, Ada!
c.log(g.greet("Ada", "Lovelace"));   // Hello, Ada Lovelace!
```

Aussom can also tell versions apart by the **type** of a parameter. Put a type name (`int`, `string`, `double`, `bool`, `list`, `map`, `object`, `callback`) in front of the parameter, and the matching version is chosen by the argument's type:

```
class Printer {
    public show(int N)    { return "the number " + N; }
    public show(string S) { return "the text \"" + S + "\""; }
}
```

```
p = new Printer();
c.log(p.show(42));      // the number 42
c.log(p.show("hi"));    // the text "hi"
```

You can't have two versions with the *same* parameter shape — that's an error, since Aussom couldn't tell them apart. Overloading is why a single built-in like `c.log` happily accepts a number, a string, or a list: the standard library defines several versions behind that one name.

Variadic Functions: Accepting Any Number of Arguments

Sometimes you don't know ahead of time how many arguments you'll get. A **variadic** function ends its parameter list with `...` to accept *any number* of extra arguments. Inside the function, those extras arrive as a ready-made list named **`etc`**:

```
class Adder {
    public sum(...) {
        total = 0;
        for (n : etc) { total += n; }   // etc holds every argument
                                        // passed
        return total;
    }
```

```
}
```

```
a = new Adder();
c.log(a.sum());              // 0    — etc is empty
c.log(a.sum(1, 2, 3));       // 6
c.log(a.sum(10, 20, 30));    // 60
```

You can require some fixed parameters first and collect the rest with `...`. The named parameters fill up in order, and everything left over lands in `etc`:

```
public join(Sep, ...) {
    out = "";
    for (i = 0; i < #etc; i++) {
        if (i > 0) { out += Sep; }
        out += etc[i];
    }
    return out;
}
// join("-", "a", "b", "c")  ->  "a-b-c"
```

Here `Sep` takes the first argument, and `etc` holds the rest. Because `etc` is an ordinary list, everything from Chapter 10 works on it — `#etc` for the count, `etc[i]` to index, or a for-each loop. You'll spot `...` in the standard library, where it's how one method name (like a print helper) swallows a whole line's worth of values.

Try It Yourself

This `Calc` class shows returns, defaults, private helpers, `this`, an overload, and a variadic function. Save it as `functions.auss` and run it:

```
class Calc {
    // A function that takes two inputs and returns their sum.
    public add(A, B) {
        return A + B;
    }

    // An overload of add: same name, three inputs.
    public add(A, B, C) {
        return A + B + C;
    }

    // A variadic function: sums however many numbers you pass.
    public sumAll(...) {
        total = 0;
        for (n : etc) { total += n; }
        return total;
    }
```

```
    // A default parameter: if Step is left out, it is 1.
    public increment(N, Step = 1) {
        return N + Step;
    }

    // One function calling another with this.
    public addThenDouble(A, B) {
        sum = this.add(A, B);
        return sum * 2;
    }

    // A private helper: usable inside the class, hidden from outside.
    private square(N) {
        return N * N;
    }

    public sumOfSquares(A, B) {
        return this.square(A) + this.square(B);
    }
}

calc = new Calc();
c.log("add(2, 3)           = " + calc.add(2, 3));
c.log("add(2, 3, 4)        = " + calc.add(2, 3, 4));
c.log("sumAll(1,2,3,4,5)   = " + calc.sumAll(1, 2, 3, 4, 5));
c.log("increment(10)       = " + calc.increment(10));
c.log("increment(10, 5)    = " + calc.increment(10, 5));
c.log("addThenDouble(2, 3) = " + calc.addThenDouble(2, 3));
c.log("sumOfSquares(3, 4)  = " + calc.sumOfSquares(3, 4));
```

Running it prints:

You can find this example here: `14-functions`.

What You Learned

- A **function** is a named, reusable block that takes inputs and returns a result. In Aussom, functions live inside a **class**; create an object with **`new`** and call a function with a dot.
- **Parameters** are the named inputs; **arguments** are the values you pass. **`return`** hands a value back and ends the function; no `return` gives `null`. Parameters can have **default values**.
- **`public`** functions can be called from outside; **`private`** ones are internal helpers.
- Call another function on the same object with **`this.name(...)`**.
- **Overloading** lets several functions share a name when their parameter lists differ (by count or type); Aussom picks the match. A **variadic** function ends with **`...`** to accept any number of extra arguments, which arrive as a list named **`etc`**.

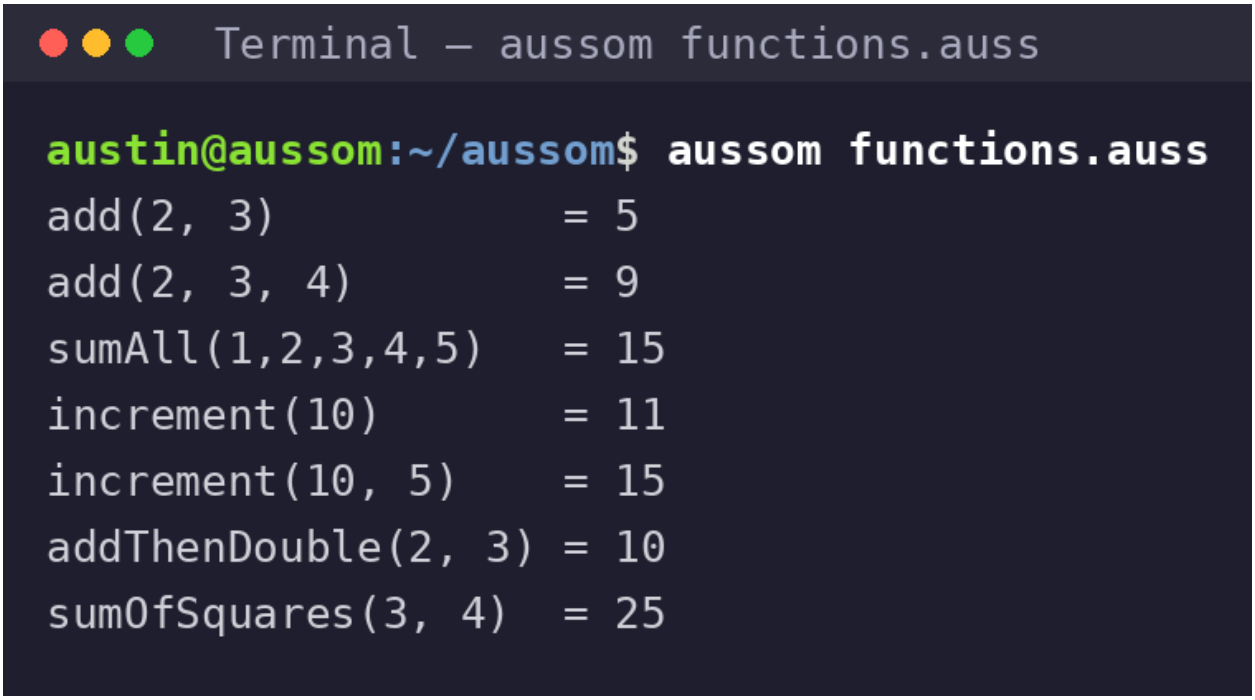


Figure 40: Running functions.auss

Next, in **Chapter 15**, we treat functions as values you can pass around — **callbacks**.

Chapter 15 - Callbacks

In Chapter 14 you called functions by name. But a function can also be treated as a **value** — something you store in a variable and hand to *other* code, which calls it later, on its own schedule. A function used this way is called a **callback**, and it's one of the most powerful ideas in programming.

Why would you want this? Because it lets you write one flexible piece of code that does a general job — "do *something* to every item," "sort using *this* rule" — and let the caller plug in the specific behavior. You met the `callback` type back in Chapter 6; now you'll put it to work.

What a Callback Is

A **callback** is a reference to a function: not the result of calling it, but a handle to the function itself. Think of it like the number for a pizza place saved in your phone. The saved number isn't a pizza — it's a way to *reach* the shop so you (or a friend you give it to) can place an order whenever you want.

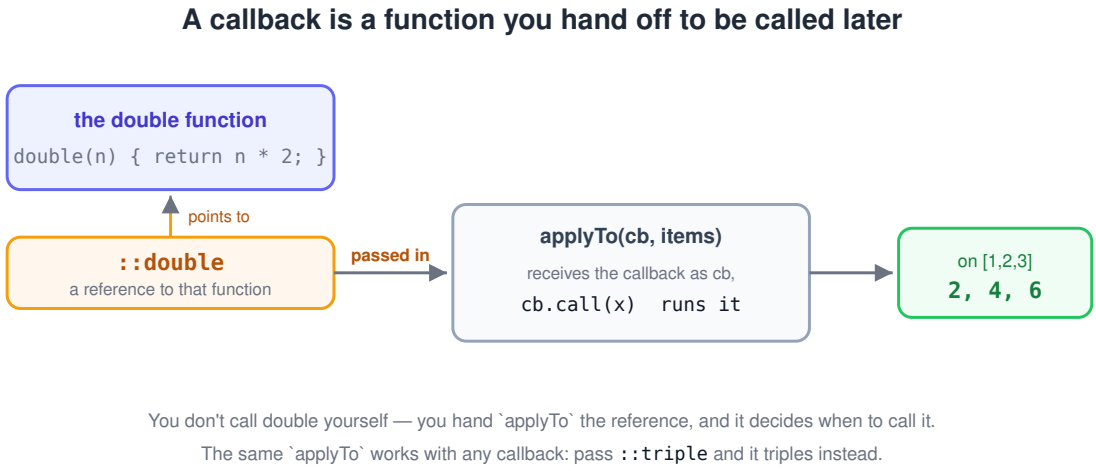


Figure 41: A callback is a reference to a function that other code calls later

Creating a Callback with `::`

You make a callback with the `::` operator, followed by the name of a function. Inside a class, `::name` refers to a function on the current object:

```
class Tools {
    public double(N) { return N * 2; }

    public demo() {
        cb = ::double;                  // a callback to the double
                                        // function
        c.log(lang.type(cb));           // callback
    }
}
```

`cb` now holds a reference to the `double` function. Notice we did **not** write `double()` with parentheses — that would *call* it. `::double` grabs the function itself, to be called later.

Calling a Callback with `.call`

To actually run a callback, use its **`.call`** method, passing any arguments the function expects:

```
cb = ::double;
result = cb.call(5);        // runs double(5)
c.log(result);              // 10
```

`cb.call(5)` reaches through the reference and runs the real `double` function with `5`, giving back `10`. (You must use `.call` — a callback can't be run by writing `cb(5)` directly.)

Passing a Callback to Other Code

On its own, a callback of your own function isn't very exciting — you could just call the function. The power shows up when you **pass** a callback to another function, which then calls it for you. That other function doesn't need to know *what* your function does; it just calls it at the right moment.

Here's a function `applyTo` that runs *any* callback on every item of a list:

```
public applyTo(Cb, Items) {
    result = [];
    for (x : Items) {
        result @= Cb.call(x);       // call the given callback on each
                                    // item
    }
    return result;
}
```

`applyTo` has no idea whether it's doubling, tripling, or something else — it just calls whatever callback it was handed. So the same `applyTo` does different jobs depending on which callback you pass:

```
this.applyTo(::double, [1, 2, 3])     // [2, 4, 6]
this.applyTo(::triple, [1, 2, 3])     // [3, 6, 9]
```

The callback you pass has to match how the caller will invoke it. `applyTo` calls your callback with exactly one argument — `Cb.call(x)` — so the function behind the callback must accept exactly one argument. `double(N)` fits perfectly. But if you handed it a two-argument function like `add(A, B)`, it would break: when `applyTo` reaches `Cb.call(x)`, Aussom goes looking for a one-argument version of `add`, doesn't find one, and throws an error (`... has no overload of 'add' matching call signature`, the same overload matching from Chapter 14). Your callback would never run. So before you hand a callback to other code, check how that code will call it — the **number and types of arguments have to line up**, or it won't work.

A Callback Remembers Its Object

Here's a natural question. A callback like `::greet` is a reference to a *method*, and methods live inside a class and use `this` to reach the object's data and its other methods (Chapter 14). When you hand that callback off to some unrelated code, does `this` still work?

Yes. A callback carries its object along with it. However far it travels, when the callback finally runs, its **`this` still points to the object the callback was made from** — so it keeps full access to that object's members (even private ones) and its other methods.

Watch what happens when a callback made from one object is called by a *different* object that knows nothing about it:

```
class Greeter {
    private name = "Ada";
    public greet() {
        return "Hi, " + this.name;      // greet reads this.name
    }
    public makeCallback() {
        return ::greet;                 // a callback bound to this
                                        // Greeter
    }
}

// runIt has no idea which object the callback belongs to.
class Runner {
    public runIt(Cb) {
        return Cb.call();
    }
}

g = new Greeter();
callback = g.makeCallback();
c.log(new Runner().runIt(callback));    // reaches g's own name
```

It prints:

```
Hi, Ada
```

`Runner` calls the callback and knows nothing about `Greeter`, yet `greet` still finds `this.name` on

the original g object. The callback didn't just capture the *function* — it captured the *object* too. That binding is what makes callbacks so useful for things like **event handlers**: you register one object's method to be run later by another part of the program — say, when a button is clicked or a timer fires — and it still runs "as" that object, with all its data intact.

One consequence to keep in mind: because the object rides along, any state the method changes — incrementing a counter, appending to a list — happens on that same original object every time the callback runs.

A Real Use: Custom Sorting

You don't have to invent uses for callbacks — the standard library already takes them. Recall from Chapter 10 that `sortAsc` orders a list ascending. When you want a *different* order, **`sortCustom`** takes a callback that compares two items and decides which comes first. Your comparator returns a negative number if `A` should come first, a positive number if `B` should, and `0` if they're equal:

```
// Compare by length, longest first.
public byLengthDesc(A, B) {
    return #B - #A;
}

// ...
words = ["bb", "a", "cccc", "ddd"];
words.sortCustom(::byLengthDesc);       // ["cccc", "ddd", "bb", "a"]
```

You wrote the *rule*; `sortCustom` handles the actual sorting and calls your rule whenever it needs to compare two items. That division of labor — general machinery plus a plug-in behavior — is exactly what callbacks are for.

Try It Yourself

This `Tools` class passes callbacks to `applyTo` and to `sortCustom`. Save it as `callbacks.auss` and run it:

```
class Tools {
    public double(N) { return N * 2; }
    public triple(N) { return N * 3; }

    // applyTo takes a callback and runs it on every item.
    public applyTo(Cb, Items) {
        result = [];
        for (x : Items) {
            result @= Cb.call(x);      // .call runs the callback
        }
        return result;
    }

    // A comparator for sortCustom: longer strings come first.
```

```
    public byLengthDesc(A, B) {
        return #B - #A;
    }

    public run() {
        nums = [1, 2, 3, 4];
        c.log("doubled:   " + this.applyTo(::double, nums).join(", "));
        c.log("tripled:   " + this.applyTo(::triple, nums).join(", "));

        words = ["bb", "a", "cccc", "ddd"];
        words.sortCustom(::byLengthDesc);
        c.log("by length: " + words.join(", "));
    }
}

new Tools().run();
```

Running it prints:

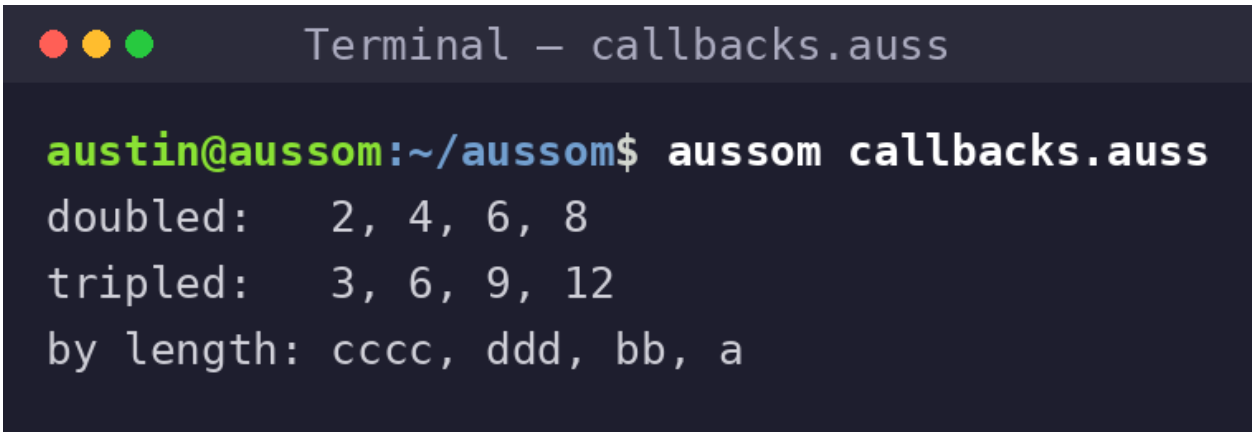


Figure 42: Running callbacks.auss

You can find this example here: `15-callbacks`.

What You Learned

- A **callback** is a function treated as a value — a reference you can store and pass around.
- Create one with **`::name`** (no parentheses — that would call it), and run it with **`.call(args)`**.
- **Passing** a callback lets general code (like `applyTo`, or the library's **`sortCustom`**) run *your* behavior at the right moment, without knowing what it does. The callback's **parameters must match how the caller invokes it** (the same number and types of arguments), or the call throws an error.
- A callback **stays bound to its object**: wherever it's called, its `this` still points to the object it was made from, so it can use that object's members and methods.

That completes Part V. You can now bundle logic into functions and pass behavior around as call-

backs. Next, in **Part VI**, we go deeper into the classes you've been using as containers — starting with **Chapter 16, Classes and Objects**.

Part VI

Object-Oriented Aussom

Chapter 16 - Classes and Objects

You've been using classes since Chapter 14 — as containers to hold your functions. That's a real use, but it's only half the story. A **class** can hold *data* too, right alongside the functions that work on it. Bundling data and behavior together is the heart of **object-oriented programming**, and it's how you build the pieces of a larger program: a `BankAccount`, a `Player`, a `ShoppingCart`.

This chapter covers classes properly: how to define one, how to create **objects** from it, and how objects keep their own data.

Defining a Class

A **class** is a *blueprint*. It describes what every object of that kind will have — its **data** (called *members*) and its **actions** (called *methods*). By itself a class doesn't do anything; it's the plan you stamp copies from.

Here's a class that describes a bank account:

```
class BankAccount {
    public owner = null;         // a member: data every account holds
    private balance = 0;         // another member

    public deposit(Amount) {     // a method: an action every account
                                 // can do
        this.balance = this.balance + Amount;
        return this.balance;
    }
}
```

The **members** (`owner`, `balance`) are variables that live inside each object, each with a starting value. The **methods** (`deposit`) are the functions that act on that data. The class ties them together.

Creating Objects with `new`

A class is just the blueprint — to actually use it, you create an **object** from it (also called an **instance**) with the **`new`** keyword:

```
account = new BankAccount();
```

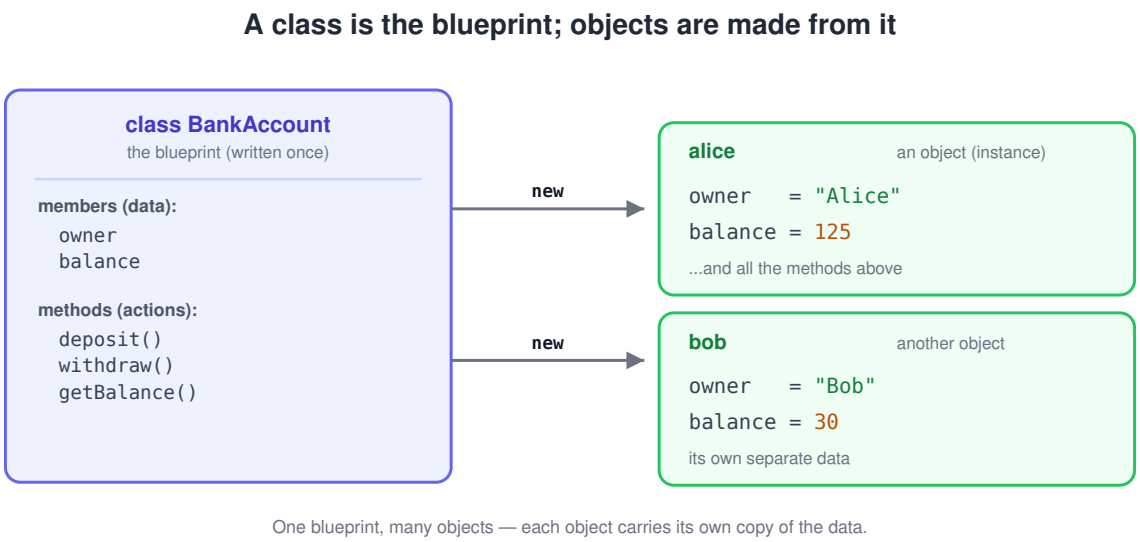


Figure 43: A class is a blueprint; each object made from it has its own data

`account` is now a real object with its own `owner` and `balance`. The key idea: **each object made from a class has its own separate copy of the members.** Create two, and they don't share data:

```
alice = new BankAccount();
bob   = new BankAccount();
```

`alice` and `bob` are two independent accounts. Changing `alice`'s balance leaves `bob`'s untouched.

Constructors

Making an account and *then* filling in its details is clumsy. A **constructor** lets you set up an object at the moment it's created. A constructor is simply a method with the **same name as the class**. Aussom runs it automatically when you say `new`, passing along whatever arguments you give:

```
class BankAccount {
    public owner = null;
    private balance = 0;

    public BankAccount(Name, StartingBalance) {   // the constructor
        this.owner = Name;
        this.balance = StartingBalance;
    }
}

alice = new BankAccount("Alice", 100);   // runs the constructor with
                                         // these args
```

Now `alice` starts life with `owner` set to `"Alice"` and `balance` set to `100`. A constructor is optional

— if you don't write one, `new ClassName()` just makes an object with the members at their starting values, as you saw above.

Members, Methods, and `this`

Inside a method, you reach the object's own data and other methods through **`this`** — the word that means "the object I'm running on" (you met it in Chapter 14). `this.balance` is *this account's* balance; `this.deposit(...)` calls *this account's* deposit method.

```
public withdraw(Amount) {
    if (Amount > this.balance) {          // read this object's balance
        return "Insufficient funds";
    }
    this.balance = this.balance - Amount; // update this object's
                                          // balance
    return this.balance;
}
```

From **outside** the object, you use the object's name instead of `this`. You call its methods, and you can read (and, for public members, set) its data, all with a dot:

```
alice.deposit(25);          // call a method
c.log(alice.owner);         // read a public member
alice.owner = "Alice B.";   // set a public member
```

Public and Private Access

You've seen `public` and `private` on methods since Chapter 14; they work the same way on **members**. The modifier controls who can touch that member or method:

- **`public`** — reachable from outside the object (like `alice.owner`). Use it for the parts you *want* the outside world to use.
- **`private`** — reachable only from *inside* the class, through `this`. Trying to touch it from outside is an error.

This matters. In our `BankAccount`, `balance` is **private** on purpose: outside code shouldn't be able to write `alice.balance = 1000000` and invent money. The only way to change the balance is through `deposit` and `withdraw`, which can enforce the rules (like refusing to overdraw). This idea — hiding data behind methods that guard it — is called **encapsulation**, and it's one of the biggest reasons to use classes.

```
alice.deposit(25);           // OK - the public, controlled way
alice.balance = 1000000;     // ERROR - balance is private
```

Try It Yourself

This `BankAccount` class shows members, a constructor, public and private access, and two independent objects. Save it as `bankaccount.auss` and run it:

```
class BankAccount {
    public owner = null;
    private balance = 0;         // private: only this class touches it
                                 // directly

    public BankAccount(Name, StartingBalance) {
        this.owner = Name;
        this.balance = StartingBalance;
    }

    public deposit(Amount) {
        this.balance = this.balance + Amount;
        return this.balance;
    }
    public withdraw(Amount) {
        if (Amount > this.balance) {
            return "Insufficient funds";
        }
        this.balance = this.balance - Amount;
        return this.balance;
    }
    public getBalance() {
        return this.balance;
    }
}

alice = new BankAccount("Alice", 100);
bob   = new BankAccount("Bob", 50);

c.log(alice.owner + " starts with " + alice.getBalance());
c.log(bob.owner + " starts with " + bob.getBalance());

alice.deposit(25);
bob.withdraw(20);

c.log(alice.owner + " now has " + alice.getBalance());
c.log(bob.owner + " now has " + bob.getBalance());
c.log("Bob tries to overdraw: " + bob.withdraw(1000));
```

Running it prints:

Alice's deposit didn't affect Bob, and Bob couldn't overdraw — the account protected its own balance. You can find this example here: `16-classes`.

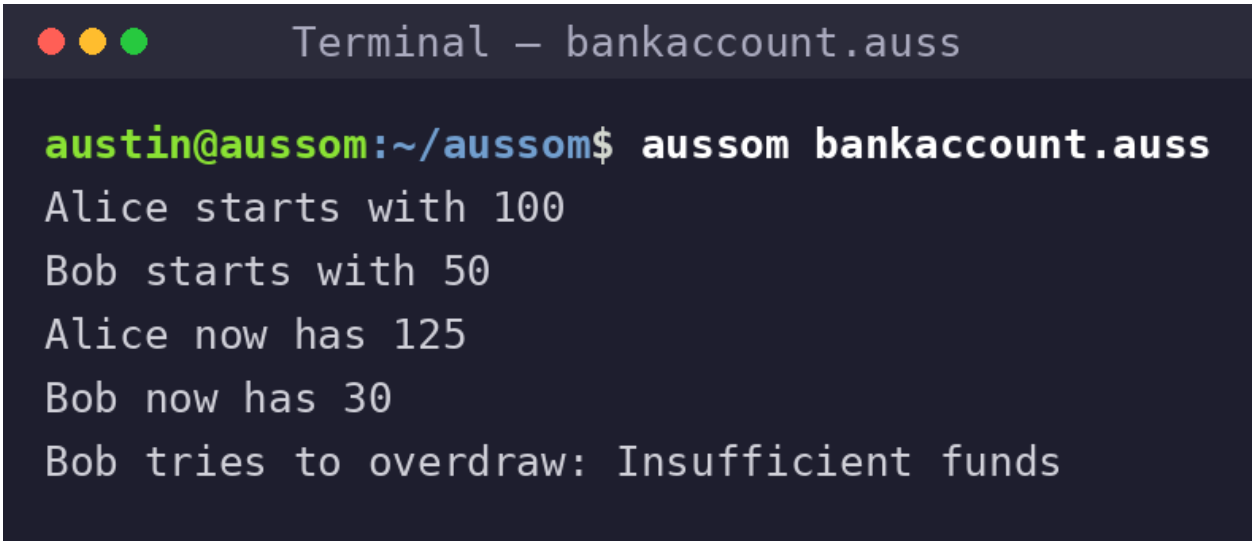


Figure 44: Running bankaccount.auss

What You Learned

- A **class** is a blueprint bundling **members** (data) and **methods** (actions); an **object** is a real instance made from it with **new**.
- Each object has its **own copy** of the members, so objects are independent.
- A **constructor** — a method named like the class — sets up an object when it's created.
- Inside methods, **`this`** refers to the current object; from outside, use the object's name.
- **`public`** members and methods are reachable from outside; **`private`** ones are hidden, letting a class guard its data (**encapsulation**).

Next, in **Chapter 17**, we let one class build on another with **inheritance**.

Chapter 17 - Inheritance and Polymorphism

Often several kinds of thing share most of their behavior but differ in a few ways. A dog, a cat, and a bird are all animals — they all have a name and can eat and sleep — but each makes its own sound. Copying the shared parts into every class would be wasteful and error-prone. **Inheritance** solves this: you write the common parts once in a **parent** class, and each **child** class builds on it, adding or changing only what's different.

Extending a Class

To make one class build on another, put a **colon** and the parent's name after the child's name. The child then **inherits** everything the parent has — all its members and methods — as if they were written in the child:

```
class Animal {
    public name = null;
    public Animal(Nm) { this.name = Nm; }
    public speak() { return "some sound"; }
    public describe() { return this.name + " says " + this.speak(); }
}

class Dog : Animal {                      // Dog is an Animal
    public Dog(Nm) { this.name = Nm; }
    public fetch() { return this.name + " fetches the ball"; }
}
```

`Dog` didn't define `describe()` or a `name` member, but it has them, because it's an `Animal`. It also adds its own method, `fetch()`. So a `Dog` object can do everything an `Animal` can, plus more:

```
rex = new Dog("Rex");
c.log(rex.describe());      // inherited from Animal
c.log(rex.fetch());         // Dog's own
```

Aussom also lets a class inherit from **more than one** parent at once — we'll come back to that, and to what happens when parents collide, later in the chapter.

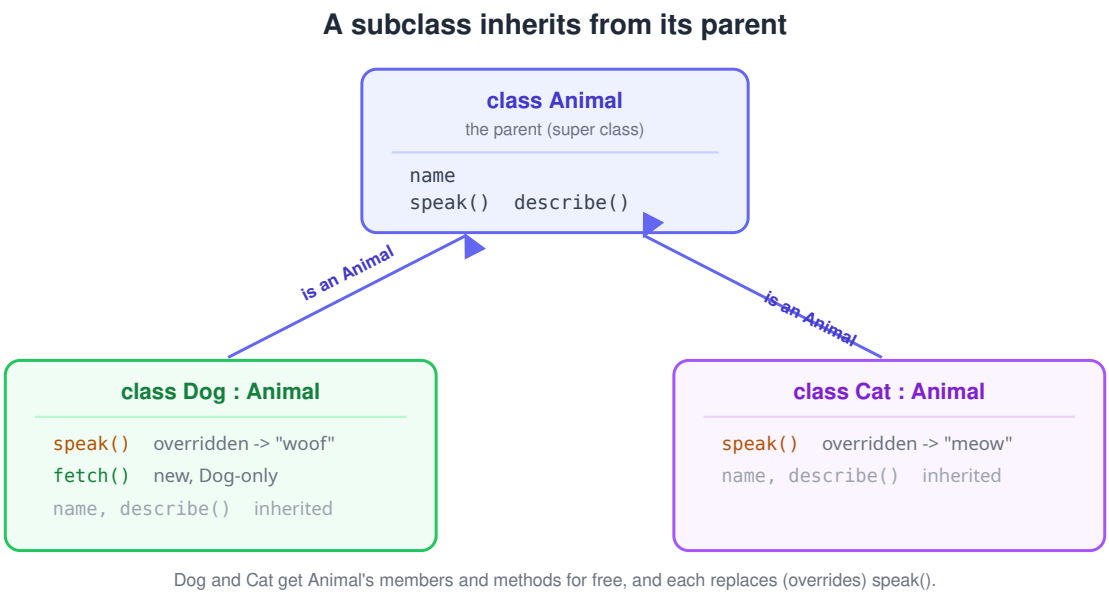


Figure 45: A subclass inherits its parent's members and methods, and can override or add

Overriding Methods

Inheriting the parent's behavior is useful, but a child often needs to do something *differently*. When a child defines a method with the **same name** as one it inherited, the child's version **replaces** the parent's for objects of that child. This is called **overriding**.

Our `Animal.speak()` returns a generic `"some sound"`. A `Dog` should bark, so `Dog` overrides `speak()`:

```
class Dog : Animal {
    public Dog(Nm) { this.name = Nm; }
    public speak() { return "woof"; }     // overrides Animal's speak()
}
```

Now `new Dog("Rex").speak()` returns `"woof"`, not `"some sound"`. A `Cat` would override it to return `"meow"`. Each subclass keeps everything else it inherited and changes only what it needs to.

Members shadow the same way. This isn't only for methods — if a child declares a **member** with the same name as one it inherited, the child's version (and its starting value) wins for objects of that child:

```
class Base { public label = "base"; }
class Sub : Base {
    public label = "sub";     // shadows the parent's label
}
c.log(new Sub().label);       // "sub"
```

In both cases the rule is the same: a child's version of a name — method or member — takes the place of the parent's for that child's objects.

Using the Parent's Methods

A child calls its inherited methods exactly like its own — with **`this`**. In our example, `describe()` is inherited from `Animal`, and inside it calls `this.speak()`. Here's the powerful part: when you call `describe()` on a `Dog`, `this.speak()` runs *the Dog's* `speak()`, because `this` is the Dog. An inherited method automatically uses the child's overrides. (You'll see why that matters in the next section.)

One thing to know: Aussom has **no `super` keyword**. When a child overrides a method, there's no built-in way to call the parent's original version from inside the override — the child's version simply takes over. If you need both behaviors, give the parent's version a distinct method name (say, `baseSpeak()`) that the child can still call through `this`.

Inheriting from More Than One Class

A class can inherit from **several parents at once** — just list them after the colon, separated by commas. The child then gets the members and methods of *all* its parents. A duck, for instance, both walks and swims:

```
class Walker {
    public habitat = "land";
    public move() { return "walking"; }
}
class Swimmer {
    public habitat = "water";
    public move() { return "swimming"; }
}
class Duck : Walker, Swimmer {
    public quack() { return "quack"; }
}
```

A `Duck` has `move()` and `habitat` (from its parents) *and* `quack()` (its own).

But notice a problem: **both** parents define `move()` and `habitat`. When two parents provide the same name, which one does the child get? The rule is simple: **the first parent you list wins.** `Duck` lists `Walker` before `Swimmer`, so it takes `Walker`'s versions:

```
d = new Duck();
c.log(d.move());       // "walking" - from Walker, the first-listed
                       // parent
c.log(d.habitat);      // "land"    - from Walker
c.log(d.quack());      // "quack"   - Duck's own
```

If you swapped the order to `class Duck : Swimmer, Walker`, you'd get `"swimming"` and `"water"` instead. The same first-listed-wins rule covers both methods and members. And as always, anything the **child** defines itself beats every parent — the child's own version wins over all of them.

So the full pecking order for a name is: the **child's own** definition first, then its parents **in the order you listed them**.

Polymorphism in Practice

Polymorphism — a big word for a simple, powerful idea — means you can treat different objects the same way and each responds in its own manner. Because a `Dog` and a `Cat` are both `Animals` with a `speak()` method, you can put them in one list and call the *same* method on each, and each does its own thing:

```
animals = [new Dog("Rex"), new Cat("Felix"), new Animal("Thing")];
for (a : animals) {
    c.log(a.describe());
}
```

That prints:

```
Rex says woof
Felix says meow
Thing says some sound
```

One loop, three different results — the code calling `describe()` doesn't need to know or care which kind of animal it has. That's polymorphism, and it's what makes programs easy to extend: add a `Bird` class later, drop one in the list, and the same loop just works.

To *check* an object's kind when you need to, use **`instanceof`** with the class name as a string. A child counts as an instance of its parent, too:

```
rex instanceof "Dog"       // true
rex instanceof "Animal"    // true  — a Dog is an Animal
rex instanceof "Cat"       // false
```

Try It Yourself

This program shows extending, overriding, `instanceof`, and polymorphism. Save it as `animals.auss` and run it:

```
class Animal {
    public name = null;
    public Animal(Nm) { this.name = Nm; }
    public speak() { return "some sound"; }
    public describe() { return this.name + " says " + this.speak(); }
}

class Dog : Animal {
    public Dog(Nm) { this.name = Nm; }
    public speak() { return "woof"; }
    public fetch() { return this.name + " fetches the ball"; }
}

class Cat : Animal {
    public Cat(Nm) { this.name = Nm; }
```

```
    public speak() { return "meow"; }
}

rex = new Dog("Rex");
c.log(rex.describe());
c.log(rex.fetch());
c.log("rex is an Animal? " + (rex instanceof "Animal"));

animals = [new Dog("Rex"), new Cat("Felix"), new Animal("Thing")];
for (a : animals) {
    c.log(a.describe());
}
```

Running it prints:

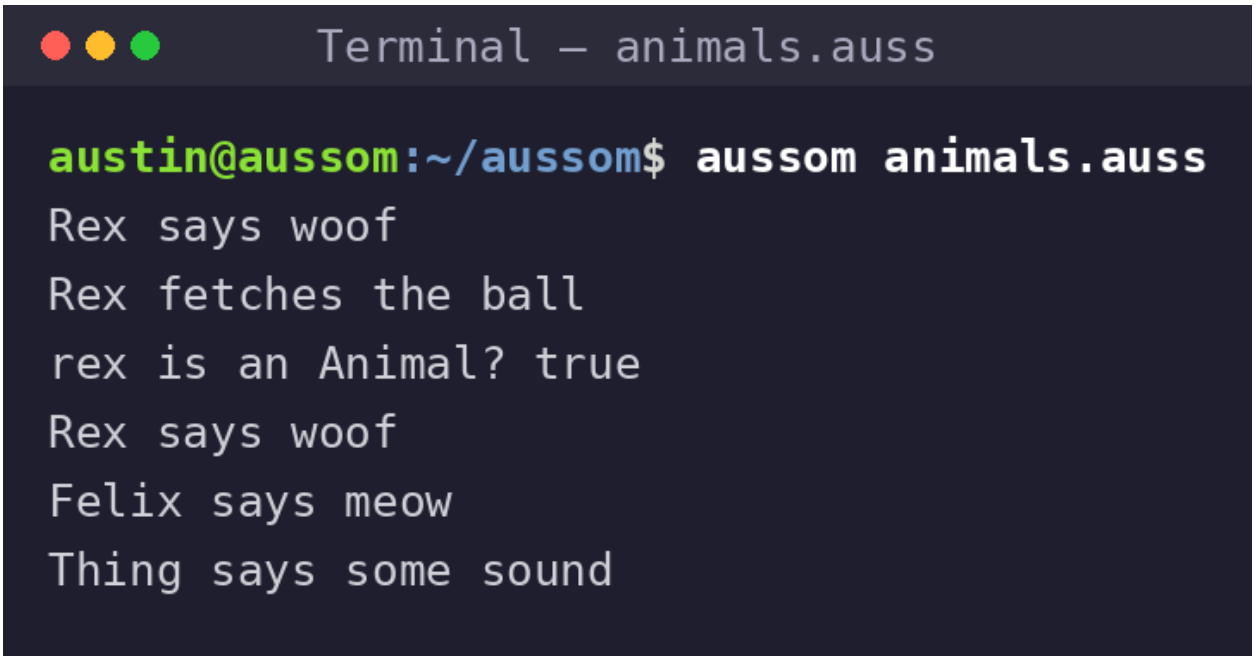


Figure 46: Running animals.auss

You can find this example here: `17-inheritance`.

What You Learned

- **Inheritance** with `class Child : Parent` gives the child all the parent's members and methods. You can inherit from **several parents** with commas; on a name clash between parents, the **first-listed parent wins**.
- **Overriding** — defining a method (or **member**) with the same name as an inherited one — replaces, or *shadows*, the parent's version for that child's objects. The child's own version always beats every parent. Aussom has no `super`, so an override fully takes over.

- Inherited methods use the child's overrides through **`this`**, which gives you **polymorphism**: treat different objects the same way and each responds its own way.
- **`instanceof "ClassName"`** checks an object's kind; a child is also an instance of its parent.

Next, in **Chapter 18**, we look at two special kinds of class — **static** and **extern**.

Chapter 18 - Static and Extern Classes

The classes in the last two chapters were blueprints: you make objects from them with new, and each object carries its own data. Aussom has two more kinds of class that work differently — **static** classes and **extern** classes. You've actually been *using* both since Chapter 4, perhaps without realizing it. This chapter explains what they are and when to reach for them.

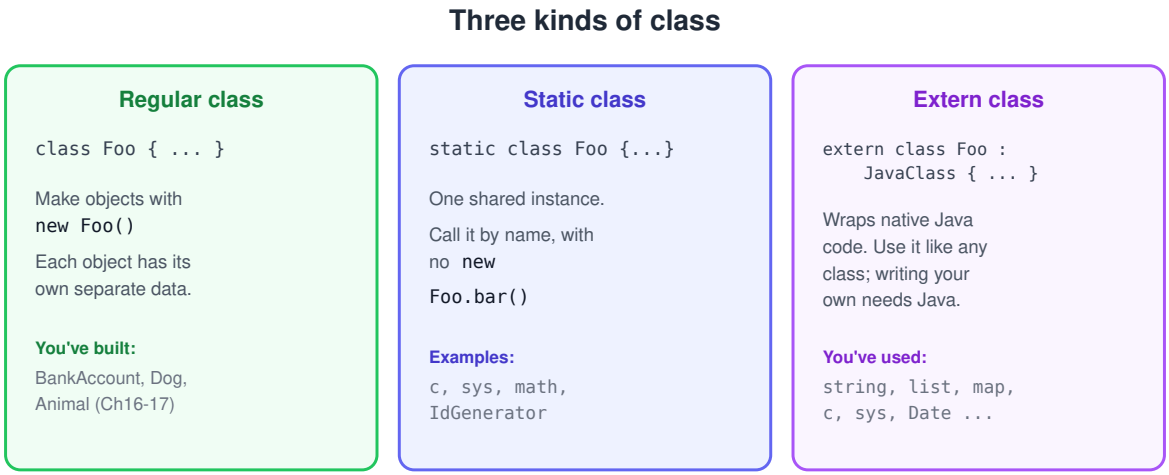


Figure 47: Regular, static, and extern classes compared

Static Classes and Static Members

Sometimes you don't want many objects — you want exactly **one**. A **static class** is a class that exists as a single, shared instance. You don't create it with new; you just call it **by its name**. Mark a class `static` to make it one:

```
static class mathHelper {
    public square(N) { return N * N; }
}

c.log(mathHelper.square(5));    // 25 - call by name, no `new`
```

You've been doing this all along. The console **c**, and the modules **sys** and **math**, are all static classes — that's why you write `c.log(...)` and `sys.getOsName()` directly, never `new c()`.

Because a static class is one shared thing, its **members hold shared state** that sticks around between calls. That makes it handy for something there should only be one of — a counter, a configuration, an ID generator:

```
static class idGenerator {
    private nextId = 1;
    public next() {
        id = this.nextId;
        this.nextId = this.nextId + 1;
        return id;
    }
}

c.log(idGenerator.next());   // 1
c.log(idGenerator.next());   // 2  - the shared nextId kept counting
```

> **When does a static class start?** Aussom builds a static class's single instance **up front, as your program loads** — before your first line of code runs, not the first time you use it. That's why its shared state (like `nextId` above) is already sitting at its starting values, ready to go, the moment you first call it.

Static classes are good for two jobs: **grouping related utility functions** under one name (like `math-Helper`), and **holding a single shared thing** (like `idGenerator`). If instead you need many independent copies, use a regular class with `new`.

> **You can't extend a static class.** Inheritance is about stamping many objects from a blueprint — but a static class isn't a blueprint; it's a single shared instance you call by name. So extending one (`class Savings : idGenerator`) doesn't make sense, and Aussom reports an error if you try. When you want to share behavior through inheritance, use a **regular** class as the parent.

`extern` Classes: Bridging to Java (A High-Level Look)

Aussom runs on the Java platform (Chapter 1), and a lot of its power comes from being able to use Java code. An **extern class** is how that connection is made: it *wraps* a Java class so you can use it from Aussom as if it were an ordinary class.

You don't need to write extern classes to use Aussom — but it helps to know they're there, because **much of what you've already used is built this way**. The `string`, `list`, and `map` types, the `c` console, the `Date` type, `sys`, `math` — all of them are extern classes wrapping Java behind the scenes. When you call `"hi".toUpper()` or `sys.getOsName()`, an extern class is quietly handing the work to Java and passing the result back.

An extern class definition names the Java class it wraps and lists its methods with the **`extern`** keyword. The declarations give only the name and arguments — the actual behavior lives in the Java code:

```
static extern class sys : com.aussom.stdlib.ASys {
    public extern getOsName();
    public extern getOsArch();
```

```
    // ...
}
```

From your side, you call these like any other method:

```
include sys;
c.log("Running on: " + sys.getOsName() + " (" + sys.getOsArch() + ")");
```

> **You can inherit from only one extern class.** An extern class wraps a single Java class, and Java lets a class have just one parent — so a class can't inherit from two extern classes at once. Aussom reports an error if you try. Mixing a *single* extern parent with ordinary Aussom parents is fine; it's only *two extern* parents that aren't allowed. This rarely comes up in a beginner program, but it's a good boundary to know.

Writing your own extern class means writing Java to back it, which is beyond this beginner book — see the advanced documentation at aussom-lang.com if you ever need to bridge to a Java library yourself. For a *lighter* way to call Java — no wrapper class, no compile step — Aussom also has **AJI**, which invokes Java directly at runtime; **Appendix F** covers it briefly, along with **Panama** for calling native C libraries. For now, the takeaway is simply: extern classes are the bridge between Aussom and Java, and you already rely on them every time you use the standard library.

When to Use Each

Three kinds of class, three jobs:

- **Regular class** (`class Foo`) — when you need **many objects**, each with its own data. Most of your own classes will be this kind: `BankAccount`, `Dog`, `Player`.
- **Static class** (`static class Foo`) — when you need **one shared thing**: a set of utility functions, or a single object that holds shared state.
- **Extern class** (`extern class Foo : JavaClass`) — the **bridge to Java**. You use these constantly through the standard library; you'd only write one to wrap a Java library, which is an advanced topic.

Try It Yourself

This program uses a static class with shared state, a static utility class, and a call into the extern `sys` class. Save it as `staticextern.auss` and run it:

```
include sys;

// A static class: ONE shared instance, called by name (no `new`).
static class idGenerator {
    private nextId = 1;
    public next() {
        id = this.nextId;
        this.nextId = this.nextId + 1;
        return id;
    }
```

```
}

c.log("id: " + idGenerator.next());
c.log("id: " + idGenerator.next());
c.log("id: " + idGenerator.next());

// A static utility class: a home for related functions, no state needed.
static class tempConverter {
    public toFahrenheit(Celsius) { return Celsius * 9 / 5 + 32; }
}
c.log("100C = " + tempConverter.toFahrenheit(100) + "F");

// sys is an EXTERN class from the standard library - it wraps Java code.
c.log("Running on: " + sys.getOsName() + " (" + sys.getOsArch() + ")");
```

Running it prints (your system details will differ on the last line):

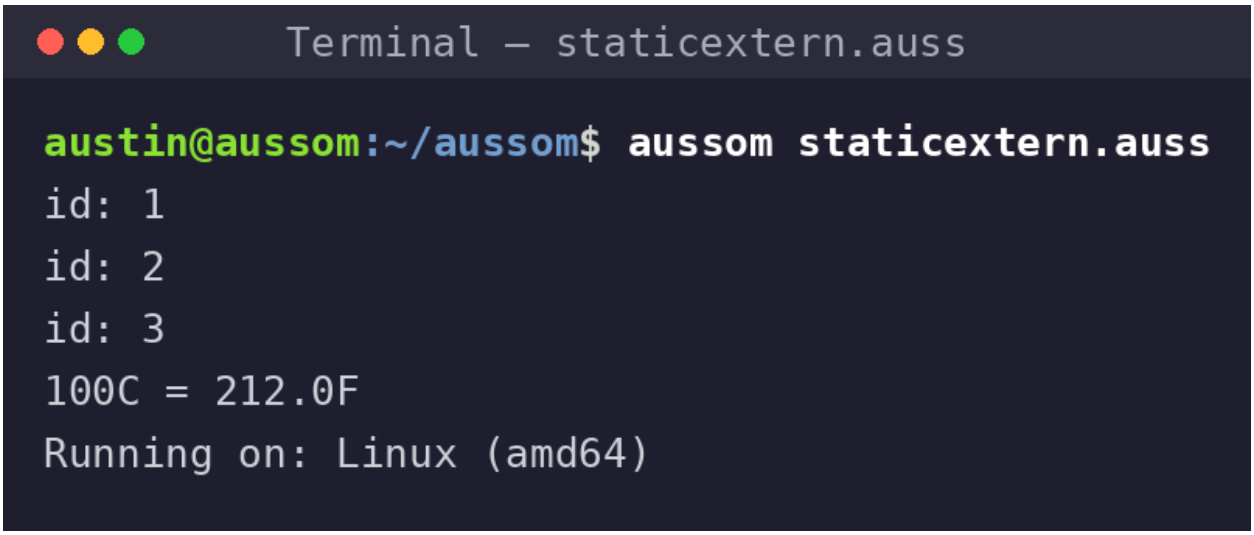


Figure 48: Running staticextern.auss

You can find this example here: `18-static-extern`.

What You Learned

- A **static class** (`static class Foo`) is a single shared instance you call by name — no new. Its members hold shared state. `c`, `sys`, and `math` are static classes. Use one for utility functions or a single shared object.
- An **extern class** wraps Java code; most of the standard library (`string`, `list`, `sys`, `Date`, ...) is built this way. You use them all the time; writing your own is an advanced, Java-side topic.
- Choose a **regular** class for many independent objects, a **static** class for one shared thing, and rely on **extern** classes through the standard library.

That completes **Part VI**. You can now build your own types with classes, share behavior with inher-

itance, and recognize the special static and extern classes the language is built from. Next, in **Part VII**, we make programs sturdier — starting with **Chapter 19, Handling Errors**.

Part VII

Writing Robust Programs

Chapter 19 - Handling Errors

No matter how carefully you write a program, things go wrong. A file isn't where you expected it, a calculation divides by zero, a list index runs off the end. When something like that happens, Aussom raises an **error** — and unless you do something about it, that error stops your program cold. This chapter shows how to *catch* errors so your program can respond to them and keep going, and how to *raise* your own errors when your code hits a situation it can't handle.

What Exceptions Are

When Aussom hits a problem it can't continue past, it creates an **exception** — an object that describes what went wrong — and *throws* it. A thrown exception travels straight up and out of your program, and if nothing catches it, the program crashes with an error message.

Here's a program that divides by zero:

```
c.log("Starting...");
x = 10 / 0;
c.log("This line never runs.");
```

Instead of printing all three lines, it stops partway through with an error:

```
Starting...
[error] astExpression.evalDiv(): Division by 0 exception.
```

The second line threw an exception, so the third line never ran. An **exception**, then, is Aussom's way of signaling "I can't do this" — and by default it ends the program. The rest of this chapter is about taking control of that moment instead of letting it crash you.

`try` and `catch`

To handle an error instead of crashing, wrap the risky code in a **`try`** block and follow it with a **`catch`** block. Aussom runs the `try` block normally; if an exception is thrown anywhere inside it, Aussom stops that block and jumps to `catch`, handing it the exception. If *no* error happens, the `catch` block is skipped entirely.

```
try {
    x = 10 / 0;                 // this throws
    c.log("never gets here");
} catch (e) {
```

```
    c.log("Something went wrong, but we handled it.");
}
c.log("The program keeps running.");
```

This time the program doesn't crash. The division throws, control jumps to `catch`, the message prints, and execution continues past the whole `try/catch` — so the last line runs too. That's the whole point: an error becomes something you *respond* to, not something that ends the program.

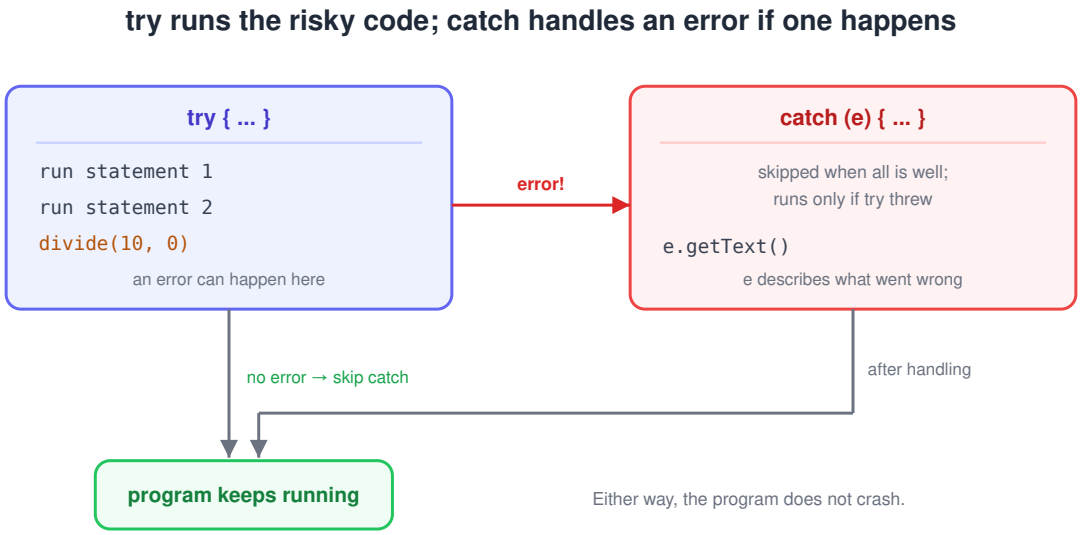


Figure 49: try runs the risky code; catch handles an error if one happens

The name in the parentheses after `catch` — here `e` — is a variable that holds the exception object, so your handler can inspect what happened. We'll look at that object next.

> **There is no `finally`.** Some languages add a third block, `finally`, that runs whether or not an error occurred. Aussom has only `try` and `catch` — put any "always run afterward" code on the line *after* the `try/catch`, which is reached in both cases.

The `exception` Object

The variable your `catch` block receives (e above) is an **exception object**. Its most useful method is **`getText()`**, which returns the error message as a string:

```
try {
    x = 10 / 0;
} catch (e) {
    c.log("Error: " + e.getText());
}
```

That prints `Error: astExpression.evalDiv(): Division by 0 exception.` The exception object offers a few methods for looking at the problem:

- **`getText()`** — the error message. This is the one you'll use most.
- **`getDetails()`** — a longer description; for many errors it's the same as the text.
- **`getLineNumber()`** — the line where the error was thrown.
- **`getStackTrace()`** — the trail of function calls that led to the error, as a string. We'll read one in the next section.
- **`getExceptionType()`** and **`getId()`** — a category name and identifier the language uses to classify errors.

You can also confirm you're holding an exception with `lang.type(e)`, which returns the string `"exception"`.

Notice that the message for a *language* error like division by zero includes some internal detail (`astExpression.evalDiv():`). That's Aussom telling you where inside itself the trouble surfaced. When you throw your *own* errors — coming up next — the message reads exactly as you wrote it, with no such prefix.

Reading the Stack Trace

An error rarely happens at the top of your program. Usually one function calls another, which calls another, and the trouble surfaces deep down in that chain. A **stack trace** is the record of exactly that chain — the trail of function calls that were in progress when the error was thrown. It's the single most useful tool for finding *where* a problem really came from.

You get the trace two ways. **`getStackTrace()`** returns it as a string, and **`printStackTrace()`** prints it for you (handy for a quick look while debugging). Here are three methods that call one another, saved as `stacktrace.auss`, with the deepest one dividing by zero. Counting lines from the top of the file (the comment is line 1), the numbers in the trace below will line up:

```
// Chapter 19 - Reading a stack trace

class Bank {
    public divide(A, B) {
        return A / B;                  // divide by zero throws HERE
    }
    public applyRate(Amount) {
        return this.divide(Amount, 0);
    }
    public process(Amount) {
        return this.applyRate(Amount);
    }
}

bank = new Bank();

try {
    bank.process(100);
} catch (e) {
    c.log("Error: " + e.getText());
    c.log("Here is where it happened:");
```

```
    e.printStackTrace();
}
```

`process` calls `applyRate`, which calls `divide`, which throws. Printing the trace shows the whole path:

```
austin@aussom:~/aussom$ aussom stacktrace.auss
Error: astExpression.evalDiv(): Division by 0 exception.
Here is where it happened:
  [stacktrace.auss:4] Defined. { Bank.divide() }
  [stacktrace.auss:8] Function called. { Bank.divide() }
  [stacktrace.auss:7] Defined. { Bank.applyRate() }
  [stacktrace.auss:11] Function called. { Bank.applyRate() }
  [stacktrace.auss:10] Defined. { Bank.process() }
  [stacktrace.auss:18] Function called. { Bank.process() }
```

Figure 50: Printing a stack trace with printStackTrace()

How to read it. Each line starts with `[file:line]` — the file and line number it points to — followed by the method it's about in `{ curly braces }`. The lines come in pairs, one per function in the chain:

- **`Defined.`** — where that method's code lives.
- **`Function called.`** — the line that *called* it.

Read the trace **top to bottom, from the inside out.** The **top** pair is where the error actually happened — `Bank.divide()`, defined at line 4 (our `a / b`), called from line 8. The next pair down is the method that called *it* — `applyRate()`, called from line 11 — and the last pair is `process()`, called from line 18, which is our `try` block where the whole chain began.

So the trace reads like a trail of breadcrumbs back to the start: **line 18** called `process`, which at **line 11** called `applyRate`, which at **line 8** called `divide`, which broke at **line 4**. When you're hunting a bug, start at the top — that top line is almost always where you need to look first.

Raising Your Own Errors

Errors aren't only for the language to raise — your own code can raise them too, with the **`throw`** keyword. You `throw` a string describing what went wrong, and it behaves just like any other exception: it stops the current work and travels up until a `catch` handles it (or crashes the program if none does).

This is how a method reports a problem it can't sensibly solve on its own. A `divide` method, for example, can refuse to divide by zero:

```
class Calculator {
    public divide(A, B) {
        if (B == 0) {
            throw "Cannot divide by zero!";   // raise our own error
        }
        return A / B;
    }
}
```

The key idea is that `throw` and `catch` work *across* method calls. The `throw` happens deep inside `divide`, but the `catch` can sit way out where the method was called. The exception finds its way from one to the other:

```
calc = new Calculator();
try {
    result = calc.divide(10, 0);      // throws inside divide()
} catch (e) {
    c.log("Error: " + e.getText());   // caught out here
}
```

That prints `Error: Cannot divide by zero!` — exactly the text we threw. This lets a method say "something is wrong" without deciding what to *do* about it; the code that called the method makes that decision in its `catch`. Throwing is how you keep your own code honest: instead of returning a nonsense value or quietly doing the wrong thing, you raise a clear error the caller can handle.

Try It Yourself

This program raises its own error, catches it, catches a language error too, and keeps running the whole time. Save it as `errors.auss` and run it:

```
class Calculator {
    public divide(A, B) {
        if (B == 0) {
            throw "Cannot divide by zero!";    // raise our own error
        }
        return A / B;
    }
}

calc = new Calculator();

// A normal call works.
c.log("10 / 2 = " + calc.divide(10, 2));

// A bad call throws - we catch it instead of crashing.
```

```
try {
    c.log("10 / 0 = " + calc.divide(10, 0));
} catch (e) {
    c.log("Caught an error: " + e.getText());
}

// Errors from the language itself are catchable too.
try {
    nums = [1, 2, 3];
    c.log("Item 99: " + nums[99]);
} catch (e) {
    c.log("Caught a language error: " + e.getText());
}

c.log("The program finished normally.");
```

Running it prints:

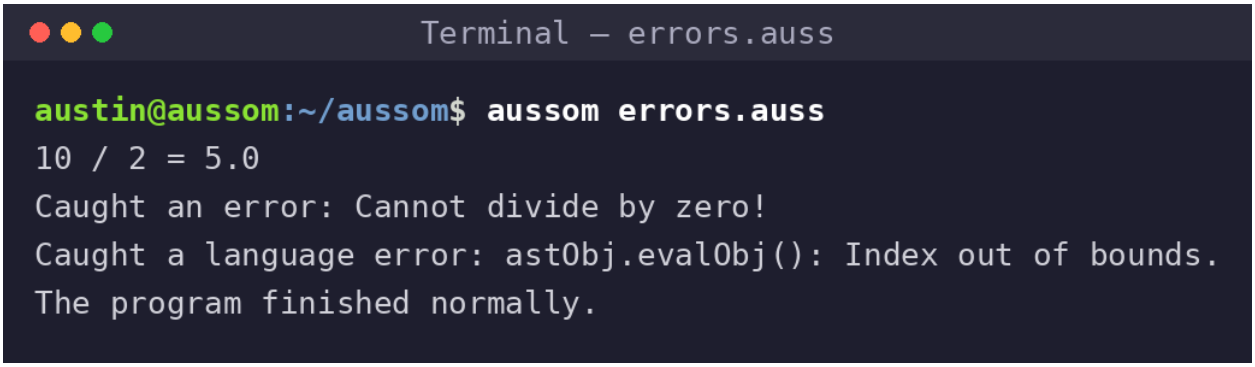


Figure 51: Running errors.auss

The first `divide` worked. The second threw *our* message, which we caught and printed word for word. Reading past the end of the list threw a *language* error, which we caught too — its message carries the internal detail we mentioned. And because every error was handled, the last line still ran. You can find this example here: `19-errors`.

What You Learned

- An **exception** is an object Aussom throws when something goes wrong; left alone, it crashes the program.
- **`try { ... } catch (e) { ... }`** runs risky code and, if it throws, jumps to the `catch` with the exception in `e` — so the program handles the error and keeps going. There is **no `finally`**; put always-run code after the block.
- The exception object's **`getText()`** gives the message; `getDetails()`, `getLineNumber()`, and a couple of others describe it further, and `lang.type(e)` is `"exception"`.
- A **stack trace** — from **`getStackTrace()`** (as a string) or **`printStackTrace()`** (printed for you) — is the trail of function calls that led to the error. Read it **top to bottom, inside out**: the top

line is where the error happened, and each pair below shows the call that led there.
- **`throw "message"`** raises your own error. It travels up through method calls until a `catch` handles it, letting a method report a problem for its caller to deal with.

Next, in **Chapter 20**, we split a growing program across several files with **modules**.

Chapter 20 - Organizing Code with Modules

So far every program you've written has lived in a single file. That's fine for a short script, but as a program grows, one giant file becomes hard to read and harder to reuse. The fix is to split your code into **modules** — separate files, each holding a related piece of the program — and pull them together with the **`include`** statement. This chapter shows how to break a program into modules, combine them, and generate documentation from them.

Splitting Code Across Files

A **module** is just an Aussom file that holds code — usually one or more classes — meant to be used by other files. You've already met the idea: `sys`, `math`, and the other standard-library pieces are modules you `include` and use. Now you'll write your own.

The convention is to give each module a clear job and its own file. A greeting helper goes in one file, math helpers in another:

```
// greetings.aus - a module holding one class
class Greeter {
    public greet(Name) {
        return "Hello, " + Name + "!";
    }
}
```

Module files use the **`.aus`** extension (the *classical* form from Chapter 3), because a module is a collection of class definitions rather than a top-to-bottom script. Your main program can still be a **`.auss`** script — it just pulls the modules in, as you'll see next.

The `include` Statement and Module Lookup

To use a module, name it in an **`include`** statement at the top of your file:

```
include greetings;
```

`include greetings;` tells Aussom to find **`greetings.aus`** — it takes the module name and adds `.aus` — and load everything in it. After that line, the `Greeter` class is available in your program as if you'd written it right there:

```
include greetings;

g = new Greeter();
c.log(g.greet("Ada"));      // Hello, Ada!
```

Notice you write `include greetings;`, **not** `include greetings.aus;` — you give the module *name*, and Aussom adds the extension.

Modules in subfolders. As you gather more modules, you'll want to tuck some into folders. A **dot** in the module name means a subfolder. If `mathutil.aus` lives in a folder named `lib`, you reach it like this:

```
include lib.mathutil;       // loads lib/mathutil.aus
```

Each dot steps down one folder, so `include a.b.thing;` loads `a/b/thing.aus`.

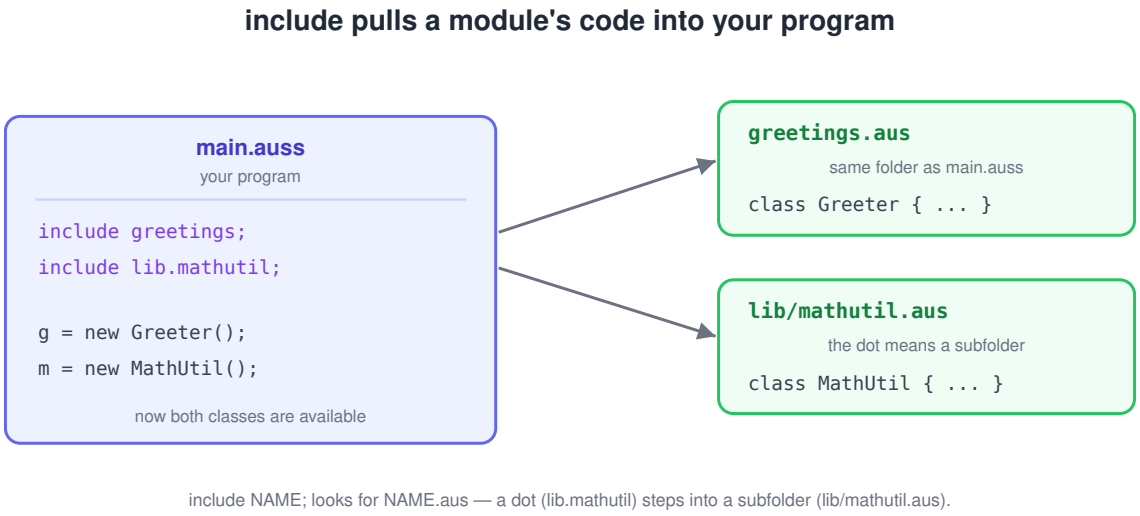


Figure 52: include pulls a module's code into your program

Where Aussom Looks for a Module, and in What Order

When you write `include name;`, Aussom searches a fixed list of locations and loads the **first** matching file it finds. Knowing this order tells you exactly where to put a module — and which one wins when two share a name. From first checked to last:

1. **Aussom's built-in standard library** — the modules baked into the interpreter, like `sys`, `math`, `file`, and `json`. This is why `include sys;` works with none of your own files around: `sys` ships *inside* Aussom.
2. **`.aussom/modules/`** — packages installed into *this project* with **APAC**, Aussom's package manager. A package here applies only to the project folder it sits in.
3. **The system-global package folder** — packages APAC installed for the whole machine, so every project can use them. The location depends on your operating system:

- **Linux:** `/var/lib/aussom/modules/`
- **macOS:** `/Library/Application Support/Aussom/modules/`
- **Windows:** `%PROGRAMDATA%\Aussom\modules\` (usually `C:\ProgramData\Aussom\modules\`)

4. **`modules/`** — a plain `modules` folder in your project, if you keep one.
5. **The current folder** (`.`) — the directory you ran `aussom` from. This is where your own module files (like `greetings.aus`) live, so your everyday modules are found here.

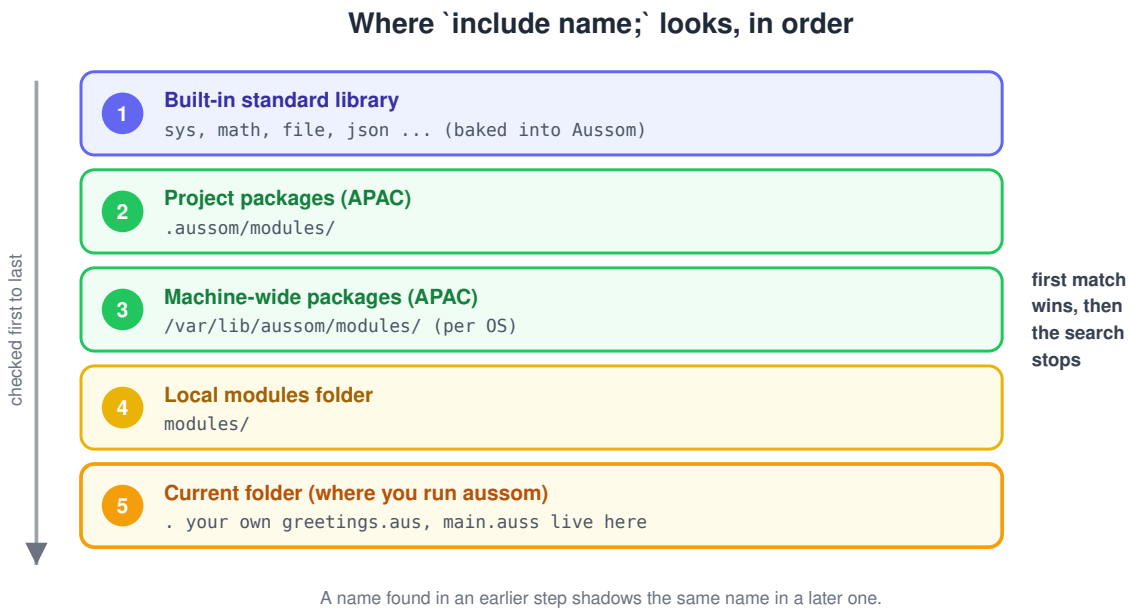


Figure 53: Aussom searches these locations in order and stops at the first match

The search **stops at the first match**, so a module in an earlier location shadows a same-named one further down — a project's own `.aussom/modules/` copy, for example, wins over a machine-wide one. Don't worry about the APAC locations (2 and 3) yet; **Chapter 35** covers the package manager in full. For now, the one to remember is step 5: your own modules sit in the folder you run from.

`include` can go anywhere — even inside your code. We put includes at the top out of habit, so every class is ready before the program does anything, and that's the right default. But `include` is really just a normal statement that runs when Aussom reaches it — so you can place one deeper in your code, even inside an `if`, to load a module only when you actually need it:

```
if (needsCharts) {
    include lib.charts;         // loaded only when this branch runs
    chart = new Chart();
    chart.draw();
}
```

If that branch never runs, the module is never loaded. Once an `include` *does* run, its classes stay available for the rest of the program — a module loaded inside a block still works after the block, and including the same module twice does no harm. Loading everything up front is simplest and is what you'll usually want; reach for a conditional include when a module is optional or costly and you'd rather load it only in the cases that need it.

Building a Multi-File Program

Let's put the pieces together into a small program made of three files. First, the two modules — one in the main folder, one in a `lib` subfolder:

```
// greetings.aus
class Greeter {
    public greet(Name) {
        return "Hello, " + Name + "!";
    }
}
```

```
// lib/mathutil.aus
class MathUtil {
    public square(N) { return N * N; }
}
```

Then the main program that includes both and uses their classes:

```
// main.auss
include greetings;          // greetings.aus, same folder
include lib.mathutil;       // lib/mathutil.aus, a subfolder

g = new Greeter();
c.log(g.greet("Ada"));

m = new MathUtil();
c.log("square(7) = " + m.square(7));
```

You run only the main program — `aussom main.auss` — and Aussom loads the two modules for it. The folder layout looks like this:

```
main.auss
greetings.aus
lib/
    mathutil.aus
```

Each class now lives in its own file, focused on one job, and `main.auss` stays short and readable. As the program grows, you add more modules instead of piling everything into one file.

Documenting Your Code with `aussomdoc`

A module is more useful when others (including future you) know what its classes do. Aussom can build documentation for you from special **doc comments** in your code.

A doc comment starts with `/**` and describes the **class, method, or member** right below it. A class or member comment is just a plain description; a method's comment can add two tags: **`@p`** documents a parameter, and **`@r`** describes the return value. Here's `greetings.aus` with a comment on the class, on a **member** (a stored field — Chapter 16), and on the method:

```
/**
 * A small greetings module.
 */
class Greeter {
    /**
     * The word each greeting starts with.
     */
    public greeting = "Hello";

    /**
     * Builds a friendly greeting.
     * @p Name is the name to greet.
     * @r string - the finished greeting.
     */
    public greet(Name) {
        return this.greeting + ", " + Name + "!";
    }
}
```

Run Aussom with the **-d** flag on the module to generate its documentation:

```
aussom -d greetings.aus
```

That writes a Markdown file, `greetings.aus.md`, describing the module — its class, the class's members, its methods, each method's parameters, and what it returns:

```
# file: greetings.aus

## class: Greeter

[4:7] **extends: object**

A small greetings module.

#### Members
- **greeting**

    > The word each greeting starts with.


#### Methods

- **greet** (`Name`)

    > Builds a friendly greeting.

    - **@p** `Name` is the name to greet.
    - **@r** `string` - the finished greeting.
```

Everything in that document came straight from the doc comments — the class summary, the **`greeting` member**, and the method with its parameter and return value. (The `[4:7]` is where the class sits in the file, line 4 and column 7, and **`extends: object`** notes that every class ultimately builds on the built-in `object` type from Chapter 17.) Write the comments once, and `aussomdoc` keeps the documentation in step with the code — no separate file to maintain by hand.

Try It Yourself

Recreate the three-file program from this chapter. Put `greetings.aus` and `main.auss` in a folder, add `mathutil.aus` inside a `lib` subfolder, then run the main program:

```
aussom main.auss
```

It prints:

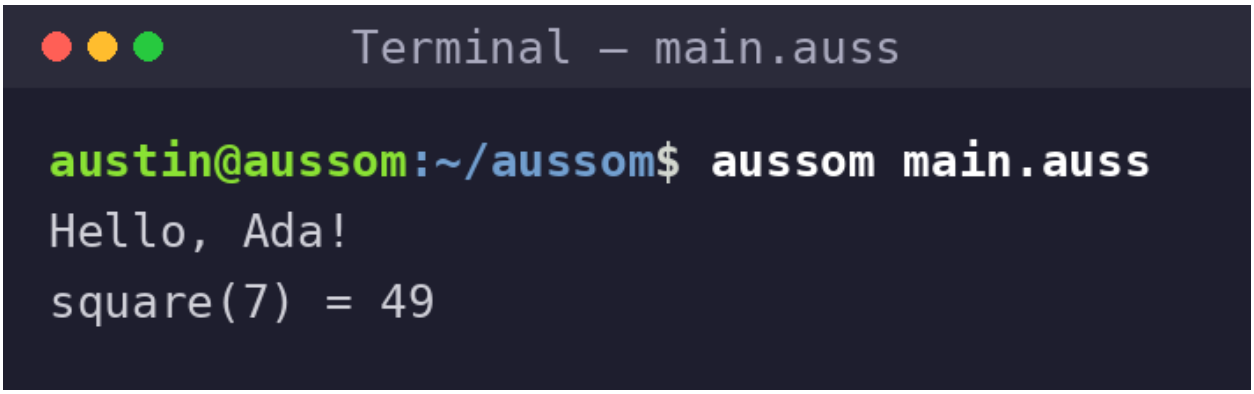


Figure 54: Running main.auss

The `Greeter` came from `greetings.aus` and the `MathUtil` from `lib/mathutil.aus` — two separate modules, combined by `include` into one working program. Then try generating the docs with `aussom -d greetings.aus`. You can find this example here: `20-modules`.

What You Learned

- A **module** is an Aussom file (`.aus`) holding classes meant to be reused; splitting a program into modules keeps each file focused and short.
- **`include name;`** loads `name.aus` and makes its classes available. Use the module *name*, not the filename — Aussom adds `.aus`. A **dot** means a subfolder, so `include lib.mathutil;` loads `lib/mathutil.aus`. `include` is a statement that runs when reached, so you can place one anywhere — even inside an `if`, to load a module only when it's needed.
- Aussom looks for a module in a **fixed order** — built-in standard library, APAC project packages (`.aussom/modules/`), machine-wide APAC packages, a `modules/` folder, then the **current folder** — and loads the **first match**. Your own modules live in the current folder.
- A **`.auss`** script can include `.aus` modules; you run the main program and Aussom loads the modules it needs.
- **Doc comments** (`/** ... */`, with **`@p`** and **`@r`** tags on methods) document **classes, members, and methods**; **`aussom -d file.aus`** turns them into Markdown straight from your code.

That completes **Part VII**. Your programs can now recover from errors and grow across many files. Next, in **Part VIII**, we tour the **standard library** — the built-in tools Aussom gives you — starting with **Chapter 21, The Console and Core Helpers**.

Part VIII

The Standard Library

Chapter 21 - The Console and Core Helpers

You've reached **Part VIII**, where we tour Aussom's **standard library** — the collection of ready-made tools that ship with the language so you don't have to build everything yourself. You've already used one of them constantly: **`c`**, the console. This chapter looks at the console properly, then covers two more everyday helpers — **`lang`**, for asking what kind of value you're holding, and **`json`**, for turning data into text and back.

These three are all **auto-included**: they load automatically, so you can use them without an `include` line. (A few extras — formatted output and reading input — come from the `os` module, which you *do* include; we'll flag it when we get there.)

Printing to the Console

The **console** is your program's connection to the terminal. You've been calling **`c.log(...)`** since Chapter 1 to print a line of text. But the console offers several ways to write output, each meant for a different purpose:

```
c.log("plain output, no prefix");     // your everyday print
c.info("an informational note");      // shows as [info]
c.warn("something to watch");         // shows as [warning]
c.err("something went wrong");        // shows as [error]
```

`c.log` writes the text exactly as-is. The other three add a **level** tag — `[info]`, `[warning]`, or `[error]` — in front of the message. These *logging levels* let you signal how important a message is: a routine note, a caution, or a real problem. A tool that reads your program's output can then filter by level — for example, showing only errors.

There are also two calls for writing **without** a whole line at a time:

```
c.print("Loading");      // no newline - the cursor stays on this line
c.print("...");          // adds to the same line
c.println(" done!");     // writes, then ends the line
```

`c.print` writes text and leaves the cursor right where it stopped, so the next write continues on the same line. **`c.println`** does the same but adds a newline at the end, finishing the line. That's the difference between `log`/`println` (which end the line) and `print` (which doesn't).

Beyond those, the console has two more calls for low-level logging:

- **`c.trc`** — a *trace* message, the most detailed (and lowest) level.
- **`c.dbg`** — a *debug* message, for information useful while developing.

Formatted Output with `printf` and `sprintf`

Gluing a message together with + gets clumsy fast. For anything more than a word or two, it's cleaner to write a **template** and drop values into it. The **`os`** module provides exactly that with **`sprintf`** and **`printf`**. They live in `os`, so bring it in first:

```
include os;
```

`sprintf` builds a formatted **string** and returns it (it prints nothing); **`printf`** does the same but **writes** the result straight to output. In the template, `{}` is a placeholder that gets filled by the next value, in order:

```
line = os.sprintf("Hello, {}! You have {} messages.", "Ada", 3);
c.log(line);                              // Hello, Ada! You have 3
                                          // messages.

os.printf("{} + {} = {}\n", 2, 3, 5);   // writes: 2 + 3 = 5
```

(`printf` adds no trailing newline of its own — the template controls line breaks, which is why the example ends the string with `\n`.) The placeholders come in three forms:

- **`{}`** — filled by the **next** value, left to right.
- **`{N}`** — filled by the value at **position `N`** (0-based), so you can reuse or reorder values: `os.sprintf("{0} and {0} and {1}", "a", "b")` → `"a and a and b"`.
- **`{name}`** — pulled from a **map** argument by key: `os.sprintf("{name} is {age}", {"name": "Ada", "age": 36})` → `"Ada is 36"`.

Write `{{` or `}}` for a literal brace. These same placeholder rules also work on any string through its **`format`** method — `"Hi, {}".format("Ada")` gives `"Hi, Ada"` — since `sprintf` is really just the `os`-module wrapper around it.

Standard Output and Standard Error

The `os` module also has two plain line-writers that control **where** output goes. **`os.out`** writes a line to *standard output* (the normal stream), and **`os.outErr`** writes a line to *standard error* — a separate stream meant for diagnostics and progress notes that shouldn't get mixed into a program's real output:

```
os.out("the result");        // -> standard output
os.outErr("a status note");  // -> standard error
```

This matters when a program's output is captured or piped somewhere: notes and errors sent to `outErr` stay out of the way of the actual data on `out`. Note this differs from `c.err` earlier — `c.err` *logs* a message tagged `[error]`, while `os.outErr` writes plainly to the real standard-error stream, with no prefix.

Reading Input from the Console

Printing is only half a conversation. To read what the user **types**, use **`os.readLine`** — also from the **`os`** module you included above:

```
include os;

name = os.readLine("What is your name? ");   // prints the prompt, waits
                                             // for a line
c.log("Hello, " + name + "!");
```

`os.readLine(prompt)` writes the prompt you give it, waits for the user to type a line and press Enter, and hands back what they typed **as a string** (without the newline). The prompt is optional — `os.readLine()` with no argument just waits for input.

Because input always arrives as a **string**, converting it to a number takes one more step. Every string has **`parseInt()`** and **`parseDouble()`** for exactly this:

```
age = os.readLine("How old are you? ");
years = age.parseInt();                 // "36" (string) -> 36 (int)
c.log("Next year you'll be " + (years + 1));
```

Without the `parseInt()`, `age + 1` would glue a `"1"` onto the text (`"361"`) instead of adding. The related string converters are:

- **`parseInt`** — text to a whole number (`"36"` → 36).
- **`parseDouble`** — text to a decimal number (`"3.14"` → 3.14).
- **`parseBool`** — text to `true` / `false`.

Inspecting Values with `lang`

Sometimes you have a value and need to know **what kind** it is — especially when it came from outside your program (user input, a file, a network reply). The **`lang`** module's **`type`** function tells you, returning the type's name as a string:

```
lang.type(42)          // "int"
lang.type(3.14)        // "double"
lang.type("hi")        // "string"
lang.type(true)        // "bool"
lang.type(null)        // "null"
lang.type([1, 2])      // "list"
lang.type({"a": 1})    // "map"
```

For an **object** you made from a class, `lang.type` returns that **class's name**:

```
class Dog { }
lang.type(new Dog())   // "Dog"
```

This is handy for checking a value before you use it — for instance, confirming user input parsed into an `int` and not something unexpected. (Recall from Chapter 17 that `instanceof "ClassName"` an-

swers a related question — "is this object of this class?" — while `lang.type` simply *names* the type.) The `lang` module also has the advanced `getClassAussomdoc`, which pulls a class's documentation structure at runtime; you won't need it as a beginner.

Working with `json`

JSON (JavaScript Object Notation) is a simple, universal text format for data. It's how programs save structured data to files and send it across the internet. Conveniently, JSON looks almost exactly like an Aussom **map** written out, which makes the two easy to convert between.

Every value has a **`toJson()`** method that turns it into a JSON **string**:

```
person = {"name": "Ada", "age": 36, "langs": ["Aussom", "Java"]};
text = person.toJson();
// text is now: {"name":"Ada","langs":["Aussom","Java"],"age":36}
```

Going the other way, the **`json`** module's **`parse`** function reads a JSON string and rebuilds it as Aussom data — a map or a list you can index normally:

```
back = json.parse(text);
c.log(back["name"]);          // "Ada"
c.log(back["langs"][0]);      // "Aussom"
```

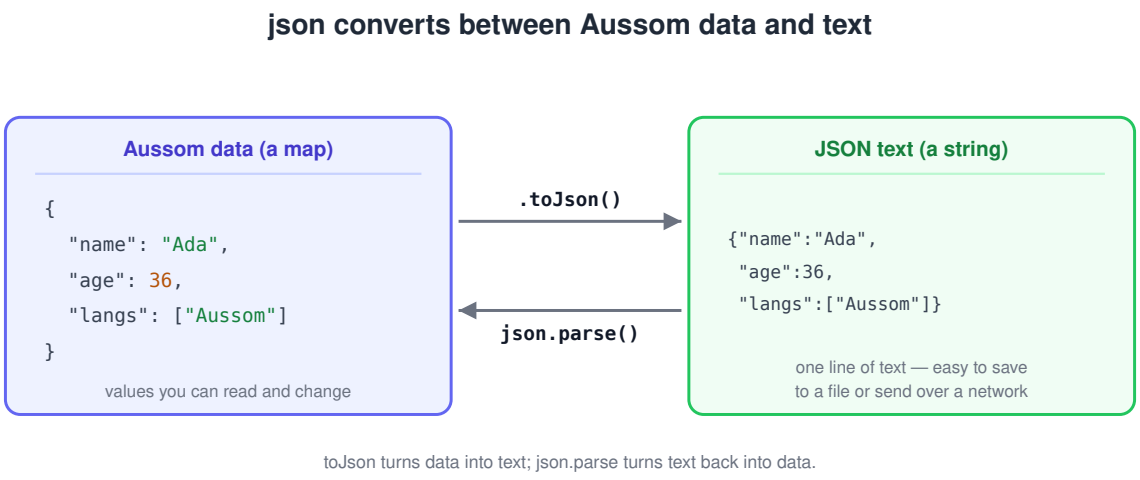


Figure 55: json converts between Aussom data and text

So `toJson` and `parse` are mirror images: one turns data into text you can store or send, the other turns that text back into data you can work with. (Notice the map's keys came out in a different order than they went in — a reminder from Chapter 11 that maps don't keep a set order.) The `json` module also has **`json.escape`**, which safely escapes a string for embedding inside JSON.

Saving Whole Objects with `pack` and `unpack`

`toJson` captures the *data* inside an object but forgets its **type** — `json.parse` always hands back a plain map, with no methods. When you want to save and restore a **whole object**, class and all, use the object's **`pack`** method together with the `json` module's **`unpack`**.

`pack()` turns an object into a string that records both its **data and its class**. **`json.unpack()`** reads that string back and rebuilds a **real, working object** of the original class — methods and all, not just a map:

```
class Player {
    public name = null;
    public score = 0;
    public Player(Nm = null, Sc = 0) { this.name = Nm; this.score = Sc; }
    public describe() { return this.name + " has " + this.score + " points"; }
}

ada = new Player("Ada", 95);
packed = ada.pack();
// packed is:
↪   {"type":"Player","members":{"score":{"type":"int","value":95},"name":{"type":"string"

clone = json.unpack(packed);
c.log(lang.type(clone));       // "Player" - a real object, not a map
c.log(clone.describe());       // "Ada has 95 points" - its methods work
```

Because the packed string is just text, you can put it anywhere text can go. Save it to a file and rebuild the object the next time your program runs, or send it over a network socket so a *different* program can rebuild the same object on the far end:

```
include file;
file.write("player.save", ada.pack());             // marshal to disk...
loaded = json.unpack(file.read("player.save"));     // ...and rehydrate
                                                    // later
```

Turning an object into a string like this is called **marshalling** (and rebuilding it, **unmarshalling**) — the standard way to store objects or move them between programs.

> **One rule for `unpack`.** To rebuild an object, `unpack` creates a fresh instance and *then* fills in its members — so the class must be creatable **with no arguments**. Give it either no constructor at all, or one whose parameters all have defaults (like `Player(nm = null, sc = 0)` above). A constructor that *requires* arguments can't be unpacked.

Try It Yourself

This program shows the console levels, `lang.type`, and a `json` round trip. Save it as `core-helpers.auss` and run it:

```
include os;
```

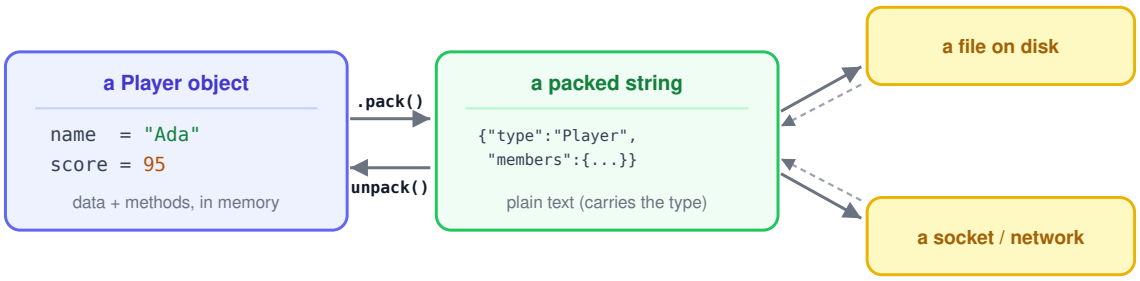


Figure 56: pack turns an object into text; unpack rebuilds the object

```
class Dog { }

c.log("log:  plain output, no prefix");
c.info("info: an informational note");
c.warn("warn: something to watch");
c.err("err:  something went wrong");
c.print("print writes with no newline... ");
c.println("and println finishes the line.");

c.log("---- formatting ----");
c.log(os.sprintf("Hello, {}! You have {} messages.", "Ada", 3));
os.printf("{0} squared is {1}\n", 9, 81);
os.out("os.out writes to standard output");

c.log("---- lang.type ----");
c.log("42     is a " + lang.type(42));
c.log("\"hi\"   is a " + lang.type("hi"));
c.log("[1,2]  is a " + lang.type([1, 2]));
c.log("new Dog() is a " + lang.type(new Dog()));

c.log("---- json ----");
person = {"name": "Ada", "age": 36, "langs": ["Aussom", "Java"]};
text = person.toJson();
c.log("encoded: " + text);
back = json.parse(text);
c.log("name = " + back["name"] + ", first lang = " + back["langs"][0]);
```

Running it prints:

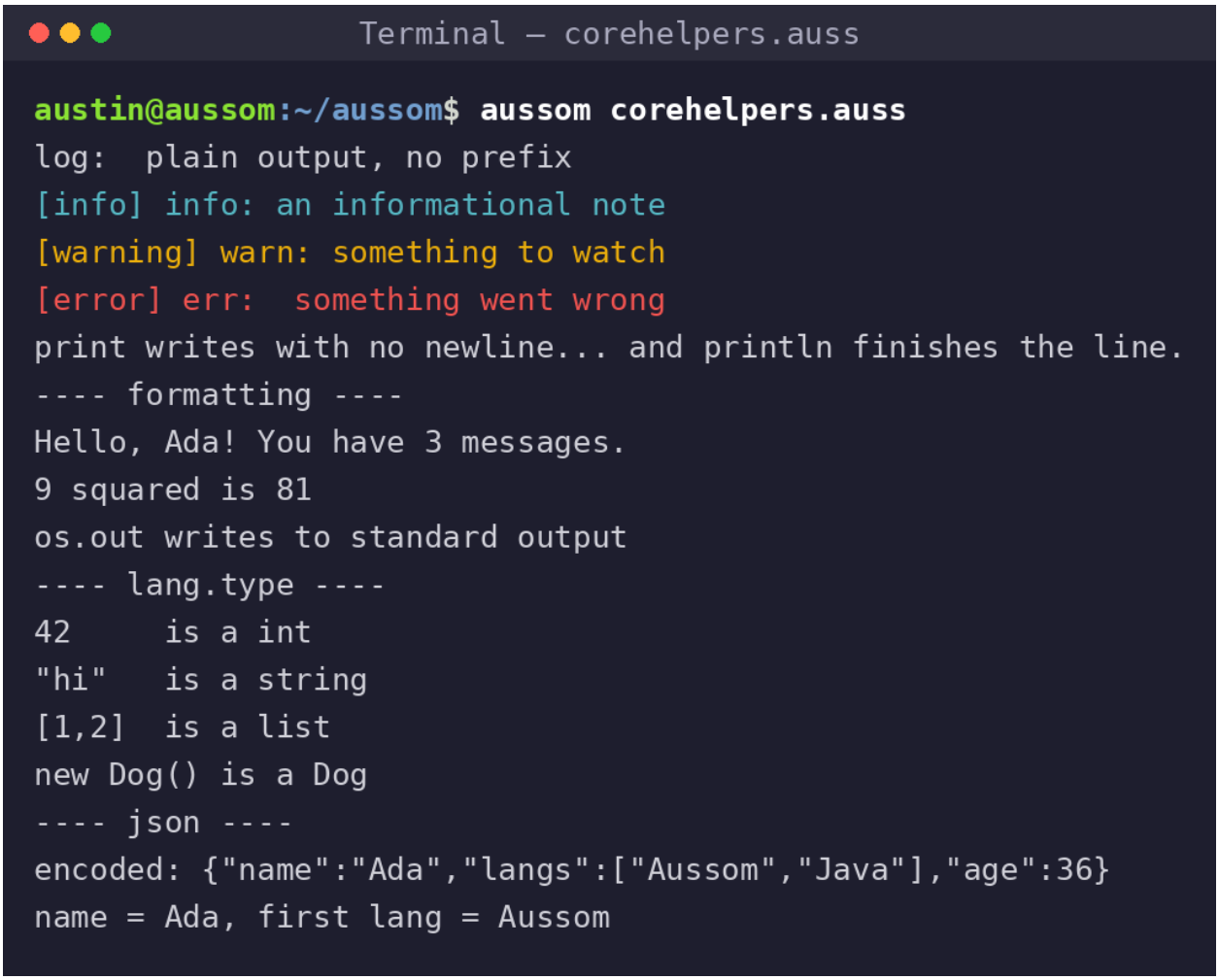


Figure 57: Running corehelpers.auss

A second short example, `ask.auss`, reads input with `os.readLine`:

```
include os;

name = os.readLine("What is your name? ");
age  = os.readLine("How old are you? ");
c.log("Hello, " + name + "! In 10 years you'll be " + (age.parseInt() + 10) +
↪   ".");
```

Typing `Ada` and `36` gives:

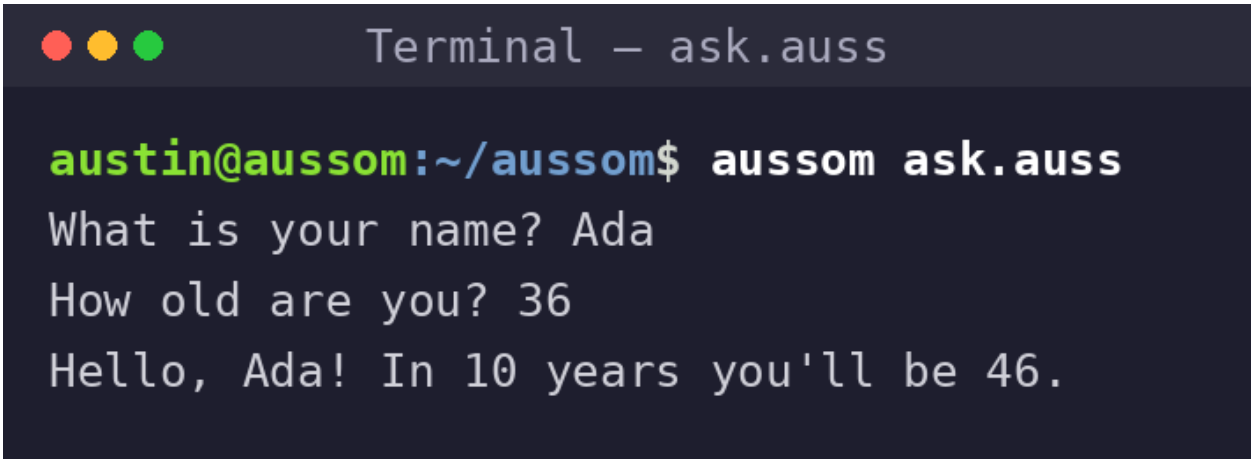


Figure 58: Running ask.auss

A third example, `saveobject.auss`, marshals a `Player` to disk and rehydrates it:

```
include file;

class Player {
    public name = null;
    public score = 0;
    public Player(Nm = null, Sc = 0) { this.name = Nm; this.score = Sc; }
    public describe() { return this.name + " has " + this.score + " points"; }
}

ada = new Player("Ada", 95);
file.write("player.save", ada.pack());          // save the object as
                                                // text
c.log("saved to player.save");

loaded = json.unpack(file.read("player.save"));  // rebuild the real
                                                 // object
c.log("type:   " + lang.type(loaded));
c.log("loaded: " + loaded.describe());
loaded.score = loaded.score + 5;
c.log("after:  " + loaded.describe());
```

Running it prints:

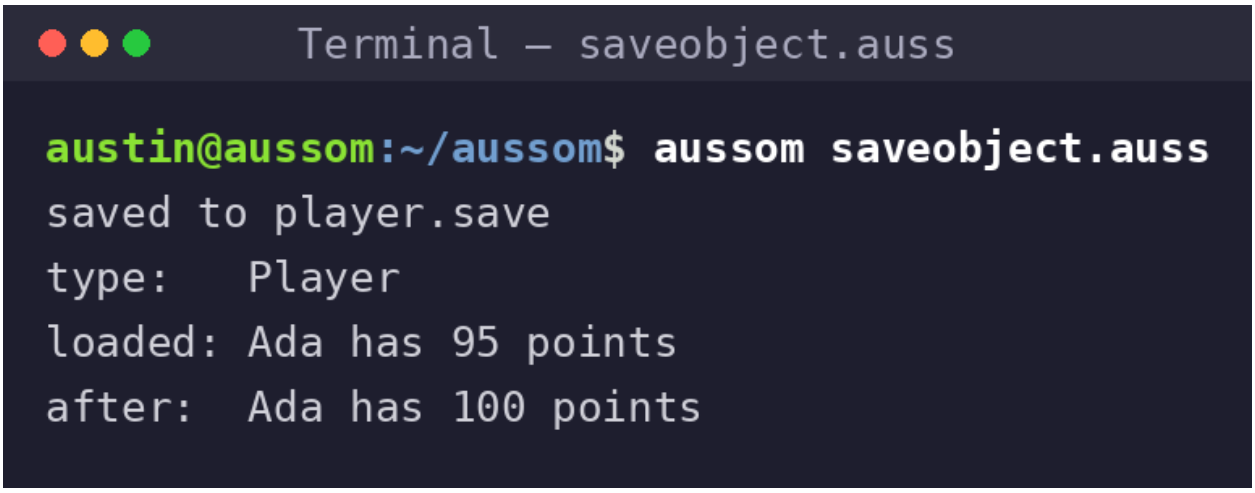


Figure 59: Running saveobject.auss

You can find all three examples here: `21-console`.

What You Learned

- The **console `c`** writes output at different **levels**: **`c.log`** (plain), **`c.info`** (`[info]`), **`c.warn`** (`[warning]`), **`c.err`** (`[error]`), plus **`c.print`** / **`c.println`** (without / with a trailing newline).
- The **`os`** module adds **`sprintf`** / **`printf`** for **formatted** output — templates with `{}`, `{N}`, and `{name}` placeholders (the same rules as a string's **`format`** method) — plus **`os.out`** and **`os.outErr`** to write a line to **standard output** or **standard error**.
- **`os.readLine(prompt)`** (also from `os`) reads a typed line as a string; convert it with **`parseInt`** / **`parseDouble`** / **`parseBool`**.
- **`lang.type(value)`** names a value's type (`"int"`, `"list"`, a class name, ...) — useful for checking data before you use it.
- **`value.toJson()`** turns data into JSON text; **`json.parse(text)`** turns JSON text back into a map or list. They're mirror images for saving and loading data.
- **`object.pack()`** and **`json.unpack()`** save and restore a **whole object** (its class *and* data) as text — for writing to disk or sending over a socket (*marshalling*). The class must be creatable with no arguments.

Next, in **Chapter 22**, we work with **dates and times**.

Chapter 22 - Dates and Times

Almost every real program deals with **time** at some point: stamping when something happened, scheduling a deadline, or working out how many days are left until an event. Aussom handles all of this with the **`Date`** type, which represents a single moment in time. `Date` is part of the auto-included standard library, so — like `c` and `lang` from the last chapter — you can use it without an `include`. This chapter shows how to create dates, format them for people to read, and do math with them.

Creating Dates

You make a date with **`new Date()`** — but there's an important catch. A bare `new Date()` does **not** give you the current time; it gives the **epoch**, a fixed reference moment (midnight, January 1st, 1970). To get **right now**, call **`.now()`** on it:

```
today = new Date().now();       // the current date and time
```

Think of `new Date()` as making a blank clock set to zero, and `.now()` as setting it to the current moment. There are two other ways to create a date:

- **`new Date(millis)`** — a specific moment, given as *milliseconds since the epoch* (the standard way computers store an instant as a single number).
- **`new Date().parse(text, pattern)`** — read a date out of a **string**, which is what you'll usually want when a date comes from a user or a file.

The **`parse`** call takes the text and a **pattern** that describes its layout:

```
launch = new Date().parse("2026-07-14 09:30:00", "yyyy-MM-dd HH:mm:ss");
```

The pattern `"yyyy-MM-dd HH:mm:ss"` says "four-digit year, dash, two-digit month, dash, day, space, hour, colon, minutes, colon, seconds." We'll break that pattern down in the next section — it's the same language `format` uses.

> **A note on time zones.** Aussom's `Date` works in **UTC** (Coordinated Universal Time, the world's reference clock) when it formats and reads dates. So the hour you get back is the UTC hour, not necessarily your local one. For a beginner program that's usually fine; just know that's why the time may not match your wall clock.

Formatting Dates

A `Date` on its own isn't text — to show it, you **format** it into a string with **`format`**, passing a **pattern** made of the same letter-codes `parse` uses. Each group of letters is a placeholder that gets filled with part of the date:

```
launch.format("yyyy-MM-dd")                    // "2026-07-14"
launch.format("yyyy-MM-dd HH:mm:ss")           // "2026-07-14 09:30:00"
launch.format("EEEE, MMMM d, yyyy")            // "Tuesday, July 14,
                                               // 2026"
```

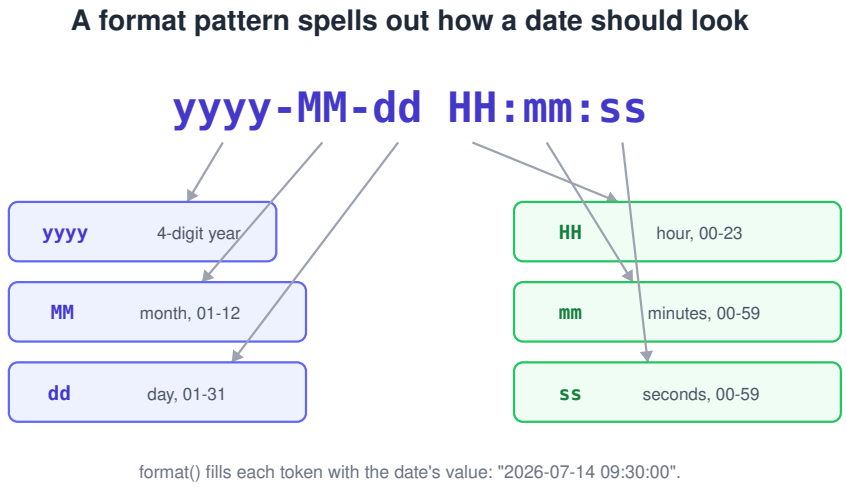


Figure 60: A format pattern spells out how a date should look

Here are the pattern letters you'll use most, with what each produces for the moment **2026-07-14 15:05:09**:

Pattern	Means	Result
yyyy	year, four digits	2026
yy	year, two digits	26
MM	month number (01-12)	07
MMM	month name, short	Jul
MMMM	month name, full	July
dd	day of month (01-31)	14
D	day of year	195
EEE	weekday, short	Tue
EEEE	weekday, full	Tuesday
a	AM / PM	PM
HH	hour, 24-hour (00-23)	15
hh	hour, 12-hour (01-12)	03
mm	minute (00-59)	05
ss	second (00-59)	09
SSS	fraction of a second (milliseconds)	250

Pattern	Means	Result
`G`	era	`AD`
`z`	time-zone name	`Z`
`Z`	time-zone offset	`+0000`

Two rules make the whole set easier to remember. **Repeating a letter zero-pads it**: a single `M` gives `7`, while `MM` gives `07` (and `MMM`/`MMMM` switch to the name). And to put **literal words** in a pattern, wrap them in single quotes so they aren't read as letters — `"h 'o''clock'"`. Call `format()` with no pattern at all and you get a sensible default (`"2026-07-14 09:30:00.000 +0000"`).

Aussom's dates are built on Java's `java.time` classes, so the pattern letters are exactly Java's **`DateTimeFormatter`** letters. The table above covers the everyday ones; for the complete list — quarters, week-of-year, localized forms, and more — see the official reference: Java DateTimeFormatter.

When you want just **one piece** of a date as a number or name, the `Date` type has a family of **getter** methods, so you don't have to format-and-parse:

```
launch.getYear()             // 2026
launch.getDayOfMonth()       // 14
launch.getMonthName()        // "JULY"
launch.getDayOfWeekName()    // "TUESDAY"
```

There are more getters in the same style — the useful ones include:

- **`getMonth`** / **`getDayOfMonth`** / **`getHours`** / **`getMinutes`** / **`getSeconds`** — the numeric parts.
- **`getDayOfWeek`** / **`getDayOfYear`** / **`getWeekOfYear`** — positions within the week and year.
- **`getMonthName`** / **`getDayOfWeekName`** — the names as text.

Date Math and Comparisons

Dates get really useful when you do arithmetic with them — "30 days from now," "is this before that," "how many days between." The **`add`** family shifts a date by some amount:

```
deadline = launch.copy().addDays(30);      // 30 days after launch
```

Notice the **`.copy()`**. The add methods change the date *in place*, so if you want to keep the original untouched, copy it first and shift the copy. (Without `copy()`, `launch.addDays(30)` would move `launch` itself.) The full add family is:

- **`addDays`** / **`addHours`** / **`addMinutes`** / **`addSeconds`** — shift by smaller units.
- **`addYears`** / **`addMonths`** — shift by larger ones.

To **compare** two dates, use the question-style methods, which return `true` or `false`:

```
launch.isBefore(deadline)    // true  - launch comes first
launch.isAfter(deadline)     // false
launch.isSame(deadline)      // false - not the same instant
```

And to measure the **gap** between two dates, **`between`** returns the amount of time from one to the other in whatever unit you name:

```
launch.between(deadline, "days")     // 30
```

The unit is a string — `"days"`, `"hours"`, `"minutes"`, and so on. The result is **positive** when the other date is later and **negative** when it's earlier. The available units are `millis`, `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, and `years`. (Aussom also provides a `dateunit` enum — for example `dateunit.days` — so you can use a named value instead of typing the raw string.)

Try It Yourself

This program creates dates, formats them, and does some date math. Save it as `dates.auss` and run it:

```
// the CURRENT time: use now(). A bare "new Date()" is the epoch (1970)!
today = new Date().now();
c.log("right now (UTC): " + today.format("yyyy-MM-dd HH:mm:ss"));

// a specific date, parsed from text
launch = new Date().parse("2026-07-14 09:30:00", "yyyy-MM-dd HH:mm:ss");
c.log("launch:    " + launch.format("EEEE, MMMM d, yyyy 'at' HH:mm"));
c.log("year " + launch.getYear() + ", " + launch.getMonthName() + ", day " +
↪   launch.getDayOfMonth());

// date math - copy() first so we don't change the original
deadline = launch.copy().addDays(30);
c.log("30 days later: " + deadline.format("yyyy-MM-dd"));

// comparisons and the gap between two dates
c.log("launch before deadline? " + launch.isBefore(deadline));
c.log("days between:           " + launch.between(deadline, "days"));
```

Running it prints (your first line — the current time — will differ):

(In the pattern `"... 'at' HH:mm"`, the word `at` is wrapped in single quotes so it prints literally instead of being read as pattern letters.) You can find this example here: `22-dates`.

What You Learned

- **`new Date().now()`** is the current moment. A bare **`new Date()`** is the **epoch**, not now; **`new Date(millis)`** is a specific instant.
- **`new Date().parse(text, pattern)`** reads a date from a string using a **pattern** of letter-codes (`yyyy`, `MM`, `dd`, `HH`, `mm`, `ss`, ...).
- **`format(pattern)`** turns a date into a string; the **getters** (`getYear`, `getMonthName`, `getDayOfWeekName`, ...) pull out one piece at a time. Formatting is done in **UTC**.
- The **add** family (`addDays`, `addMonths`, ...) shifts a date **in place** — use **`.copy()`** to keep the original. **`isBefore`** / **`isAfter`** / **`isSame`** compare dates, and **`between(other, unit)`** measures the gap.

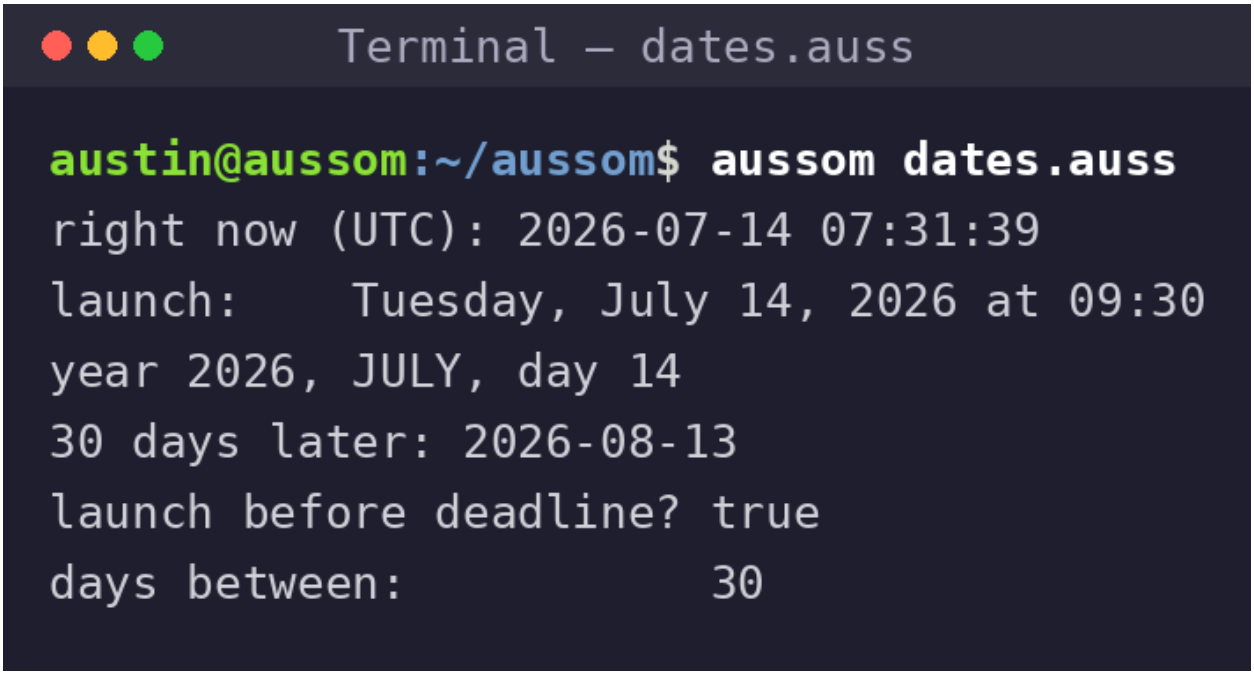


Figure 61: Running dates.auss

Next, in **Chapter 23**, we crunch numbers with the **`math`** module.

Chapter 23 - Math

Beyond the basic +, -, *, and / you met back in Chapter 9, real programs often need more: rounding a price, finding a square root, working with angles, or rolling a random number. Aussom bundles all of that into the **`math`** module. Unlike the console and `Date`, `math` is **not** auto-included, so every program in this chapter starts with one line:

```
include math;
```

With that in place, you call its functions by name, like `math.sqrt(...)`. Let's walk through the ones you'll reach for most.

Rounding, Powers, Roots, and Trig

Rounding turns a decimal number into a whole one. There are three ways to do it, depending on which direction you want to go:

```
math.round(3.7)     // 4  - to the NEAREST whole number
math.ceil(3.2)      // 4  - always UP
math.floor(3.9)     // 3  - always DOWN
```

`round` goes to whichever whole number is closest; **`ceil`** (short for *ceiling*) always rounds up; **`floor`** always rounds down. This picture shows all three acting on `3.2` and `3.7`:

Powers and roots are just as direct:

```
math.pow(2, 10)     // 1024.0 - 2 to the 10th power (base, exponent)
math.sqrt(144)      // 12.0   - the square root
```

`pow(base, exponent)` raises the first number to the power of the second; **`sqrt`** finds the square root; and **`cbrt`** finds the cube root. A few everyday helpers round out the basics:

- **`abs`** — the absolute value (distance from zero), so `abs(-7)` is 7.
- **`max`** / **`min`** — the larger / smaller of two numbers.
- **`signum`** — the sign of a number: `-1`, `0`, or `1`.

  A note on number types. Most `math` functions hand back a **double** (a decimal number), even when the answer is whole — that's why `sqrt(144)` prints as `12.0`, not 12. If you need a plain integer, call **`.toInt()`** on the result (for example `math.floor(x).toInt()`).

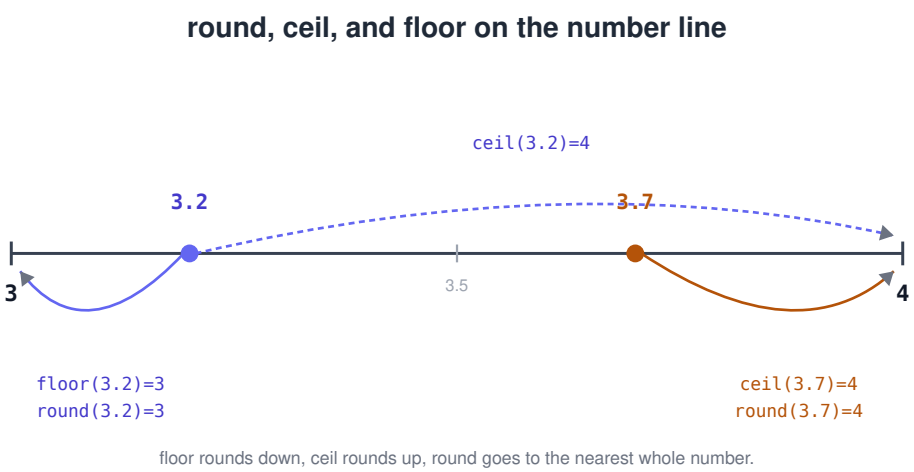


Figure 62: round, ceil, and floor on the number line

Finally, **trigonometry** — the math of angles, used in graphics, games, and anything involving circles or waves. The catch is that these functions measure angles in **radians**, not degrees, so you convert first with **`toRad`**:

```
math.sin(math.toRad(90))    // 1.0 - the sine of a 90-degree angle
```

The trig family includes **`sin`**, **`cos`**, and **`tan`**, their inverses **`asin`** / **`acos`** / **`atan`**, and **`toRad`** / **`toDeg`** to convert between radians and degrees. If you've never used trig, don't worry — you can skip it until a project needs it.

Random Numbers

To get a **random** number, call **`rand()`**. It returns a `double` somewhere from `0` up to (but not including) 1:

```
math.rand()     // e.g. 0.472... - a different value each time
```

A value between 0 and 1 doesn't sound like much, but it's the building block for *any* random range. To get a whole number from 1 to 6 — a dice roll — you scale it up, floor it, and shift it:

```
roll = (math.floor(math.rand() * 6) + 1).toInt();    // a whole number,
                                                     // 1 to 6
```

Reading that inside out: `rand() * 6` spreads the value across `0–6`, `floor(...)` drops it to a whole number `0–5`, `+ 1` shifts the range to `1–6`, and `.toInt()` makes it a clean integer. The same recipe — `floor(rand() * size) + start` — gives you a random whole number in any range you like.

Constants

Some numbers come up so often in math that `math` provides them ready-made, as functions you call:

```
math.pi()     // 3.141592653589793 - the ratio of a circle's
              // circumference to its diameter
math.e()      // 2.718281828459045 - Euler's number, the base of natural
              // logarithms
```

`pi()` is π, essential whenever circles or angles are involved; **`e()`** is Euler's number, which shows up in growth and probability. You use them like any other value — for instance, `math.pi() / 2` is a right angle in radians.

Try It Yourself

This program tours the `math` module — constants, rounding, powers, a bit of trig, and a random dice roll. Save it as `math.auss` and run it:

```
include math;

// constants
c.log("pi = " + math.pi());
c.log("e  = " + math.e());

// rounding
c.log("round(3.7) = " + math.round(3.7));
c.log("ceil(3.2)  = " + math.ceil(3.2));
c.log("floor(3.9) = " + math.floor(3.9));

// powers and roots
c.log("2 ^ 10    = " + math.pow(2, 10));
c.log("sqrt(144) = " + math.sqrt(144));

// everyday helpers
c.log("abs(-7)   = " + math.abs(-7));
c.log("max(3, 9) = " + math.max(3, 9));

// trig - angles are in radians; toRad converts from degrees
c.log("sin(90 deg) = " + math.sin(math.toRad(90)));

// random: rand() gives a double in [0, 1); scale it for a dice roll
roll = (math.floor(math.rand() * 6) + 1).toInt();
c.log("dice roll (1-6): " + roll);
```

Running it prints (the last line — the dice roll — will differ each run):

You can find this example here: `23-math`.

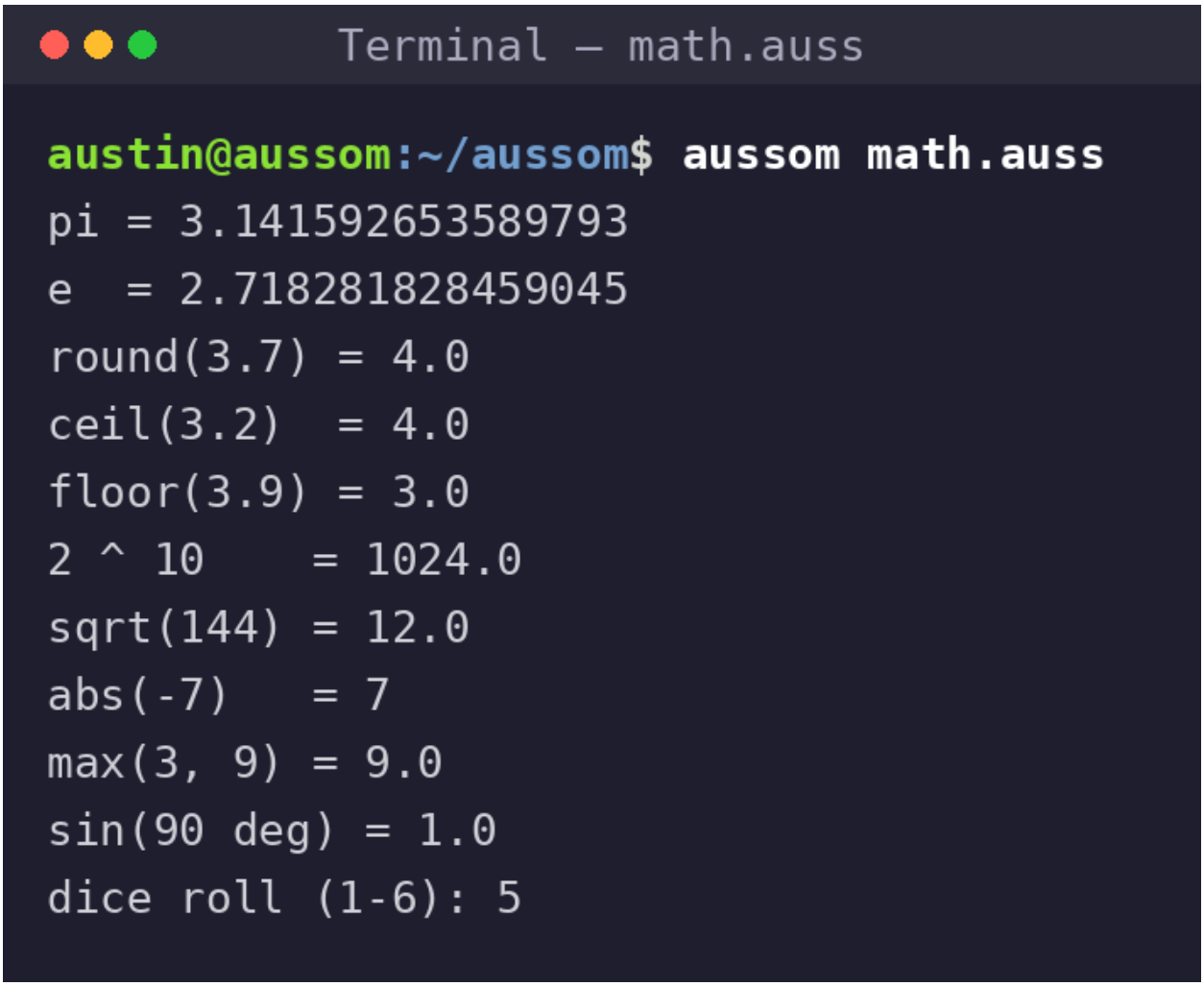


Figure 63: Running math.auss

What You Learned

- The **`math`** module needs **`include math;`**, then you call functions by name.
- **Rounding: `round`** (nearest), **`ceil`** (up), **`floor`** (down). **Powers/roots: `pow(base, exp)`**, **`sqrt`**, **`cbrt`**. Plus **`abs`**, **`max`**, **`min`**.
- Most `math` functions return a **double**; use **`.toInt()`** when you need a whole number.
- **Trig** (`sin`, `cos`, `tan`, ...) works in **radians** — convert with **`toRad`**.
- **`rand()`** gives a `double` in `[0, 1)`; the recipe `floor(rand() * size) + start` turns it into a random whole number in any range.
- **`pi()`** and **`e()`** are the built-in constants.

Next, in **Chapter 24**, we work with text tools — **regular expressions**, **encoding**, and **unique IDs**.

Chapter 24 - Text Tools: Regex, Encoding, and IDs

This chapter gathers three handy text tools that live in Aussom's **`util`** module: **regular expressions** for finding patterns in text, **encoding** for turning text into safe, portable characters, and **unique IDs** for labeling things. Like `math`, the `util` module isn't automatic, so each program here begins with:

```
include util;
```

Matching Text with `regex`

A **regular expression** (or *regex*) is a small pattern that describes a *shape* of text — "one or more digits," "an email address," "a word starting with a capital." It lets you search and edit text by pattern instead of by exact match. Aussom's **`regex`** module runs these patterns for you.

The core call is **`match`**, which finds **every** place the pattern occurs and returns them as a **list** of strings:

```
text = "Order 12 shipped, order 7 pending, order 108 done.";
nums = regex.match("[0-9]+", text);      // finds each run of digits
// nums is now the list: ["12", "7", "108"]
```

The pattern `"[0-9]+"` reads as "one or more characters in the range 0 to 9" — in other words, a whole number. `regex.match` scans the text and collects each stretch that fits. If nothing matches, you get an **empty list** back.

When you only want a single hit, or want to **replace** matches, `regex` has focused calls:

```
// "12"  - just the first match
regex.matchFirst("[0-9]+", text)
// "Order # shipped, order # pending, order # done."
regex.replace("[0-9]+", "#", text)
// only the first number is replaced
regex.replaceFirst("[0-9]+", "#", text)
```

The full set is:

- **`match`** — a list of every match.
- **`matchFirst`** / **`matchLast`** — just the first / last match, as a string.

- **`replace`** — swap **all** matches for a replacement string.
- **`replaceFirst`** — swap only the first.

  Watch the backslashes. Many regex shortcuts use a backslash — `\d` for a digit, `\w` for a word character. But in an Aussom string, a backslash is itself special, so you must **double it**: write `"\\d+"` to mean the regex `\d+`. (The bracket form `"[0-9]+"` above avoids the issue, which is why it's a friendly place to start.)

Regular expressions are a big topic — this section only scratches the surface of what patterns can describe. The good news is that Aussom doesn't invent its own regex dialect: it uses **Java's regular expressions** under the hood (the `java.util.regex.Pattern` engine). So anything you learn about Java regex works here unchanged. To go deeper — character classes, groups, quantifiers, look-ahead, and the rest — search for "**Java regular expressions**" or read the official reference: java.util.regex.Pattern.

Encoding with base64 and hex

Encoding rewrites data using a limited, safe set of characters so it can travel through systems that only handle plain text — email, URLs, config files. **base64** and **hex** (hexadecimal) are two common schemes. They don't hide or encrypt data; they just repackage it.

Most of the time you have a **string** to encode, and base64 handles that directly. **`base64.encode`** takes a string and returns the base64 text; **`base64.decode`** does the reverse:

```
msg = "Hello, Aussom!";
encoded = base64.encode(msg);      // "SGVsbG8sIEF1c3NvbSE="
decoded = base64.decode(encoded);  // "Hello, Aussom!"
```

That's the whole round trip — string in, string out, no extra steps. The base64 text can contain the characters +, /, and =, which are fine in most places but can cause trouble in a URL or a filename. For those spots, **`base64.encodeSafe`** produces a form using only `0-9` and `a-f`, safe to drop **anywhere**, with **`base64.decodeSafe`** to read it back:

```
safe = base64.encodeSafe(msg);      // "53475673...53d" - safe in URLs,
                                    // filenames, headers
base64.decodeSafe(safe);            // "Hello, Aussom!"
```

hex turns each byte into two hexadecimal characters. It works on raw **bytes** rather than a string, so you first place the text in a **`Buffer`** — a fixed-size box of bytes (you'll meet `Buffer` fully in Chapter 25). Size it to your text with `#text`, add the string, then encode:

```
buf = new Buffer(#msg);             // a buffer sized to hold the text
buf.addString(msg);
hex.encode(buf)                     // "48656c6c6f2c20417573736f6d21"
hex.decode(hexString).getString()  // "Hello, Aussom!" (decode returns a
                                   // Buffer)
```

So base64 works straight on a string, while hex routes through a Buffer of bytes. If you ever need base64 on binary data instead of a string, **`base64.encodeBinary`** and **`base64.decodeBinary`** take and return a `Buffer` the same way hex does.

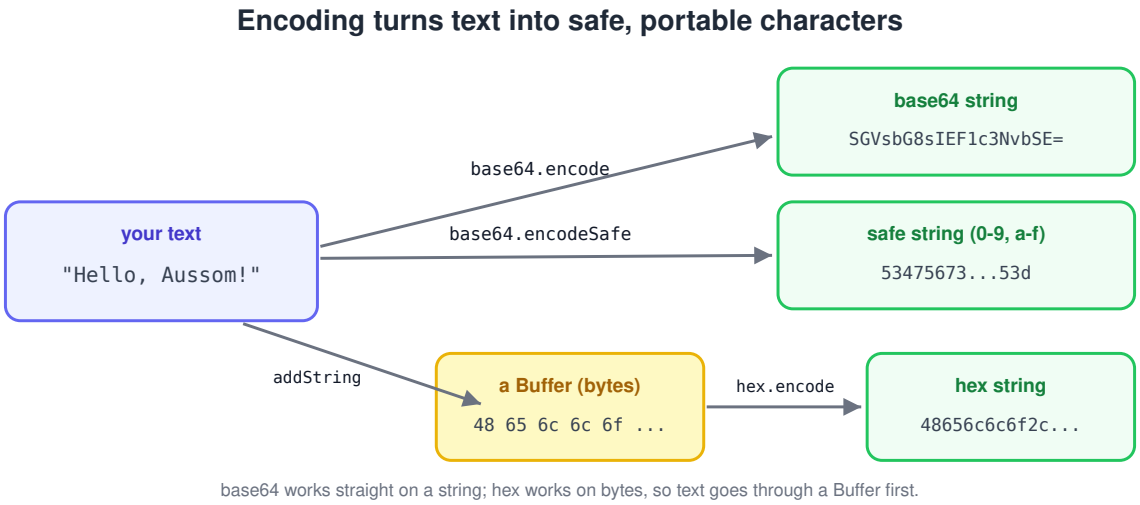


Figure 64: Encoding turns text into safe, portable characters

Generating Unique IDs with `uuid`

Often you need a label that's guaranteed to be **unique** — an order number, a session token, a filename that won't collide with any other. A **UUID** (Universally Unique Identifier) is a long string built to be unique across every machine on earth. The **`uuid`** module makes one with **`get`**:

```
uuid.get()      // e.g. "1b0e171a-88ef-44b2-8104-e05ab0f4a448"
```

Every call returns a brand-new 36-character ID, so you never have to invent or track your own. There's also **`getSecure`**, which generates the ID in a harder-to-predict way (it uses SHA-1 internally) — reach for it when the ID shouldn't be guessable, such as a security token.

Try It Yourself

This program uses all three tools — a regex search and replace, a base64 and hex round trip, and a fresh UUID. Save it as `text.auss` and run it:

```
include util;

// ---- regex: find and replace with patterns ----
text = "Order 12 shipped, order 7 pending, order 108 done.";
nums = regex.match("[0-9]+", text);           // list of every match
c.log("numbers: " + nums.join(", "));
c.log("first:   " + regex.matchFirst("[0-9]+", text));
c.log("masked:  " + regex.replace("[0-9]+", "#", text));

// ---- base64: turn text into safe, portable characters ----
msg = "Hello, Aussom!";
encoded = base64.encode(msg);                 // string in, base64
                                              // string out
```

```
c.log("base64:  " + encoded);
c.log("decoded: " + base64.decode(encoded));

// a variant that's safe in URLs, filenames, and headers (0-9, a-f only)
c.log("safe:    " + base64.encodeSafe(msg));

// ---- hex: works on raw bytes, via a Buffer ----
hbuf = new Buffer(#msg);
hbuf.addString(msg);
c.log("hex:     " + hex.encode(hbuf));

// ---- uuid: a unique identifier ----
c.log("uuid:    " + uuid.get());
```

Running it prints (the `uuid` line will differ every time):

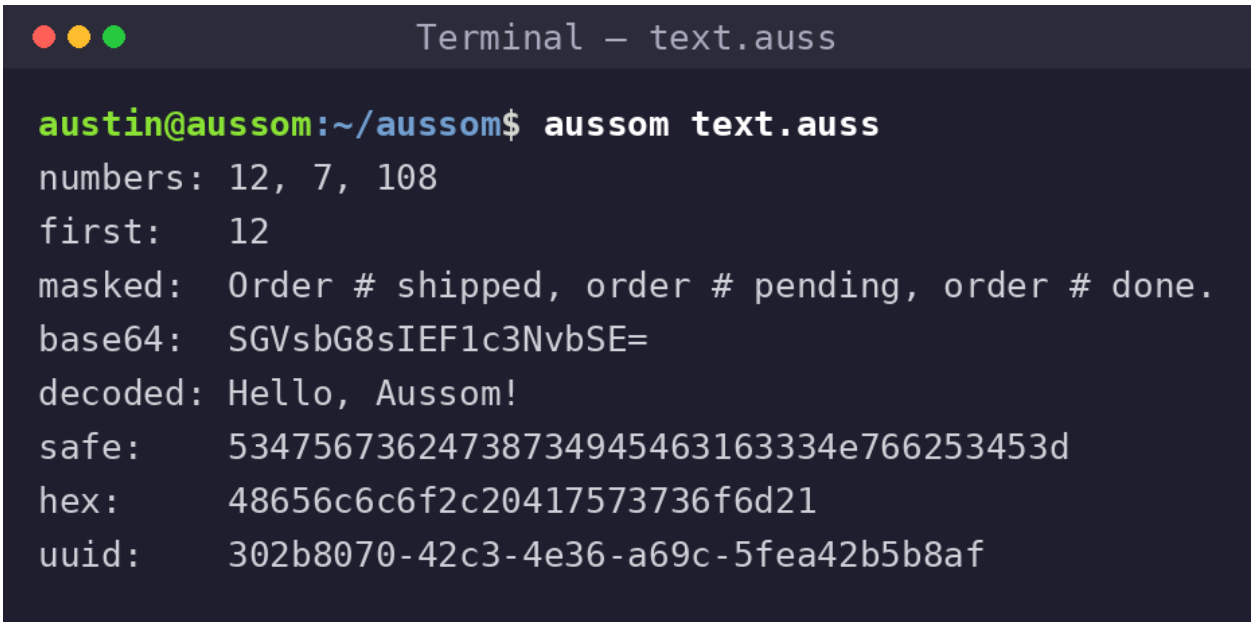


Figure 65: Running text.auss

We used the list's **`join(", ")`** from Chapter 10 to print the matches as one tidy line. You can find this example here: `24-text`.

What You Learned

- The **`util`** module (with **`include util;`**) holds `regex`, `base64`, `hex`, and `uuid`.
- **`regex.match(pattern, text)`** returns a list of every match; **`matchFirst`** / **`matchLast`** return one; **`replace`** / **`replaceFirst`** swap matches out. Double any backslash in a pattern (`"\\d+"`).
- **`base64.encode`** / **`base64.decode`** work straight on **strings** (the everyday path); **`base64.encodeSafe`** / **`decodeSafe`** give a URL/filename-safe form, and **`base64.encodeBinary`** / **`decodeBinary`**

handle raw bytes. **`hex`** works on **bytes**, so put text in a **`Buffer`** first (`new Buffer(#text)` then `addString`).
- **`uuid.get()`** makes a unique 36-character ID; **`uuid.getSecure()`** makes a harder-to-guess one.

Next, in **Chapter 25**, we read and write **files** — and meet the **`Buffer`** in full.

Chapter 25 - Files and the Filesystem

Programs that only keep data in memory forget everything the moment they stop. To *remember* — to save a document, load a config, keep a log — a program reads and writes **files**. Aussom handles files through the **`file`** module, a set of tools for reading, writing, and managing files and folders. Like the other CLI modules, it isn't automatic, so every program here starts with:

```
include file;
```

Reading and Writing Text Files

The two calls you'll use most are **`write`** and **`read`**. **`file.write`** takes a filename and a string, and saves that text to the file (creating it, or replacing what was there). **`file.read`** takes a filename and hands back the whole file as a string:

```
file.write("greeting.txt", "Hello, files!\n");   // save some text
text = file.read("greeting.txt");                 // read it all back
c.log(text);                                       // Hello, files!
```

By default `write` **replaces** the file's contents. To **add** to the end instead, pass `true` as a third argument — that's *append* mode:

```
file.write("greeting.txt", "A second line.\n", true);   // append, don't
                                                        // replace
```

Often a text file is really a *list of lines*, and you'd rather work with it that way than as one big string. **`file.readLines`** reads a file into a **list**, one entry per line (with the line breaks stripped off), and **`file.writeLines`** does the reverse — writes a list of strings as separate lines:

```
lines = file.readLines("greeting.txt");   // ["Hello, files!", "A second
                                          // line."]
c.log("line count: " + #lines);

file.writeLines("fruits.txt", ["apple", "banana", "cherry"]);
```

So there are two matched pairs: **`read`/`write`** for a file as one string, and **`readLines`/`writeLines`** for a file as a list of lines. Both writers take the same optional `true` third argument to append.

Working with Paths and Directories

A **path** is the text that names where a file lives, like `/home/ada/reports/july.csv`. The `file` module has tools to take paths apart, put them together, and manage the folders they point to — and the path-splitting tools work on the *text* alone, whether or not the file exists.

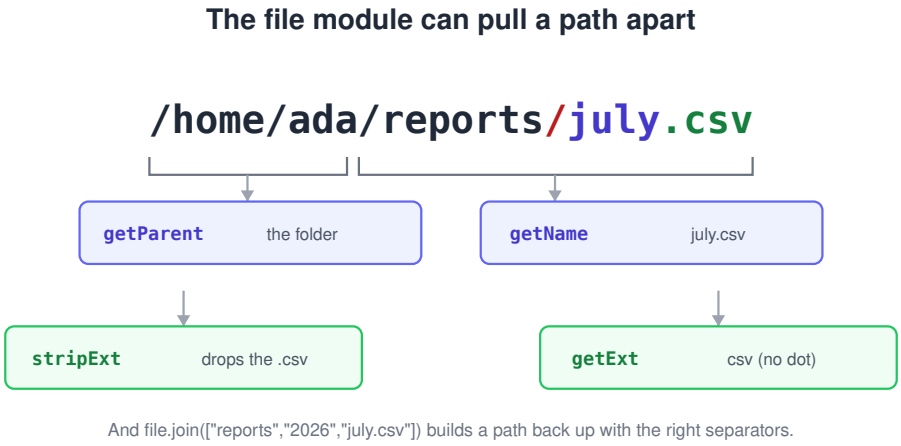


Figure 66: The file module can pull a path apart

The path pieces are:

- **`file.getName(path)`** — just the filename at the end (`july.csv`).
- **`file.getParent(path)`** — the folder containing it (`/home/ada/reports`).
- **`file.getExt(path)`** — the extension, with no dot (`csv`); **`file.stripExt`** gives the name *without* the extension.
- **`file.join(parts)`** — builds a path from a list of pieces, inserting the right separator: `file.join(["reports", "2026", "july.csv"])` → `reports/2026/july.csv`.

To ask **about** a path, and to manage folders:

```
file.exists("greeting.txt")    // true if it's there at all
file.isFile("greeting.txt")    // true if it's a file
file.isDir("data")             // true if it's a folder
file.mkdirs("data/cache")      // create a folder (and any missing
                               // parents)
```

The other everyday management calls are:

- **`file.mkdir`** / **`file.mkdirs`** — make one folder / a whole nested path of them.
- **`file.rm`** — delete a file; **`file.rmr`** — delete a folder and everything in it.
- **`file.rename`** — rename or move; **`file.cp`** — copy a file.
- **`file.ls(dir)`** — list a folder's contents. It returns a **list of maps**, one per entry, each describing a file (its `name`, whether it's a directory, its size, and more), so you read `entry["name"]` and `entry["isDirectory"]` from each.

For finding files by pattern there's also **`file.glob`** (match a wildcard like `*.txt`) and **`file.walk`** (visit every file under a folder), which you can explore when you need them.

Binary Data with `Buffer`

Text files are the common case, but sometimes you deal in raw **bytes** — an image, a compressed file, a network message. You met the **`Buffer`** briefly in Chapter 24; here it is in full. A `Buffer` is a **fixed-size box of bytes** that you fill with typed values and read back the same way.

You make one with a size, then **add** values to it — a byte, a whole number, a string — each one written right after the last:

```
buf = new Buffer(32);       // a 32-byte buffer
buf.addByte(65);            // one byte (65 is the letter 'A')
buf.addInt(2026);           // a 4-byte whole number
buf.addString("Aussom");    // the text's bytes
c.log(buf.size());          // 32 - the buffer's total capacity
```

To read the values back, rewind to the start with **`readSeek(0)`**, then pull them out in the same order with the matching **`get`** calls:

```
buf.readSeek(0);            // back to the beginning
buf.getByte()              // 65
buf.getInt()               // 2026
buf.getStringAt(6)         // "Aussom" (6 bytes of text)
```

The add/get family covers the common sizes — **`addByte`**, **`addInt`**, **`addShort`**, **`addLong`**, **`addFloat`**, **`addDouble`**, and **`addString`**, each with a matching getter. The key thing to remember from Chapter 24 holds: a `Buffer` has a **fixed size**, so make it big enough for what you put in.

To save bytes to a file, use **`file.writeBinary`** (which takes a `Buffer`) and **`file.readBinary`** (which hands one back):

```
file.writeBinary("data.bin", buf);            // save the raw bytes
loaded = file.readBinary("data.bin");         // read them into a new
                                              // Buffer
c.log(loaded.getString());                     // read the text back out
```

Try It Yourself

The first program reads and writes text, and takes a path apart. Save it as `files.auss` and run it:

```
include file;

file.write("greeting.txt", "Hello, files!\n");
file.write("greeting.txt", "A second line.\n", true);   // append
c.log("contents:");
c.log(file.read("greeting.txt"));

lines = file.readLines("greeting.txt");
c.log("line count: " + #lines);
c.log("first line: " + lines[0]);
```

```
path = "/home/ada/reports/july.csv";
c.log("name:   " + file.getName(path));
c.log("parent: " + file.getParent(path));
c.log("ext:    " + file.getExt(path));
c.log("joined: " + file.join(["reports", "2026", "july.csv"]));

file.mkdirs("data/cache");
c.log("data/cache is a dir? " + file.isDir("data/cache"));

file.rm("greeting.txt");
file.rmr("data");
c.log("cleaned up. greeting.txt exists? " + file.exists("greeting.txt"));
```

Running it prints:

A second program, `buffer.auss`, works with raw bytes:

```
include file;

buf = new Buffer(32);
buf.addByte(65);
buf.addInt(2026);
buf.addString("Aussom");
c.log("buffer size: " + buf.size());

buf.readSeek(0);
c.log("byte: " + buf.getByte());
c.log("int:  " + buf.getInt());
c.log("text: " + buf.getStringAt(6));

msg = new Buffer(#"binary data");
msg.addString("binary data");
file.writeBinary("data.bin", msg);
loaded = file.readBinary("data.bin");
c.log("from file: " + loaded.getString());
file.rm("data.bin");
```

Running it prints:

You can find both examples here: `25-files`.

What You Learned

- The **`file`** module (with **`include file;`**) reads and writes files. **`read`** / **`write`** treat a file as one string (`write` replaces, or appends with a `true` third argument); **`readLines`** / **`writeLines`** treat it as a list of lines.

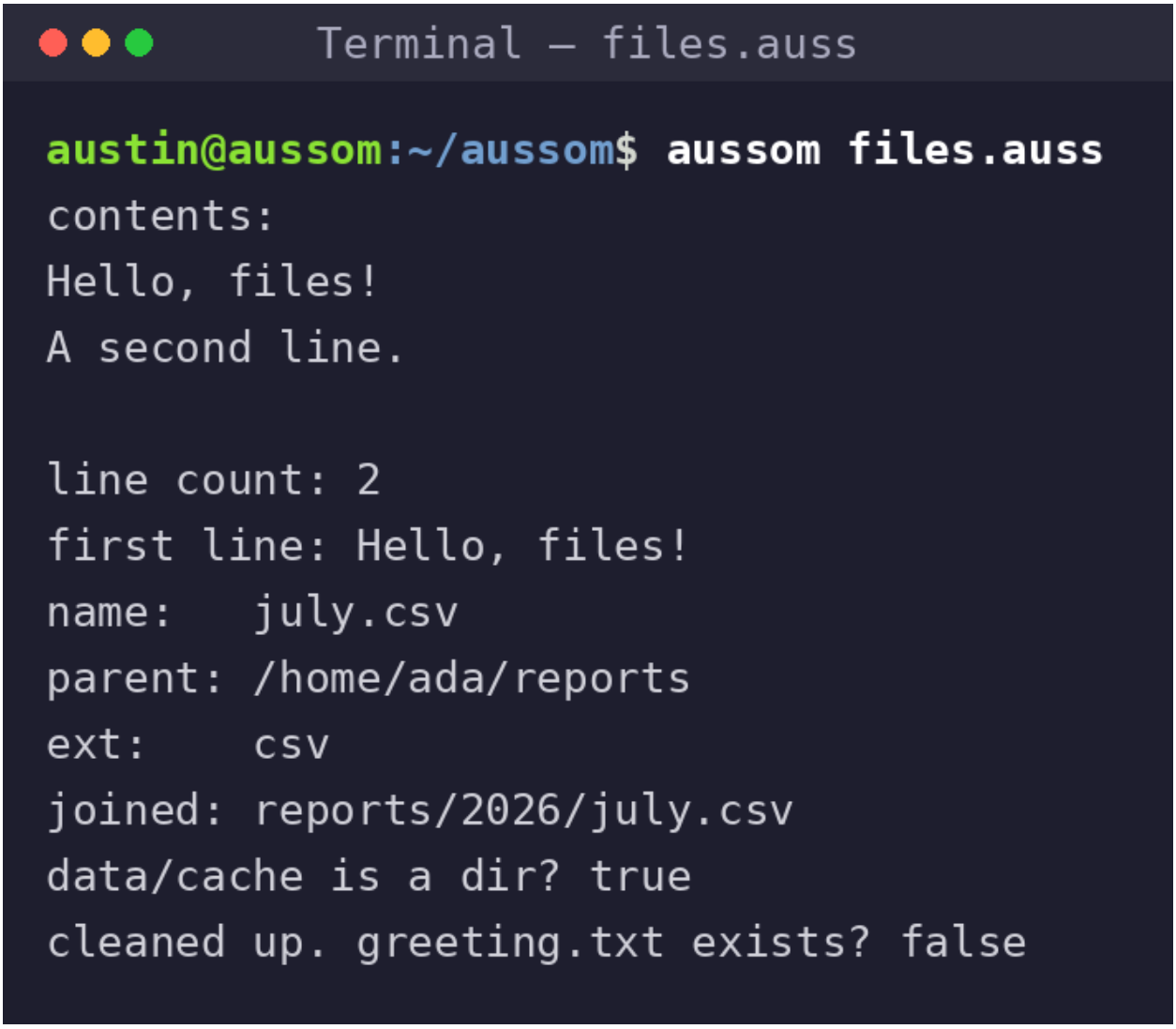


Figure 67: Running files.auss

Figure 68: Running buffer.auss

- Path tools work on the text alone: **`getName`**, **`getParent`**, **`getExt`**, **`stripExt`**, and **`join`**. Folder tools include **`exists`**, **`isDir`**, **`mkdirs`**, **`rm`** / **`rmr`**, **`rename`**, and **`ls`** (which returns a list of detail maps).
- A **`Buffer`** is a fixed-size box of bytes: **`add`**/**`get`** a byte, int, string, and so on, and **`readSeek(0)`** to rewind. **`file.writeBinary`** / **`file.readBinary`** save and load a `Buffer`.

Next, in **Chapter 26**, we read and write structured **data formats** — JSON, XML, and YAML.

Chapter 26 - Data Formats: JSON, XML, and YAML

When programs share data — with a config file, a web service, or each other — they need an agreed-upon **text format** so both sides read it the same way. Three formats show up everywhere: **JSON**, **XML**, and **YAML**. They look different, but they all do the same job: represent structured data (maps, lists, values) as plain text you can save or send. Aussom can read and write all three, turning that text into ordinary Aussom data and back.

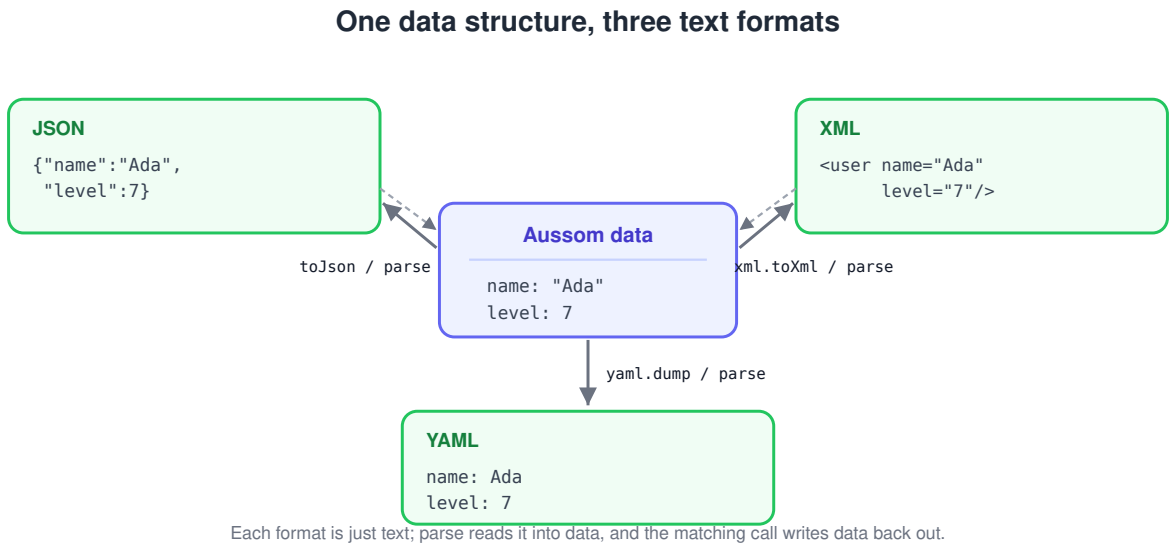


Figure 69: One data structure, three text formats

You met JSON back in Chapter 21. This chapter puts it alongside XML and YAML, and pairs each with the files from Chapter 25.

Reading and Writing JSON

JSON is the most common of the three. From Chapter 21, recall the two calls: any value's **`toJson()`** turns it into a JSON string, and the auto-included **`json.parse`** turns a JSON string back into Aussom data. Combined with the `file` module, that's all you need to save and load a **JSON file**:

```
include file;

config = {"name": "Ada", "level": 7, "tags": ["cli", "jvm"]};
file.write("config.json", config.toJson());       // write data as a
                                                  // JSON file

loaded = json.parse(file.read("config.json"));     // read the file back
                                                   // into data
c.log(loaded["name"]);            // "Ada"
c.log(loaded["tags"].join(", "));  // "cli, jvm"
```

`config.toJson()` produces the text `{"name":"Ada","level":7,"tags":["cli","jvm"]}`, which `file.write` saves; reading and parsing rebuilds the same map and list. This write-then-parse pattern is how most programs keep settings and data between runs.

Parsing XML

XML wraps data in tags, like `<title>Aussom</title>`. It's common in older systems, office documents, and many configuration files. The **`xml`** module reads it — bring it in with `include xml;` — and **`xml.parse`** turns an XML string into a **tree of maps**:

```
include xml;

doc = "<book id=\"42\"><title>Aussom</title><author>Ada</author></book>";
tree = xml.parse(doc);
```

XML is more layered than JSON, so the result is more layered too. Each element becomes a map with a **`name`** (the tag), a **`type`** (`ELEMENT`, `TEXT`, or `DOCUMENT`), an **`attributes`** map, and a **`children`** list. The text inside a tag is itself a child node. You walk down the tree with indexes and keys to reach a value:

```
book = tree["children"][0];                // the root <book> element
c.log(book["name"]);                       // "book"
c.log(book["attributes"]["id"]);           // "42"  - an attribute

title = book["children"][0];               // the <title> element
c.log(title["children"][0]["value"]);      // "Aussom" - its text
```

It takes a few hops, but the rule is consistent: elements have `children`, attributes live in `attributes`, and text sits in a child node's `value`. To go the other way, **`xml.toXml(map)`** turns a map structure back into an XML string (pretty-printed by default), and helpers like `xml.newNode` help you build one up.

Reading and Writing YAML

YAML is designed to be easy for *people* to read and write — it uses indentation instead of brackets or tags, which is why it's popular for configuration files. The **`yaml`** module (`include yaml;`) mirrors

JSON's two calls: **`yaml.parse`** reads a YAML string into Aussom data, and **`yaml.dump`** turns data into a YAML string:

```
include yaml;

settings = yaml.parse("host: localhost\nport: 8080\n");
c.log(settings["host"] + ":" + settings["port"]);   // localhost:8080

out = yaml.dump({"ok": true, "count": 3});
c.log(out);
// count: 3
// ok: true
```

Because YAML is so often stored in files, `yaml` also offers file shortcuts so you don't have to read the text yourself:

- **`yaml.parseFile(name)`** — read and parse a YAML file in one step.
- **`yaml.dumpFile(value, name)`** — dump a value straight to a YAML file.
- **`yaml.parseAll`** / **`yaml.dumpAll`** — for files that hold *several* documents at once (YAML allows more than one, separated by `---`).

Try It Yourself

This program reads and writes all three formats. Save it as `dataformats.auss` and run it:

```
include file;
include xml;
include yaml;

// JSON: write data to a file and read it back
config = {"name": "Ada", "level": 7, "tags": ["cli", "jvm"]};
file.write("config.json", config.toJson());
loaded = json.parse(file.read("config.json"));
c.log("JSON:  name=" + loaded["name"] + " level=" + loaded["level"] + "
↪   tags=" + loaded["tags"].join("/"));

// XML: parse a document into a tree of maps
doc = "<book id=\"42\"><title>Aussom</title><author>Ada</author></book>";
tree = xml.parse(doc);
book = tree["children"][0];
title = book["children"][0];
c.log("XML:   id=" + book["attributes"]["id"] + " title=" +
↪   title["children"][0]["value"]);

// YAML: parse and produce
settings = yaml.parse("host: localhost\nport: 8080\n");
c.log("YAML:  " + settings["host"] + ":" + settings["port"]);
c.log("--- dumped YAML ---");
```

```
c.log(yaml.dump({"ok": true, "count": 3}));

file.rm("config.json");
```

Running it prints:

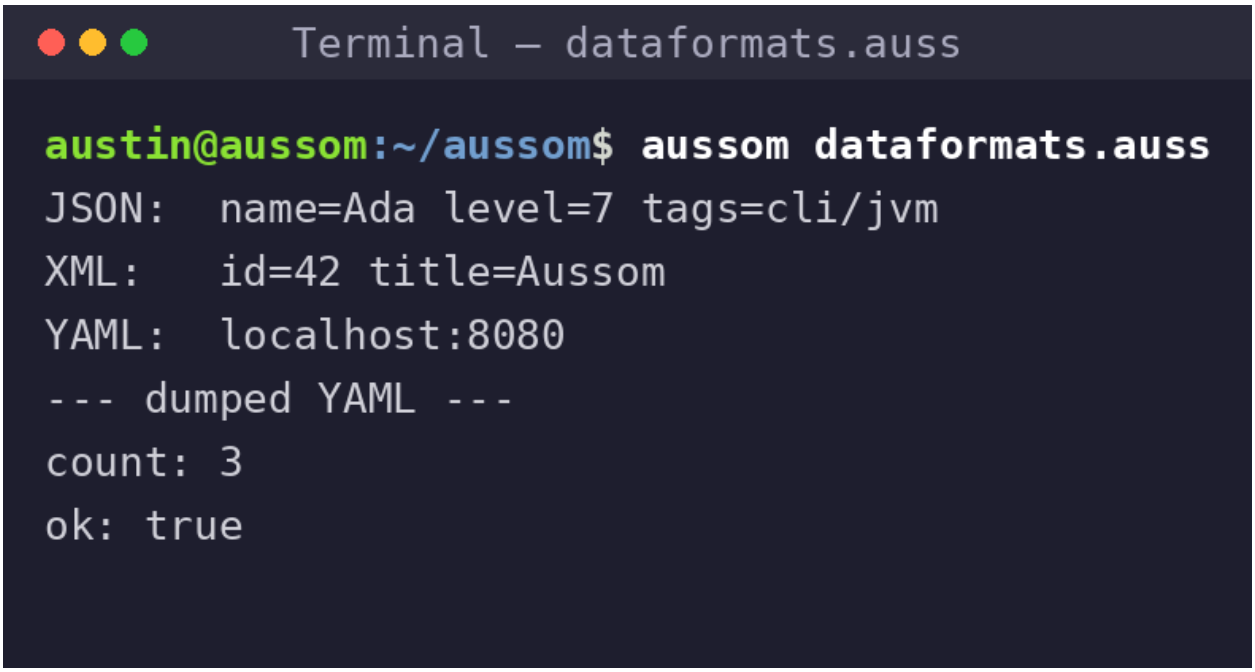


Figure 70: Running dataformats.auss

You can find this example here: `26-data-formats`.

What You Learned

- **JSON**, **XML**, and **YAML** are three text formats for the same job: representing structured data. Aussom parses each into ordinary maps and lists, and writes them back.
- **JSON** — any value's **`toJson()`** and the auto-included **`json.parse`**; pair with the `file` module to read and write JSON files.
- **XML** — **`xml.parse`** (with `include xml;`) builds a **tree of maps** (each element has `name`, `attributes`, and `children`; text sits in a child's `value`); **`xml.toXml`** writes one back.
- **YAML** — **`yaml.parse`** and **`yaml.dump`** (with `include yaml;`), plus **`yaml.parseFile`** / **`yaml.dumpFile`** to go straight to and from a file.

Next, in **Chapter 27**, we reach outside the program itself — to the **system and OS**.

Chapter 27 - System and OS Access

A program doesn't run in a vacuum — it runs on a computer, started by a user, with an operating system around it. Sometimes you need to reach out to that world: find out what OS you're on, read an environment variable, pass in arguments, or run another program. Aussom splits this across two modules. **sys** reports **information** about the system, and **os** lets you **interact** with it. This chapter closes Part VIII with both.

Figure 71: A program and the world around it

System Information with sys

The **sys** module (`include sys;`) answers questions about the machine and the runtime your program is running on. You already saw one of its calls back in Chapter 18. Each is a simple getter that returns a string:

```
include sys;

sys.getOsName()          // "Linux"   - the operating system
sys.getOsArch()          // "amd64"   - the processor architecture
sys.getUserName()        // "austin"  - who's running the program
sys.getHomePath()        // "/home/austin" - their home folder
sys.getAussomVersion()   // "1.2.10"  - the Aussom version
```

These are handy for logging (recording *where* a program ran), for adapting behavior to the platform, or just for a diagnostic "about" screen. The fuller set includes:

- **`getOsName`** / **`getOsArch`** / **`getOsVersion`** — details of the operating system.
- **`getUserName`** / **`getHomePath`** — who's running it and their home folder.
- **`getAussomVersion`** / **`getJavaVersion`** — the language and Java runtime versions.
- **`getMills`** — the current time in milliseconds; **`sleep(millis)`** — pause the program for a while.

Environment Variables, Arguments, and Exit Codes

These three are how a program **exchanges** small bits of information with whoever starts it. All the interaction lives in the **`os`** module (`include os;`).

Environment variables are named values the operating system hands to every program — things like `HOME` or `PATH`. Read one with **`os.getEnv`**, giving an optional fallback for when it isn't set:

```
include os;

os.getEnv("HOME")                    // "/home/austin"
os.getEnv("API_KEY", "(not set)")    // the fallback, if API_KEY isn't
                                     // defined
```

The related calls are **`os.hasEnv`** (is it set?), **`os.setEnv`** (set one for this program), and **`os.getEnvs`** (get them all as a map).

Arguments are the words typed after your program's name on the command line. They arrive through the program's entry point, **`main(args)`** — the method Aussom runs first in a classical-mode program (a class with a `main`). `args` is a **list** of those words:

```
class Greet {
    public main(args) {
        c.log("got " + #args + " argument(s)");
        c.log("first: " + args[0]);
    }
}
```

Run as `aussom greet.aus World`, that prints `got 1 argument(s)` and `first: World`.

An **exit code** is a single number a program leaves behind when it finishes, telling whoever ran it whether things went well — **0 means success**, and anything else signals a problem. Set it with **`os.exit(code)`**, which stops the program immediately:

```
if (#args == 0) {
    os.outErr("usage: greet <name>");   // a message to standard error
    os.exit(1);                          // stop now, with a failure
                                         // code
}
```

A program that simply runs to the end exits with `0` on its own; you only call `os.exit` when you want to stop early or report a specific failure code.

Running External Commands with os

Your program can also run **other programs** — a shell command, a system tool — and read what they produce. **`os.exec`** takes a command line, runs it, and returns a **map** of results:

```
res = os.exec("echo hello from the shell");
c.log(res["result"]);      // "hello from the shell\n" - what the
                           // command printed
c.log(res["exitCode"]);    // 0  - the command's own exit code
c.log(res["success"]);     // true
```

The result map has three keys: **`result`** (the command's output text), **`exitCode`** (its exit code), and **`success`** (`true` if it ran without error). A couple of related calls round it out:

- **`os.execRaw(list)`** — run a command from a **list** of parts (`["ls", "-l", "/tmp"]`) instead of one string, which avoids surprises when arguments contain spaces.
- **`os.getOsType`** — a simple **`"windows"`**, **`"mac"`**, or **`"linux"`** so you can branch on the platform.

Try It Yourself

The first program reports system info, reads the environment, and runs a command. Save it as `system.auss` and run it:

```
include sys;
include os;

c.log("OS:       " + sys.getOsName() + " (" + sys.getOsArch() + ")");
c.log("user:     " + sys.getUserName());
c.log("home:     " + sys.getHomePath());
c.log("aussom:   " + sys.getAussomVersion());

c.log("os type:  " + os.getOsType());
c.log("HOME:     " + os.getEnv("HOME"));
c.log("missing:  " + os.getEnv("NOT_SET", "(default)"));

res = os.exec("echo hello from the shell");
c.log("output:   " + res["result"].trim());
c.log("exit:     " + res["exitCode"] + ", ok? " + res["success"]);
```

Running it prints (your details will differ):

The second program, `greet.aus`, uses arguments and an exit code. Because it reads `args`, it's a classical-mode program with a `main`:

```
include os;
```

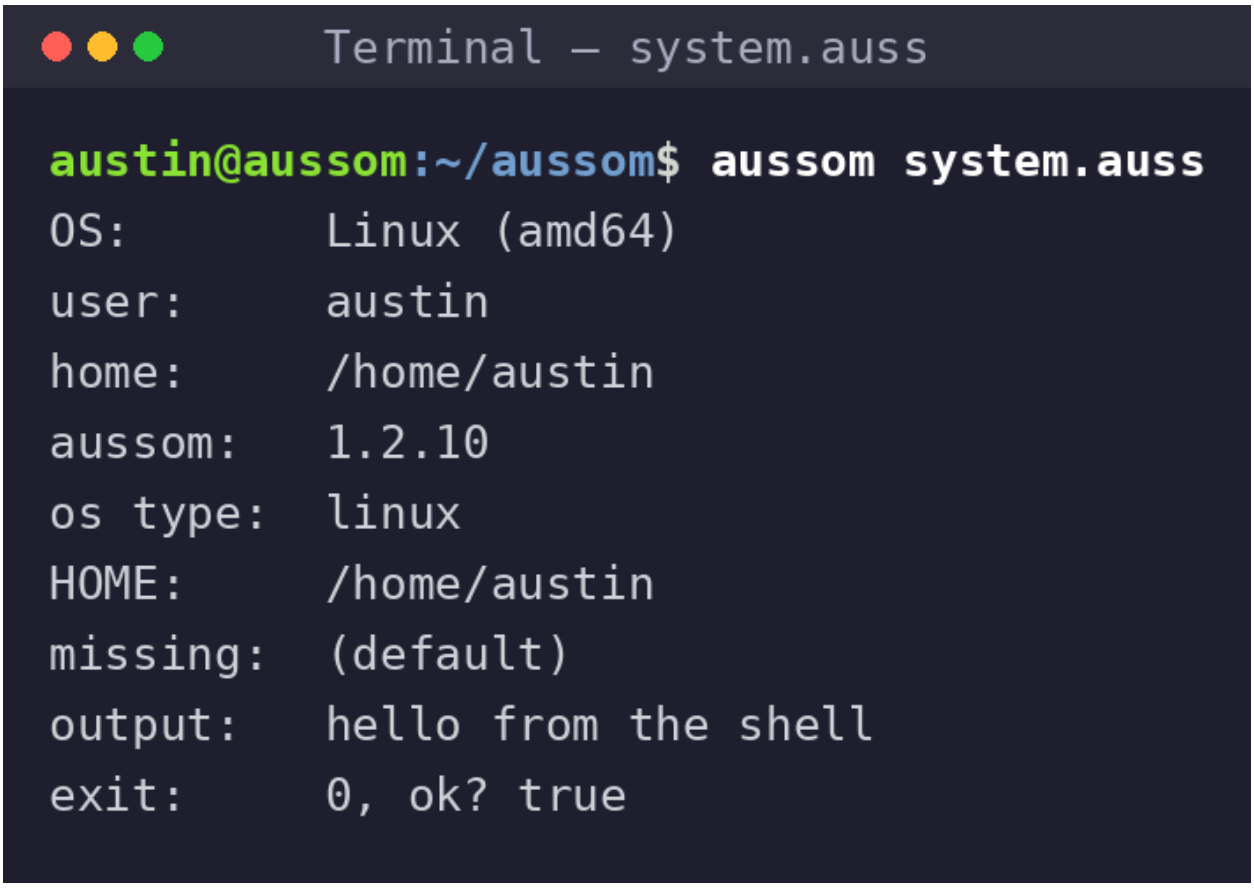


Figure 72: Running system.auss

```
class Greet {
    public main(args) {
        if (#args == 0) {
            os.outErr("usage: greet <name>");   // to standard error
            os.exit(1);                          // stop with a non-zero
                                                 // code
        }
        c.log("Hello, " + args[0] + "!");
    }
}
```

Run it with and without an argument (`echo $?` shows the exit code afterward):

Figure 73: Running greet.aus

You can find both examples here: `27-system`.

What You Learned

- **`sys`** (with `include sys;`) reports **system information** — **`getOsName`**, **`getOsArch`**, **`getUserName`**, **`getHomePath`**, **`getAussomVersion`**, and more.
- **`os`** (with `include os;`) is how a program **interacts** with its surroundings: **`getEnv`** / **`setEnv`** for environment variables, **`main(args)`** for command-line arguments, and **`os.exit(code)`** to finish with an exit code (`0` = success).
- **`os.exec(command)`** runs another program and returns a map with **`result`**, **`exitCode`**, and **`success`**; **`os.getOsType`** tells you the platform.

That completes **Part VIII** and your tour of the standard library. Next, in **Part IX**, we go further — starting with **Chapter 28, Script Mode**.

Part IX

Going Further

Chapter 28 - Script Mode

Welcome to **Part IX**, where we go further — putting Aussom to work on real, practical jobs. We start with something you've actually been using this whole book without a name for it: **script mode**. Almost every example you've run has been a `.auss` file full of top-to-bottom statements, with no `class` and no `main`. That's script mode, and this chapter finally explains what's been going on and how to get the most out of it.

Writing Scripts Without a Class

Back in Chapter 1 you learned the "proper" shape of an Aussom program: a **class** with a **`main`** method that the interpreter runs first. That's **classical mode**, and it's the default for `.aus` files. But there's a second, lighter way to run code — **script mode** — where the file is just a list of statements that run **top to bottom**, exactly like a shell or Python script. No class, no `main`, no ceremony:

```
c.log("Hello!");     // that's a complete script
```

Figure 74: The same program, two ways to write it

The two modes run the *same language* — the same types, operators, control flow, and standard library. The only difference is the starting shape. In script mode you can put almost anything at the top level:

- **assignments and expressions** — `x = 5;`, `c.log(x);`
- **control flow** — `if`, `while`, `for`, `try/catch`, `switch`
- **`include` directives** — `include math;`
- **class and enum declarations** — you can still define a `class` and use it lower down

There's **one thing you can't do** at the top level: define a **standalone function**. Aussom has no free-floating functions (you learned back in Chapter 14 that functions live inside classes), so if a script needs a reusable helper, put it in a class:

```
include math;

class Circle {
    public r = 0;
    public Circle(Radius) { this.r = Radius; }
    public area() { return math.pi() * this.r * this.r; }
}

circle = new Circle(3.0);          // top-level code using the class
                                   // above
c.log("area = " + circle.area());
```

This is a normal script — top-level statements — that happens to declare one helper class along the way. Handy when a script needs a type or two but doesn't deserve to be split into separate files.

Running Scripts: `-s`, the `.auss` Extension, and Shebangs

How does Aussom know to run a file in script mode instead of looking for a `main`? It checks for **any one** of three signals:

- **The `.auss` extension.** `aussom hello.auss` runs in script mode automatically. This is the recommended way — the extra `s` marks the file as a script at a glance.
- **The `-s` flag.** `aussom -s hello.aus` forces script mode no matter the extension.
- **A shebang line.** If the first line starts with `#!`, Aussom switches to script mode and strips that line before running.

If none of those is present, the file runs in classical mode — and a file full of top-level statements will fail to parse, with an error telling you top-level statements aren't allowed there. (That's your clue you meant to run it as a script.)

The **shebang** is what lets you run a script *directly*, like any other command-line tool, without typing `aussom` in front. Put this as the very first line, mark the file executable with `chmod +x`, and you can run `./greet.auss` on Linux or macOS:

```
#!/usr/bin/env -S aussom -s
c.log("run me directly!");
```

The `#!/usr/bin/env -S aussom -s` form is the portable choice on modern systems. (On an older system, `#!/usr/bin/env aussom` also works — the `#!` line alone is enough to turn on script mode.)

Arguments and Exit Codes in Scripts

A script talks to whoever runs it the same way any command-line tool does: through **arguments** coming in and an **exit code** going out.

In Chapter 27 you saw a classical program receive its arguments through `main(args)`. A script gets the very same list, but as a ready-made top-level variable simply named **`args`** — no `main` required:

```
c.log("got " + #args + " argument(s)");
for (a : args) {
    c.log("- " + a);
}
```

Run as `aussom greet.auss Ada Grace`, `args` holds `["Ada", "Grace"]` — the words typed after the filename, each a string (quoted arguments with spaces arrive intact).

The **exit code** — the success/failure number a program leaves behind — works just as it did in Chapter 27. A script that runs to the end exits with **0** (success). If an uncaught exception stops it, the exit code is **1**. And you can set one yourself with **`os.exit(code)`** to stop early and report a specific result:

```
include os;
if (#args == 0) {
    os.outErr("usage: greet.auss <name>");   // to standard error
    os.exit(1);                              // stop now, signal failure
}
```

When to Use Script Mode vs. Classical Mode

Both modes run identical code, so the choice is about *fit*:

- **Reach for script mode** (`.auss`) for small, direct jobs — a quick file-processing task, a build step, glue between shell commands, or anything you'd otherwise write as a Bash or Python script. Less ceremony, faster to write, easy to run with `./script.auss`.
- **Reach for classical mode** (`.aus`) for larger programs — anything that grows into several classes and files, or that you want to run with Aussom's testing (`-t`) and documentation (`-d`) tools, which expect a class-based file. (Script mode can't be combined with those flags.)

A good rule of thumb: start a quick idea as a `.auss` script, and if it grows into a real application, graduate it to a class-based `.aus` structure.

Try It Yourself

This script greets everyone named on the command line. It shows top-level code, `args`, an exit code, and a shebang so you can run it directly. Save it as `greet.auss`:

```
#!/usr/bin/env -S aussom -s
include os;

```

```
if (#args == 0) {
    os.outErr("usage: greet.auss <name> [name ...]");
    os.exit(1);
}

for (name : args) {
    c.log("Hello, " + name + "!");
}
c.log("Greeted " + #args + " name(s).");
```

Run it with `aussom`, directly after `chmod +x greet.auss`, and with no arguments (`echo $?` shows the exit code):

Figure 75: Running greet.auss three ways

You can find this example here: `28-scripts`.

What You Learned

- **Script mode** runs a file's statements **top to bottom** — no `class` or `main` needed. It's the same language as classical mode; only the starting shape differs. The one rule: no standalone functions at the top level (put helpers in a class).
- Aussom runs a file in script mode when it sees any of: the **`.auss`** extension, the **`-s`** flag, or a **`#!` shebang** line. A shebang lets you run a script directly as `./script.auss`.
- A script reads its command-line **`args`** as a top-level list, and reports an **exit code** (0 on success, 1 on an uncaught error, or whatever you pass to **`os.exit`**).
- Use **script mode** for small, direct jobs and **classical mode** for larger programs and anything

using the test or doc tools.

Next, in **Chapter 29**, we turn a script into a real **command-line tool** with proper options and help text.

Chapter 29 - Building Command-Line Tools

The script in Chapter 28 read its arguments straight out of the `args` list. That's fine for one or two, but real command-line tools accept **options** — named switches like `--name Ada`, `--count 3`, or `--verbose` — that can appear in any order, in short or long form. Parsing all that by hand is fiddly and easy to get wrong. Aussom's **`cliparser`** module does it for you, and even prints a polished help screen. This chapter uses it to build a small tool from start to finish.

Bring the module in with:

```
include cliparser;
```

Parsing Arguments with `cliparser`

A command line is made of a few different kinds of piece, and `cliparser`'s job is to sort them out:

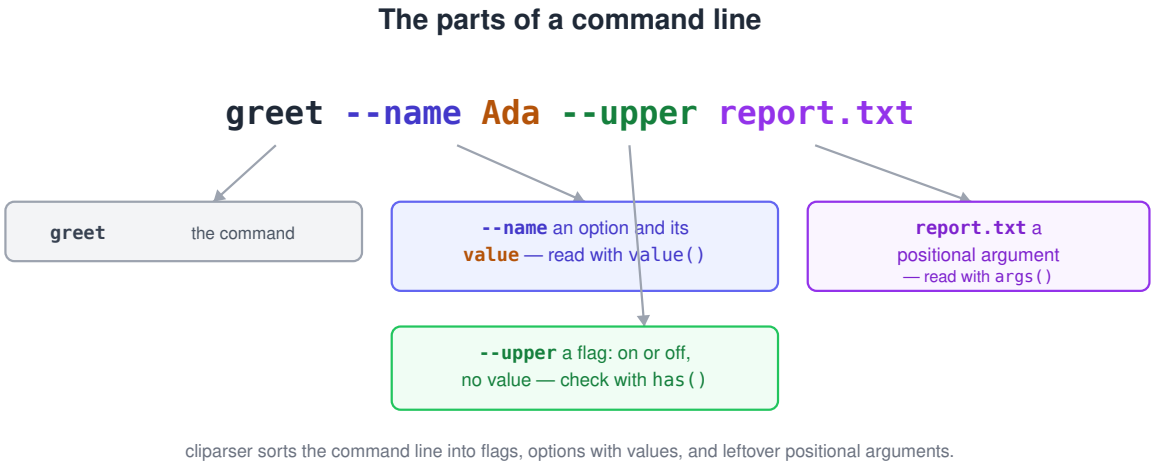


Figure 76: The parts of a command line

- **Options with values** (`--name Ada`) — a named setting and the value that follows it.
- **Flags** (`--upper`) — an on/off switch with no value; either it's there or it isn't.
- **Positional arguments** (`report.txt`) — the leftover words that aren't options.

Working with `cliparser` is a three-step routine. First, **declare** the options your tool accepts. Second, **parse** the `args`. Third, **read** the results:

```
// 1. declare
opts = new CliOptions();
opts.value("n", "name", "Name to greet");        // takes a value: -n /
                                                 // --name
opts.flag("u", "upper", "Shout in uppercase");   // a flag: -u / --upper

// 2. parse
parser = new CliParser();
parser.options(opts);
result = parser.parse(args);

// 3. read
result.has("upper")            // true if -u/--upper was given
result.value("name")           // the value after -n/--name (or null)
result.value("count", "1")     // ...with a fallback when it's missing
result.args()                  // the leftover positional arguments, as
                               // a list
```

Each option gets **two names** — a short one (`n`, used as `-n`) and a long one (`name`, used as `--name`) — plus a **description** for the help screen. You can query the result by either name. **`value`** reads an option's value (with an optional default), **`has`** tests whether a flag or option was present, and **`args`** returns the positional leftovers.

Options, Flags, and Help Text

The two kinds of option come from two methods:

- **`opts.flag(short, long, description)`** — a switch with no value. `result.has("upper")` tells you whether it appeared.
- **`opts.value(short, long, description)`** — an option that takes a value. `result.value("name")` gives it back.

The real payoff is **help text**, which `cliparser` builds for you from those same descriptions — no need to write and maintain a usage message by hand. Call **`parser.help(usage)`**, passing the one-line usage summary, and it prints a neatly aligned screen:

```
parser.help("greet -n <name> [options]");
```

That produces output like:

```
usage: greet -n <name> [options] [-c <arg>] [-h] [-n <arg>] [-u]
  -c,--count <arg>    How many times to greet (default 1)
  -h,--help           Show this help and exit
  -n,--name <arg>     Name to greet
  -u,--upper          Shout the greeting in uppercase
```

Two practical notes on robustness. Reading the command line can fail — an unknown option, or a

value option with no value — so wrap **`parse`** in a `try/catch` and show help if it throws. And when your tool *needs* a certain option, check for it yourself with **`has`** and print a clear message; that keeps you in control of exactly what the user sees.

> **A more automatic option.** `cliparser` can also mark an option **required** (a fourth `true` argument to `value`) and handle failures for you with **`parseOrExit(args, usage)`**, which prints an error plus the help screen and returns `null` when parsing fails. It's less code, but the manual `has` check above lets you word the error message yourself — which is why we use it here.

A Small CLI Tool from Start to Finish

Putting it together, here's a complete `greet` tool. It accepts a name, an optional repeat count, an uppercase flag, and a help flag — and handles the missing-name and help cases cleanly:

```
// greet.auss - a small command-line tool built with cliparser
include cliparser;
include os;

// 1. Declare the options.
opts = new CliOptions();
opts.value("n", "name", "Name to greet");
opts.value("c", "count", "How many times to greet (default 1)");
opts.flag("u", "upper", "Shout the greeting in uppercase");
opts.flag("h", "help", "Show this help and exit");

// 2. Parse the command line.
parser = new CliParser();
parser.options(opts);
usage = "greet -n <name> [options]";

result = null;
try {
    result = parser.parse(args);
} catch (e) {
    c.err("could not read the options");
    parser.help(usage);
    os.exit(2);
}

// 3. Handle help and required input.
if (result.has("help")) {
    parser.help(usage);
    os.exit(0);
}
if (!result.has("name")) {
    c.err("the --name option is required");
    parser.help(usage);
```

```
    os.exit(2);
}

// 4. Do the work.
name = result.value("name");
count = result.value("count", "1").parseInt();
greeting = "Hello, " + name + "!";
if (result.has("upper")) { greeting = greeting.toUpper(); }
for (i = 0; i < count; i++) {
    c.log(greeting);
}
```

Read it top to bottom and you can see the three-step routine in action: declare the four options, parse `args` (guarding against a parse error), then branch on `has` and `value` to do the work. The `--count` value arrives as a string, so `parseInt()` turns it into a number for the loop.

Try It Yourself

Save the program above as `greet.auss` and run it a few different ways:

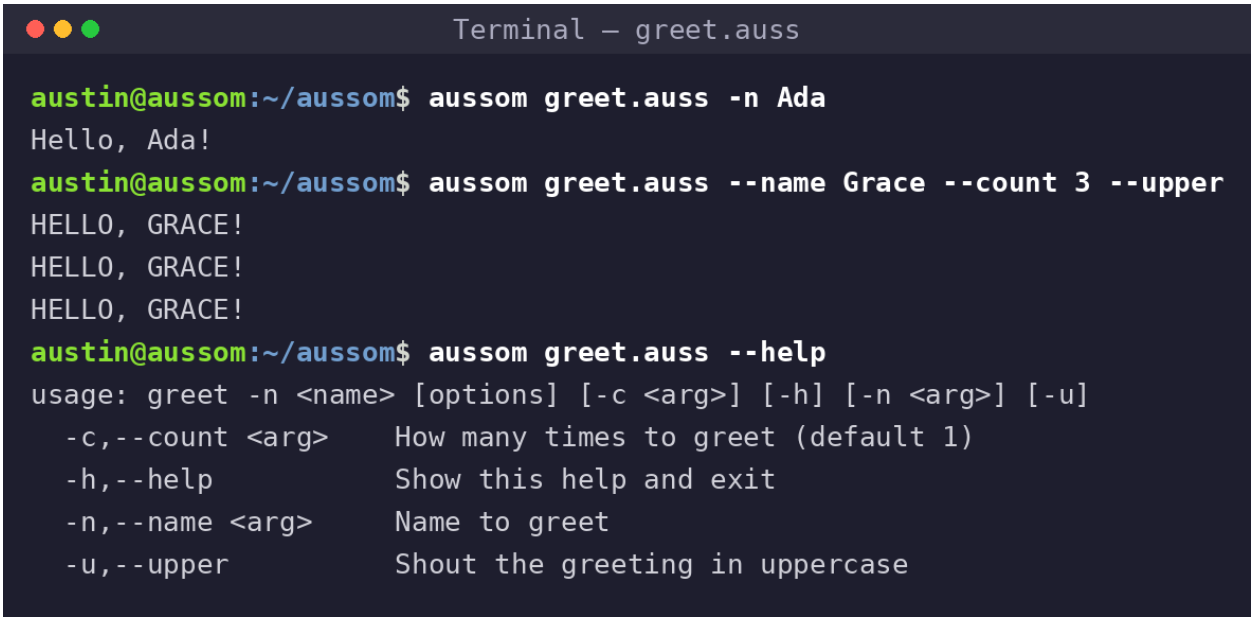


Figure 77: Running greet.auss with several option combinations

The tool takes options in short (`-n`) or long (`--name`) form, in any order; treats `--upper` as an on/off flag; falls back to a count of 1; and prints its own help screen on `--help` or when `--name` is missing. You can find this example here: `29-cli-tools`.

What You Learned

- **`cliparser`** (with `include cliparser;`) parses command-line options so you don't have to pick apart `args` by hand. The routine is: **declare** options, **parse**, then **read** the result.
- **`opts.value(short, long, desc)`** declares an option that takes a value; **`opts.flag(short, long, desc)`** declares an on/off flag. Each has a short and long name and a description.
- From the result, **`has(name)`** tests presence, **`value(name, default)`** reads a value, and **`args()`** returns the positional leftovers.
- **`parser.help(usage)`** prints an aligned help screen built from your option descriptions. Wrap **`parse`** in a `try`/`catch`, and validate required options yourself with **`has`** for clear error messages.

Next, in **Chapter 30**, we make terminal programs **colorful and interactive** with `jansi` and `curses`.

Chapter 30 - Colorful Terminals and Text UIs

So far your programs have printed plain, one-color text. But the terminal can do much more — **colored** and **styled** text, output placed anywhere on the screen, even full **interactive interfaces** with forms and menus, all without leaving the console. Aussom ships two modules for this: **`jansi`** for color and cursor control, and **`curses`** for building complete text-based user interfaces (TUIs). Both come with the CLI.

> **Run these in a real terminal.** The programs in this chapter draw directly to your terminal — colors, cursor moves, full-screen UIs. Run them in an actual terminal window (not piped into a file or an editor's output pane) to see them work.

Styling Output with `jansi`

`jansi` adds **ANSI styling** — the color and formatting codes terminals understand. Start by including it and calling **`systemInstall()`** once, which turns on ANSI handling (and makes it work on Windows too):

```
include jansi;
jansi.systemInstall();
```

You build styled text with an **`ansi`** builder. Create one with `new ansi()`, then **chain** calls: a color or style method turns an effect *on*, **`a(text)`** appends text, and **`reset()`** clears the styling back to normal. Finally, hand the builder to **`jansi.println`** to print it:

```
a = new ansi();
a.fgRed().a("red ").fgGreen().a("green ").fgBlue().a("blue").reset();
jansi.println(a);
```

Each `fg...` call sets the **foreground** (text) color; there's a matching **`bg...`** for the background. The style methods work the same way — turn one on, append text, reset:

```
b = new ansi();
b.bold().a("bold text").reset().a("   ").underline().a("underlined
↪   text").reset();
jansi.println(b);
```

The full set you'll reach for most:

- **Colors:** `fgRed`, `fgGreen`, `fgYellow`, `fgBlue`, `fgCyan`, `fgMagenta`, `fgWhite`, `fgBlack` (and a `bg...` version of each for backgrounds).
- **Styles:** `bold`, `underline`, `faint` (dim), and `reset` to clear them all.

There's also a compact **markup** form via **`render`**, where `@|style text|@` applies styles to a span — handy for a quick colored line:

```
e = new ansi();
e.render("@|red,bold Error:|@ @|green done|@ - finished");
jansi.println(e);
```

Here's a program pulling these together:

Figure 78: Running colors.auss

When you're finished with styled output, call **`jansi.systemUninstall()`** to turn ANSI handling back off.

Cursor Control and Clearing the Screen

Besides color, `jansi` can **move the cursor** and **clear the screen** — the building blocks of output that updates in place (progress bars, live dashboards) rather than just scrolling. Two calls do most of the work:

- **`eraseScreen()`** — wipe the whole screen clean.
- **`cursor(row, col)`** — jump the cursor to a specific row and column, so the next text lands exactly there.

With those you can paint a layout at fixed positions instead of printing line by line. This little status board clears the screen, then places each line where it wants it:

```
a = new ansi();
a.eraseScreen();
a.cursor(2, 3).fgCyan().bold().a("System Status").reset();
```

```
a.cursor(4, 5).fgGreen().a("[ OK ] ").reset().a("Database");
a.cursor(6, 5).fgRed().bold().a("[DOWN] ").reset().a("Mailer");
jansi.println(a);
```

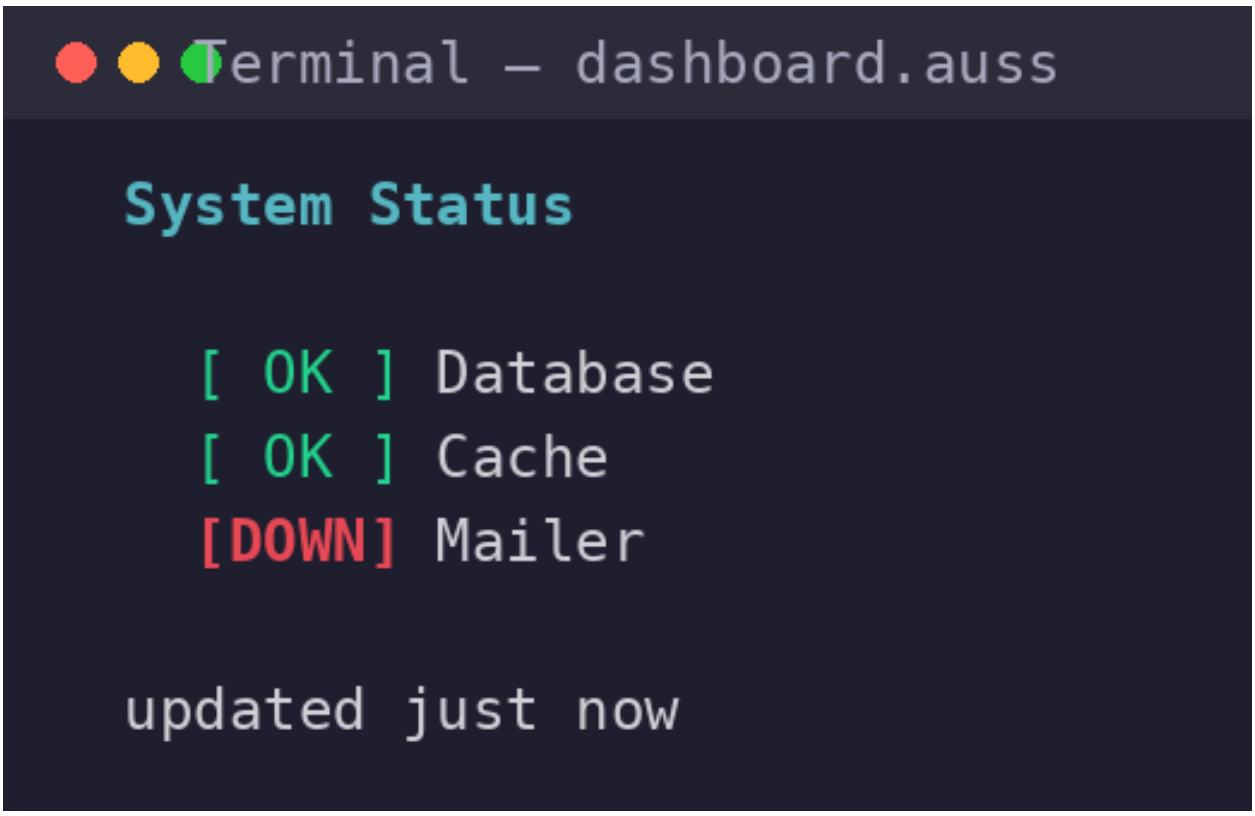


Figure 79: Running dashboard.auss

For relative moves there are **`cursorUp`**, **`cursorDown`**, **`cursorLeft`**, and **`cursorRight`** (each takes a number of cells), plus **`eraseLine`** to clear just the current line and **`saveCursorPosition`** / **`restoreCursorPosition`** to bookmark a spot. Moving the cursor up and overwriting a line is exactly how a program shows a progress bar that fills in place.

Building a Text UI with `curses`

For a real interface — one with input fields, dropdowns, and buttons the user tabs between — reach for **`curses`**. It builds a full-screen **text user interface** out of ready-made components, and handles the keyboard and redrawing for you.

A curses UI is a set of **nested pieces**, each sitting inside the one before it:

- A **`screen`** is the whole terminal. A **`gui`** draws on it and runs the event loop.
- A **`win`** (window) holds your content and an optional menu bar.
- A **`panel`** arranges **components** — labels, text boxes, buttons — using a **layout**.

You build from the **inside out**: create the screen and gui, fill a panel with components, put the panel in a window, then hand the window to the gui to display. First, though, let's meet the controls that

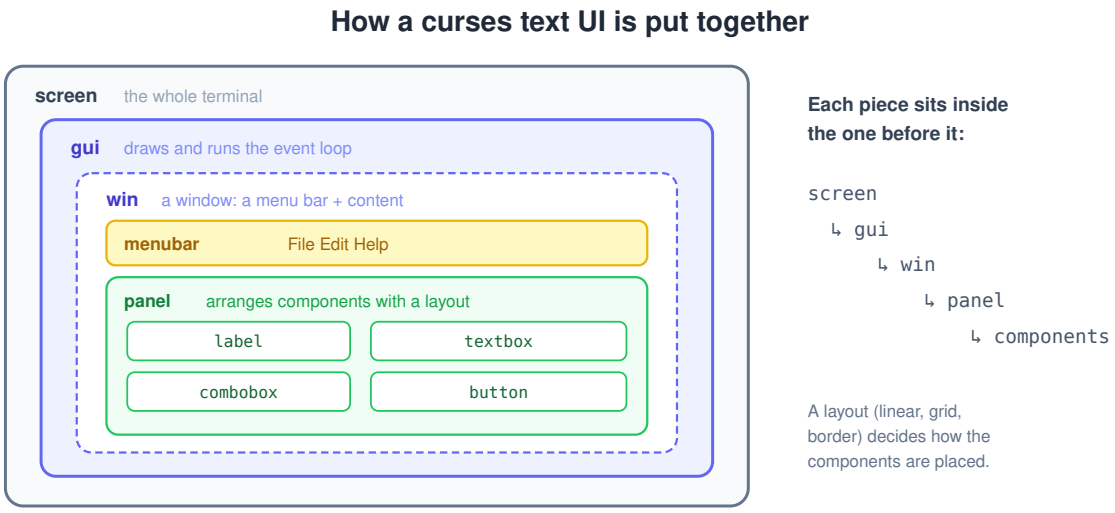


Figure 80: How a curses text UI is put together

go in the panel.

The Core Controls

Most text UIs are built from a small set of controls. Here are the ones you'll reach for most, and how to create each.

Text input is the workhorse. The **`textbox`** is where the user types. By default it's a **multi-line** box — a text *area* — so pass **`false`** as the second argument when you want a **single-line** input field instead:

```
// single-line input field
name  = new textbox("Ada Lovelace", false);
// multi-line text area: 30 columns x 3 rows
notes = new textbox("", true, 30, 3);
```

A textbox can even guard what's typed: **`setValidation(pattern)`** rejects input that doesn't match a regular expression (Chapter 24), and **`setReadOnly(true)`** shows text the user can't change.

Choice controls let the user pick from a list, in three flavors:

- **`combobox`** — a **dropdown**; the user opens it and picks **one** item: `new combobox(["Denver", "London", "Paris"])`.
- **`checkbox`** — a list where **any number** can be ticked: `new checkbox(["Newsletter", "Product news"])`.
- **`radiobox`** — a list where exactly **one** is selected at a time: `new radiobox(["Free", "Pro", "Team"])`.

Actions and text round it out: a **`button`** runs a callback when pressed (`new button("Save", ::onSave)`), a **`label`** is static text, and a **`table`** shows rows of data. Finally, a **layout** decides how the panel arranges everything — **`linearlayout`** stacks components top to bottom, **`gridlayout`**

places them in a grid, and **`borderlayout`** pins them to the edges and center.

Assembling a form is just adding controls to a panel, then wrapping it in a window:

```
p = new panel();
p.setLayout(new linearlayout());        // stack components top to
                                        // bottom
p.add(new label("Name:"));
p.add(name);                            // the single-line field
p.add(new label("City:"));
p.add(city);                            // a combobox
p.add(new label("Notes:"));
p.add(notes);                           // the multi-line text area
p.add(subs);                            // a checkbox
p.add(new button("Save", ::onSave));

w = new win();
w.set(p);
g.addWindowAndWait(w);                  // draw it and run until the
                                        // window closes
```

`addWindowAndWait` shows the window and takes over — drawing the UI and handling the keyboard, letting the user **Tab** between controls and edit them — until the window closes.

Reading What the User Entered

A form is only useful if you can get the values back out. Every input control has a **getter** you call — usually inside a button's callback, after the user has filled things in:

- **`textbox.getText()`** — the typed text.
- **`combobox.getSelectedItem()`** — the chosen item (or **`getSelectedIndex()`** for its position).
- **`checkbox.getCheckedItems()`** — a **list** of the ticked items.
- **`radiobox.getSelected()`** — the single selected item.

To reach the controls from a callback, the form keeps each one as a member (`this.name`, `this.city`, ...). The **Save** button's callback then reads them and acts — here, it builds a summary and shows it in a dialog:

```
public onSave() {
    summary = "Name:  " + this.name.getText() + "\n"
            + "City:  " + this.city.getSelectedItem() + "\n"
            + "Lists: " + this.subs.getCheckedItems();
    dialog.show(this.g, "Saved", summary, dlgbtn.ok);
}
```

Menus, Dialogs, and Keyboard Input

Real UIs need **menus** and pop-up **dialogs** too. A **menu bar** goes across the top of a window. Build one from a **`menubar`** with **`menu`** items, giving each item a callback:

```
mbar = new menubar();
fmenu = new menu("File");
fmenu.addItem("Save", ::onMenuSave);
fmenu.addItem("Exit", ::onExit);        // onExit calls g.stop() to
                                        // close the UI
mbar.add(fmenu);
w.setMenuBar(mbar);
```

With the menu bar added, the window looks like this:

A **dialog** is a modal pop-up — a message the user must acknowledge before continuing. **`dialog.show`** displays one, given the gui, a title, a message, and which button to show:

```
dialog.show(g, "Saved", "Your contact was saved.", dlgbtn.ok);
```

There are richer dialogs too — **`inputdialog`** (ask for a line of text), **`filedialog`** and **`directorydialog`** (pick a file or folder), and **`actiondialog`** (offer a set of actions). For **keyboard input** at a lower level, the gui's **`getKey()`** waits for and returns a single keypress — useful when you want to handle keys yourself rather than let components do it. And whenever you need to close the UI, call **`g.stop()`**.

Try It Yourself

Save the four programs and run each **in a real terminal**:

- **`colors.auss`** — colored and styled text (shown above).
- **`dashboard.auss`** — a status board placed with cursor control (shown above).
- **`form.aus`** — the interactive contact form with a menu (shown above). Tab between the single-line name field, the city dropdown, the multi-line notes area, and the checkboxes; then press **Save** to see the values you entered read back in a dialog.
- **`dialog.aus`** — a stand-alone message dialog (shown above).

The `form.aus` and `dialog.aus` programs are classical-mode `.aus` files because their buttons and menus need callback methods. You can find all four here: `30-terminal-ui`.

What You Learned

- **`jansi`** styles terminal text. Call **`systemInstall()`**, build with **`new ansi()`** by chaining color/style methods (**`fgRed`**, **`bold`**, **`underline`**, ...) and **`a(text)`**, then **`jansi.println`**. **`reset()`** clears styling; **`render`** offers a compact `@|style text|@` markup.
- `jansi` also does **cursor control** — **`eraseScreen`**, **`cursor(row, col)`**, and the relative moves — for output that updates in place.
- **`curses`** builds a full **text UI** from nested pieces: **`screen`** → **`gui`** → **`win`** → **`panel`** (with a **layout**) → **components**. **`addWindowAndWait`** runs it.

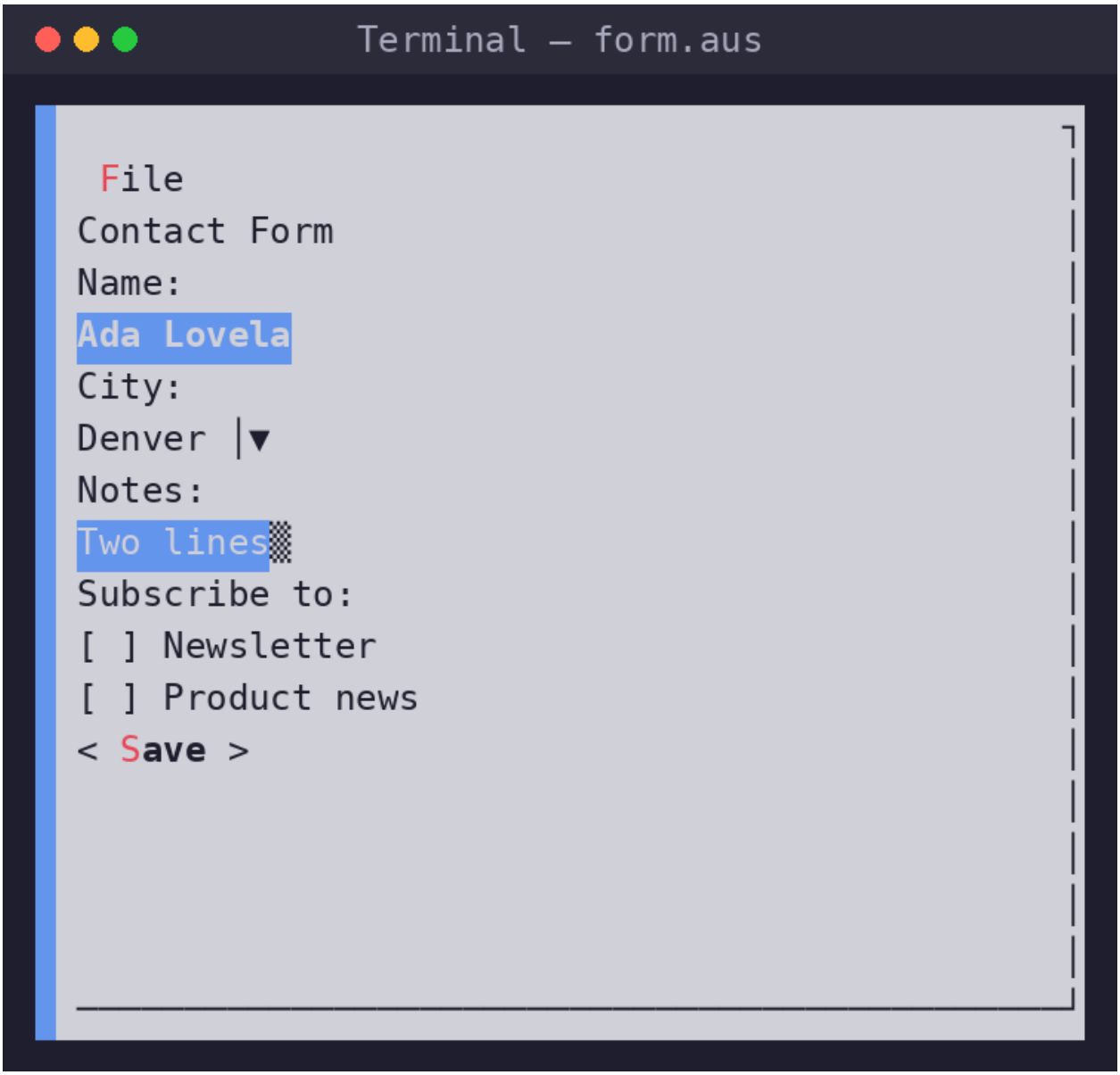


Figure 81: Running form.aus

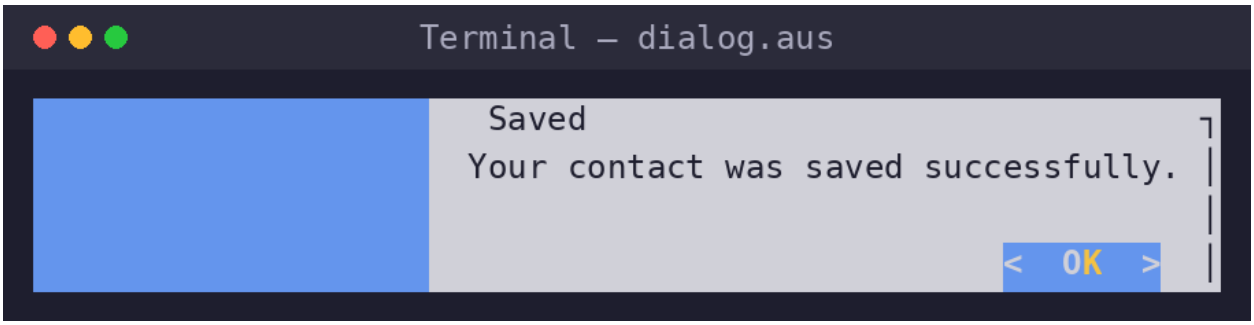


Figure 82: Running dialog.aus

- The core controls: **`textbox`** — a **single-line field** (`new textbox("", false)`) or a **multi-line area** (the default) — plus **`combobox`** (pick one), **`checkbox`** (pick many), **`radiobox`** (one of several), **`button`**, **`label`**, and **`table`**.
- Read the entered values with each control's getter: **`textbox.getText()`**, **`combobox.getSelectedItem()`**, **`checkbox.getCheckedItems()`**, **`radiobox.getSelected()`** — usually from a button's callback.
- Add a **`menubar`** of **`menu`** items for menus, show pop-ups with **`dialog.show`** (and `inputdialog`, `filedialog`, ...), read keys with **`getKey()`**, and close with **`g.stop()`**.

Next, in **Chapter 31**, we reach across the network to **talk to the web**.

Chapter 31 - Talking to the Web

A huge amount of what programs do today involves the **web**: fetching data from an API, posting a form, checking a service. Underneath it all is **HTTP**, the simple request-and-response protocol browsers and servers speak. Aussom's **`http`** module lets your program speak it too — send a request, get a response back. This chapter connects to a real public API to show how.

```
include http;
```

> **These examples need internet access.** They talk to a live test API (`jsonplaceholder.typicode.com`), so run them on a connected machine.

Making HTTP Requests

Every HTTP conversation is a **request** you send and a **response** you get back:

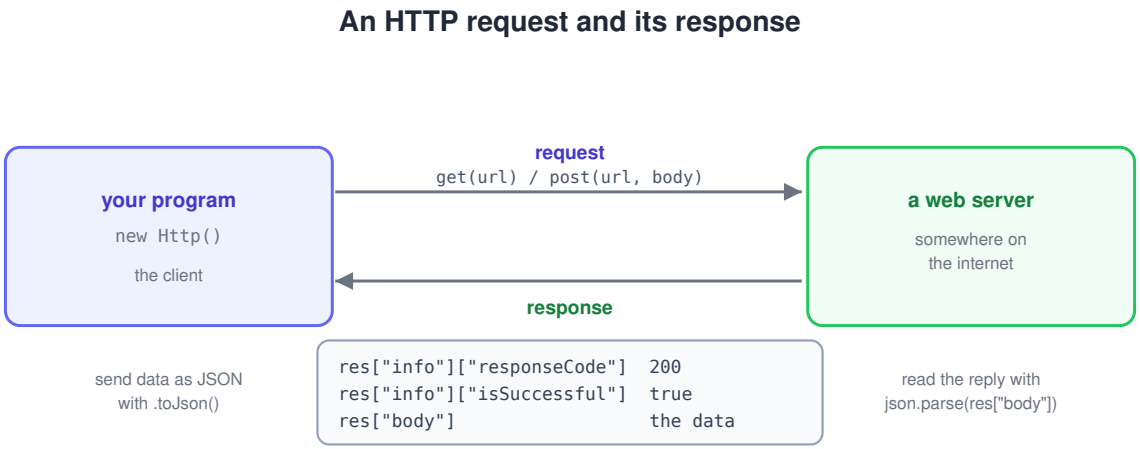


Figure 83: An HTTP request and its response

You make requests with an **`Http`** object. Create one with `new Http()`, then call a method named after the HTTP **verb** you want. The simplest is **`get`**, which fetches a resource at a URL:

```
h = new Http();
res = h.get("https://jsonplaceholder.typicode.com/todos/1");
```

The **response** comes back as a map with everything the server sent. The two parts you'll use most are the **body** and the **status code**:

```
res["body"]                      // the response content, as a string
res["info"]["responseCode"]      // the HTTP status number, e.g. 200
res["info"]["isSuccessful"]      // true when the status is a success
                                 // (2xx)
```

The `info` map also carries the response **`headers`** and a few other details. The four other verbs work the same way, each returning the same kind of response:

- **`get(url)`** — fetch a resource.
- **`post(url, body)`** — create something, sending a body.
- **`put(url, body)`** / **`patch(url, body)`** — replace or update a resource.
- **`delete(url)`** — remove a resource.

Each method also takes an optional **headers** map as a final argument — for example `h.get(url, {"X-API-KEY": "abc123"})` — to send things like authentication tokens.

Sending and Receiving JSON

Most web APIs speak **JSON**, and you already know how to handle it from Chapter 21. The body of a response is JSON **text**, so parse it into Aussom data with **`json.parse`**:

```
res = h.get("https://jsonplaceholder.typicode.com/todos/1");
todo = json.parse(res["body"]);          // JSON text -> a map
c.log(todo["title"]);                    // "delectus aut autem"
```

Sending JSON is the mirror image: build a map, turn it into JSON text with **`toJson`**, and pass it as the body of a **`post`**. `post` defaults to a JSON content type, so there's nothing else to set:

```
payload = {"title": "Learn Aussom", "completed": false}.toJson();
created = h.post("https://jsonplaceholder.typicode.com/posts", payload);
newId = json.parse(created["body"])["id"];    // read the server's reply
```

So the round trip is: `toJson` on the way **out**, `json.parse` on the way **back** — the same two calls from Chapter 21, now carrying data across the network.

Handling Responses and Errors

Two very different things can go wrong with a web request, and they're handled in two different ways.

The first is an **HTTP error status** — the server answered, but with a code like `404` (not found) or `500` (server error). This is **not** an exception; you get a normal response back, and you check whether it succeeded:

```
res = h.get("https://jsonplaceholder.typicode.com/todos/99999999");
if (!res["info"]["isSuccessful"]) {
    // prints, e.g., "lookup failed with status 404"
```

```
    c.log("lookup failed with status " + res["info"]["responseCode"]);
}
```

The second is a **connection failure** — the server couldn't be reached at all (no network, bad hostname, timeout). *That* does throw an exception, so wrap network calls in a **`try/catch`** when a failure is possible:

```
try {
    res = h.get("https://no-such-host.invalid/");
} catch (e) {
    c.log("could not connect to the server");
}
```

The rule of thumb: **check `isSuccessful` for the server's answer, and `try/catch` for the connection itself.** A robust request does both.

Try It Yourself

This program does a GET, a POST, an error-status check, and a connection-error catch — a tour of a real HTTP session. Save it as `web.auss` and run it (you'll need internet):

```
include http;

h = new Http();

// GET: fetch a resource
res = h.get("https://jsonplaceholder.typicode.com/todos/1");
c.log("GET status: " + res["info"]["responseCode"]);
todo = json.parse(res["body"]);
c.log("todo: " + todo["title"] + " (done: " + todo["completed"] + ")");

// POST: send JSON, read the reply
payload = {"title": "Learn Aussom", "completed": false}.toJson();
created = h.post("https://jsonplaceholder.typicode.com/posts", payload);
c.log("POST status: " + created["info"]["responseCode"]);
c.log("server assigned id " + json.parse(created["body"])["id"]);

// an error status is reported in the response, not thrown
missing = h.get("https://jsonplaceholder.typicode.com/todos/99999999");
if (!missing["info"]["isSuccessful"]) {
    c.log("lookup failed with status " + missing["info"]["responseCode"]);
}

// a connection failure DOES throw - catch it
try {
    h.get("https://no-such-host.invalid/");
} catch (e) {
```

```
    c.log("could not connect to the server");
}
```

Running it prints:

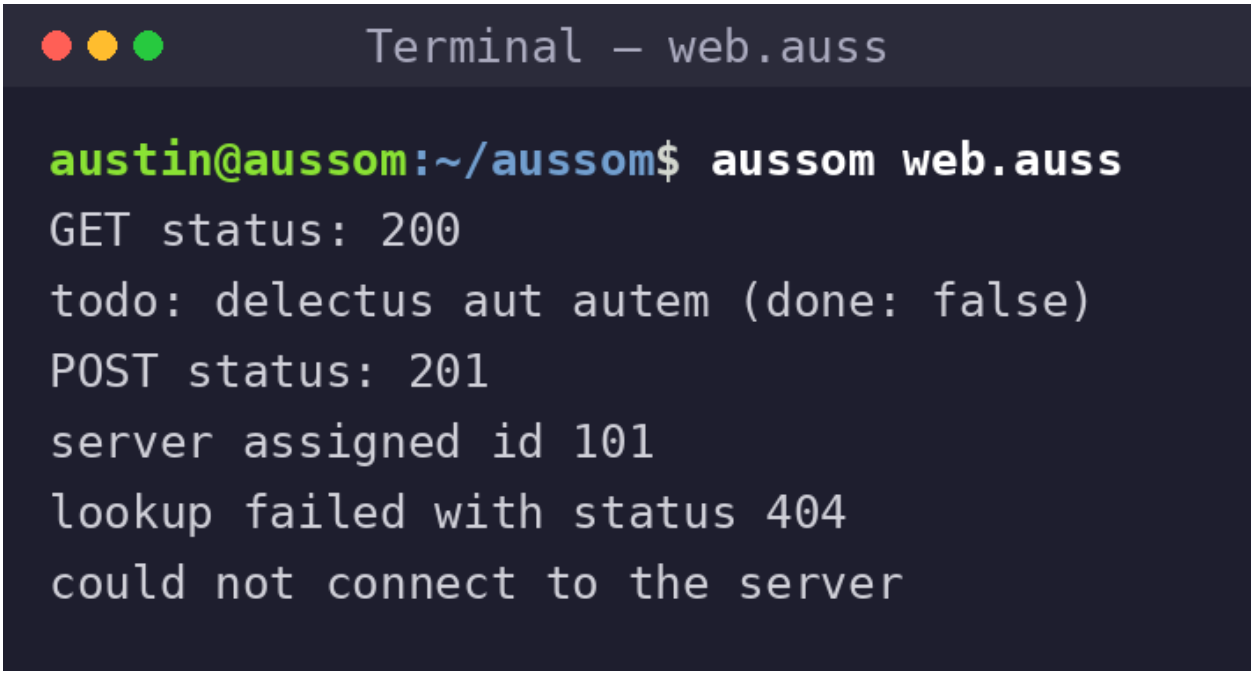


Figure 84: Running web.auss

You can find this example here: `31-web`.

What You Learned

- The **`http`** module (with `include http;`) makes HTTP requests through an **`Http`** object: **`get`**, **`post`**, **`put`**, **`patch`**, and **`delete`**, each returning a response map.
- A response's **`res["body"]`** is the content (a string) and **`res["info"]["responseCode"]`** / **`["isSuccessful"]`** are the status; an optional **`headers`** map is passed as the last argument to send auth tokens and the like.
- Send JSON with **`.toJson()`** as a `post` body and read it back with **`json.parse(res["body"])`** — the same two calls from Chapter 21, over the network.
- Handle the two failure modes differently: **check `isSuccessful`** for an HTTP error status, and **`try/catch`** for a connection failure.

That wraps up **Part IX**. Next, in **Part X**, we start connecting Aussom to bigger systems — beginning with **Chapter 32, Working with Databases**.

Chapter 32 - Working with Databases

Files are fine for saving a little data, but once you have *lots* of records — contacts, orders, users — you want a **database**: a program built to store, search, and update structured data quickly and safely. Most databases speak **SQL**, a standard language for asking questions like "give me every contact in London." Aussom talks to them through the **`jdbc`** module, which connects to any database with a JDBC driver. This chapter uses **SQLite** — a small, file-based database that ships with Aussom, so you can follow along with no server to install.

> **A note on SQL.** This chapter shows *how Aussom runs* SQL, not SQL itself — that's a big topic of its own. The queries here (`SELECT`, `INSERT`, `UPDATE`, `CREATE TABLE`) are the common ones; a short SQL tutorial will fill in the rest.

Connecting with `jdbc`

A database lives behind a **driver** — a jar that teaches Java how to talk to that specific database. Aussom bundles the **SQLite** driver and keeps it on the classpath for you, so you don't have to load anything — just `include jdbc` and connect:

```
include jdbc;
```

You create a **`Jdbc`** object and give it the connection details — the driver's class name and a **URL** that names the database. For SQLite, the URL is just `jdbc:sqlite:` followed by a filename:

```
db = new Jdbc();
db.setConnectionInfo("org.sqlite.JDBC", "jdbc:sqlite:contacts.db", "", "");
db.connect();
```

`setConnectionInfo(driver, url, user, password)` sets everything in one call (SQLite needs no username or password, so those are empty). Then **`connect()`** opens the connection. When you're done, **`disconnect()`** closes it. A few related calls:

- **`connect`** / **`disconnect`** — open and close the connection.
- **`isConnected`** — check whether it's open.
- **`setUrl`** / **`setDriver`** / **`setUserName`** / **`setPassword`** — set the pieces one at a time, if you prefer, instead of `setConnectionInfo`.

To connect to a different database — PostgreSQL, MySQL, and so on — you supply *that* database's driver jar yourself and load it at runtime with **`app.loadJar`** before connecting, then use its class name and URL; the rest of this chapter works the same for all of them:

```
include app;
include jdbc;

app.loadJar("postgresql-42.7.3.jar");   // a driver Aussom doesn't
                                        // bundle
```

When you package an application with APAC (Chapter 37), the bundled SQLite driver is copied into your installer automatically, so a program that ships with SQLite just works on the user's machine — nothing extra to install.

Running Queries and Reading Results

To ask the database a question, you run a **`SELECT`** query. The friendliest method is **`selectMaps`**, which returns the matching rows as a **list of maps** — one map per row, keyed by column name:

```
rows = db.selectMaps("SELECT id, name, city FROM contact");
for (r : rows) {
    c.log(r["name"] + " lives in " + r["city"]);    // read each column
                                                    // by name
}
```

Each element of `rows` is a map like `{"id": 1, "name": "Ada", "city": "London"}`, so you pull out a column with `r["name"]` — exactly the map access from Chapter 11.

Most queries need to be **filtered** by a value, and here's the important part: never glue that value into the query text yourself. Instead, put a **`?`** placeholder in the SQL and pass the values as a **list** — Aussom fills them in safely:

```
rows = db.selectMaps("SELECT * FROM contact WHERE city = ?", ["London"]);
```

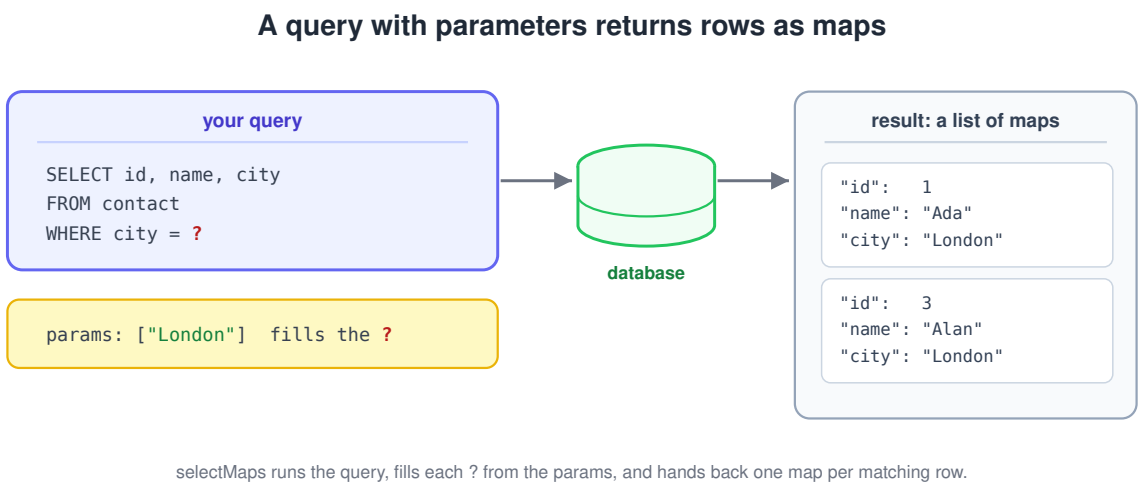


Figure 85: A query with parameters returns rows as maps

The ? gets replaced with `"London"` from the list. This isn't just convenient — it's **safe**: it stops a stray quote or a malicious value in the data from breaking (or hijacking) your query, a classic bug called *SQL injection*. Always use ? placeholders for values. Two sibling readers exist for special cases:

- **`select`** — returns rows as plain **lists** instead of maps (positional, no column names).
- **`selectEach(query, params, callback)`** — hands each row to a callback one at a time, handy for very large results you don't want to hold in memory all at once.

Inserting and Updating Data

Changing data — adding, updating, or deleting rows — uses **`update`**, which runs an `INSERT`, `UPDATE`, or `DELETE` and returns the **number of rows affected**. It takes the same ? placeholders and params list:

```
db.update("INSERT INTO contact (name, city) VALUES (?, ?)", ["Ada",
↪   "London"]);

changed = db.update("UPDATE contact SET city = ? WHERE name = ?",
↪   ["Cambridge", "Ada"]);
c.log(changed + " row(s) updated");     // 1
```

Statements that don't return a count and don't take values — like creating a table — go through **`execute`**:

```
db.execute("CREATE TABLE contact (id INTEGER PRIMARY KEY, name TEXT, city
↪   TEXT)");
```

For the common jobs beyond those, `jdbc` offers:

- **`insertReturningKeys`** — insert a row and get back its new auto-generated id.
- **`updateBatch(query, paramRows)`** — run one statement many times (a list of param lists), efficient for bulk inserts.
- **`commit`** / **`rollback`** with **`setAutoCommit(false)`** — group several changes into a single **transaction** that either all succeed or all undo together.

Try It Yourself

This program creates a table, inserts three contacts, queries by city, and updates a row — a full tour of the module against a real SQLite file. Save it as `contacts.auss` and run it:

```
include jdbc;

db = new Jdbc();
db.setConnectionInfo("org.sqlite.JDBC", "jdbc:sqlite:contacts.db", "", "");
db.connect();

db.execute("DROP TABLE IF EXISTS contact");
db.execute("CREATE TABLE contact (id INTEGER PRIMARY KEY, name TEXT, city
↪   TEXT)");
```

```
db.update("INSERT INTO contact (name, city) VALUES (?, ?)", ["Ada",
↪   "London"]);
db.update("INSERT INTO contact (name, city) VALUES (?, ?)", ["Grace", "New
↪   York"]);
db.update("INSERT INTO contact (name, city) VALUES (?, ?)", ["Alan",
↪   "London"]);

rows = db.selectMaps("SELECT id, name, city FROM contact WHERE city = ?",
↪   ["London"]);
c.log("Contacts in London (" + #rows + "):");
for (r : rows) {
    c.log("  #" + r["id"] + "  " + r["name"] + " - " + r["city"]);
}

changed = db.update("UPDATE contact SET city = ? WHERE name = ?",
↪   ["Cambridge", "Ada"]);
c.log("Rows changed: " + changed);
ada = db.selectMaps("SELECT city FROM contact WHERE name = ?", ["Ada"])[0];
c.log("Ada now lives in " + ada["city"]);

db.disconnect();
```

Running it prints:

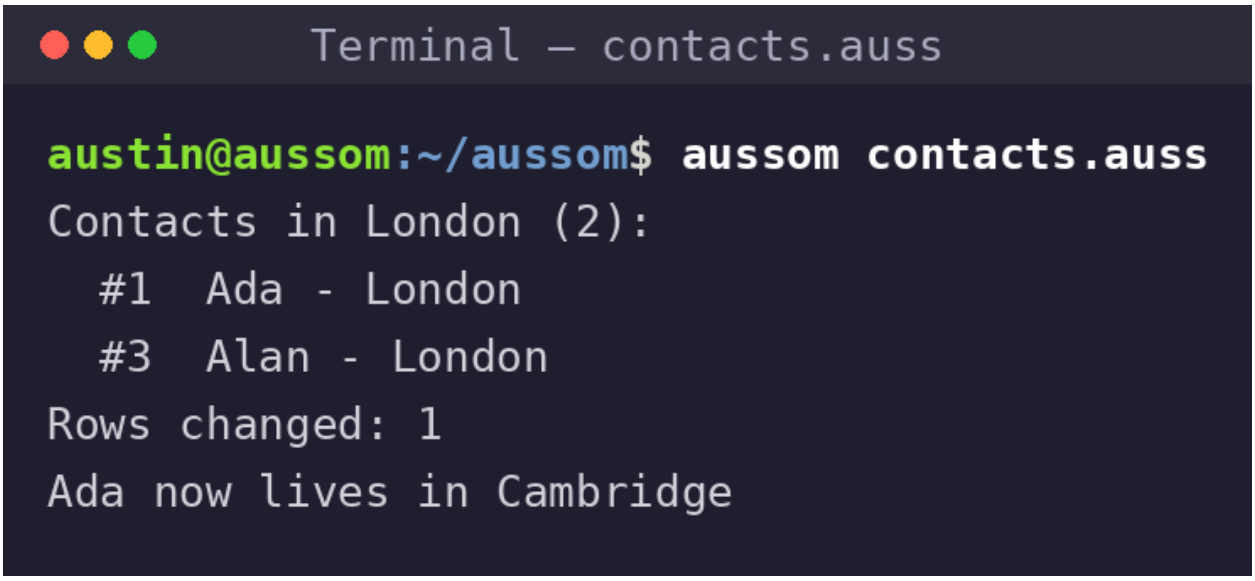


Figure 86: Running contacts.auss

The program creates a `contacts.db` file in the current folder — that's the whole database, in one file. You can find this example here: `32-databases`.

What You Learned

- The **`jdbc`** module connects to SQL databases. Aussom bundles **SQLite** (a file-based database) and keeps its driver on the classpath, so you just `include jdbc`; a **`Jdbc`** object connects with **`setConnectionInfo(driver, url, user, pass)`** and **`connect`** (and **`disconnect`** when done). For other databases, load their driver jar with **`app.loadJar`** first.
- **`selectMaps(query, params)`** runs a SELECT and returns rows as a **list of maps** (keyed by column); **`select`** returns lists, and **`selectEach`** streams rows to a callback.
- **`update(query, params)`** runs INSERT / UPDATE / DELETE and returns the row count; **`execute`** runs statements like CREATE TABLE.
- Always use **`?`** placeholders with a **params list** for values — it's safe against **SQL injection**. Bigger jobs: `insertReturningKeys`, `updateBatch`, and transactions (`setAutoCommit(false)` + `commit/rollback`).

Next, in **Chapter 33**, we run work on more than one thread with **concurrency**.

Chapter 33 - Concurrency Basics

Everything so far has run **one step at a time**, top to bottom. But a computer has several processor cores, and sometimes you want work to happen **at the same time** — download two files at once, keep a UI responsive while crunching numbers, or split a big job across workers. That's **concurrency**, and Aussom provides it through the **`thread`** module (to run work in parallel) and the **`concurrent`** module (to share data between those threads safely). This chapter covers the basics of both.

Running Work on Threads

A **thread** is a separate line of execution that runs alongside your main program. You create one with **`new Thread()`**, tell it what to run with **`setOnRun`** (giving it a callback — a bound method, from Chapter 15), and start it with **`startThread`**:

```
include thread;

t = new Thread();
t.setOnRun(::doWork);     // doWork() is a method to run on the thread
t.startThread();          // it now runs alongside the main program
```

Once started, the thread runs **independently** — your main program keeps going without waiting. When you *do* need to wait for a thread to finish, call **`join`**, which blocks until it's done:

```
t.join();                 // pause here until t has finished
```

That fork-and-join shape — start several threads, then wait for them all — is the backbone of most parallel work:

A few more `Thread` calls are handy:

- **`join(millis)`** — wait, but give up after a timeout.
- **`isAlive`** — is the thread still running?
- **`sleep(millis)`** — pause the current thread for a while.
- **`setName`** / **`setDaemon`** — label a thread, or mark it a background thread that won't keep the program alive on its own.

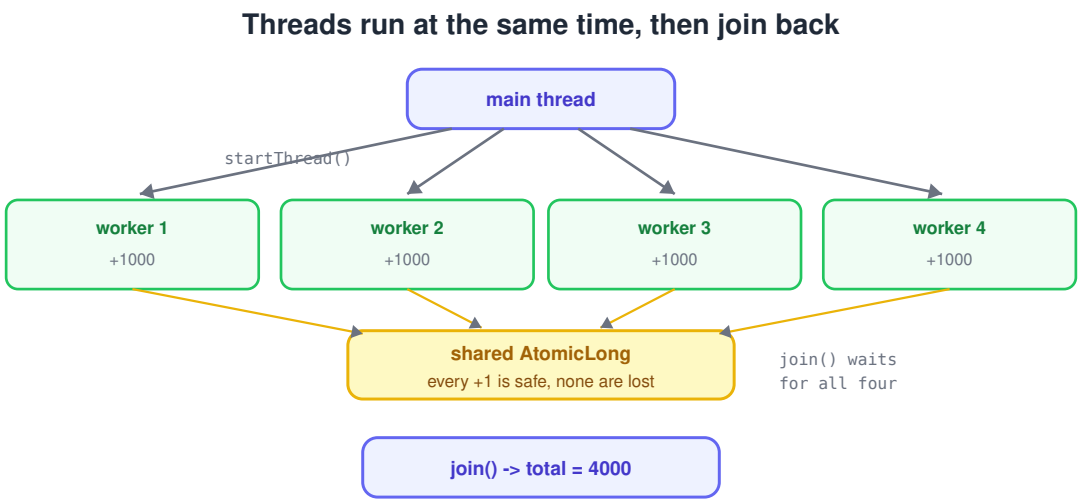


Figure 87: Threads run at the same time, then join back

Sharing Data Safely

Threads get tricky the moment two of them touch the **same data** at once. Say two threads each want to add 1 to a shared counter. Adding 1 is really *three* steps — read the value, add one, write it back — and if both threads read the same starting value before either writes, they'll both write the same result, and **one update is lost**. This is a **race condition**, and it produces bugs that are rare, random, and maddening to track down.

The fix is to use data structures built to be touched by many threads at once. The **`concurrent`** module provides them. The simplest is **`AtomicLong`** — a number whose updates happen as one **indivisible** step, so none can be lost:

```
include concurrent;

total = new AtomicLong(0);
total.incrementAndGet();      // read-add-write as ONE safe step
```

Because `incrementAndGet` can't be interrupted halfway, any number of threads can call it and every increment counts. That's why the example at the end can run four threads adding to one `AtomicLong` and always get the exact right total. Its common operations:

- **`incrementAndGet`** / **`decrementAndGet`** — add or subtract 1, returning the new value.
- **`addAndGet(n)`** — add any amount; **`get`** / **`set`** — read or replace the value.
- **`compareAndSet(expected, new)`** — set it only if it still holds the expected value.

Two more tools cover the cases a single atomic number can't:

A `Lock` guards a *block of code* so only one thread runs it at a time — use it when an update is too complex for an atomic. **`withLock`** runs a callback while holding the lock and releases it automatically when done:

```
lock = new Lock();
lock.withLock(::updateAccount);    // only one thread runs
                                   // updateAccount() at a time
```

A **`BlockingQueue`** hands work *between* threads safely. One thread **`puts`** items in; another **`takes`** them out — and `take` politely **waits** if the queue is empty, so a consumer never spins or misses an item:

```
queue = new BlockingQueue(10);     // holds up to 10 items
queue.put("job-1");                // a producer thread adds work
job = queue.take();                // a consumer thread pulls it out
                                   // (waits if empty)
```

Coordinating with Latch and Semaphore

Beyond sharing data, threads often need to **coordinate** — wait for each other, or take turns. Two more `concurrent` tools handle that.

A **`Latch`** is a one-time gate that lets one thread **wait until others are done**. You start it with a count, each finishing thread calls **`countDown`**, and a waiting thread calls **`await`** until the count reaches zero:

```
done = new Latch(3);          // waiting on 3 things
// ...each worker, when finished, calls:
done.countDown();
// ...the main thread waits for all three:
done.await(5000);             // wait up to 5 seconds
```

A **`Semaphore`** limits **how many** threads may do something at once — useful for capping load, like "no more than 3 downloads at a time." It holds a number of **permits**; a thread **`tryAcquires`** one before proceeding and **`releases`** it after:

```
gate = new Semaphore(3);      // allow 3 at a time
if (gate.tryAcquire(1000)) {  // take a permit (wait up to 1s for one)
    // ... do the limited work ...
    gate.release();           // give the permit back
}
```

Between them, a **`Latch`** answers "wait until everyone's finished" and a **`Semaphore`** answers "let only N through at once."

> **A shortcut for batch work.** A very common job is running the *same* operation over every item in a list, in parallel — a **parallel `foreach`**. You can build one from the threads above, but there's also a ready-made one in the **`integration-core`** package (it works here in the CLI and in Aussom Server). Packages like that are installed with **APAC**, Aussom's package manager — the very next chapter. Once you've met APAC, reach for that parallel `foreach` whenever you need to process a batch quickly.

Try It Yourself

This program starts four threads that each add 1000 to a shared `AtomicLong`, waits for them all with `join`, and prints the total — which is always exactly 4000, with no lost updates. Save it as `workers.aus` and run it:

```
include thread;
include concurrent;

class Counter {
    public total = null;
    public Counter() { this.total = new AtomicLong(0); }

    // Each thread runs this: add 1 to the shared total, 1000 times.
    public count() {
        for (i = 0; i < 1000; i++) {
            this.total.incrementAndGet();
        }
    }

    public main(args) {
        c.log("Starting 4 worker threads...");
        threads = [];
        for (i = 0; i < 4; i++) {
            t = new Thread();
            t.setOnRun(::count);
            t.startThread();
            threads @= t;
        }
        for (t : threads) {
            t.join();
        }
        c.log("All threads finished.");
        c.log("Total counted: " + this.total.get() + "  (4 x 1000)");
    }
}
```

Running it prints:

Try swapping the `AtomicLong` for a plain number and you'll sometimes see a total *less* than 4000 — that's the race condition, live. You can find this example here: `33-concurrency`.

What You Learned

- A **thread** runs work alongside your program. Create one with **`new Thread()`**, set its job with **`setOnRun(::method)`**, launch with **`startThread`**, and wait for it with **`join`**.
- Two threads touching the same data can hit a **race condition** and lose updates. The **`concurrent`** module's thread-safe types prevent it: **`AtomicLong`** (a number whose updates are

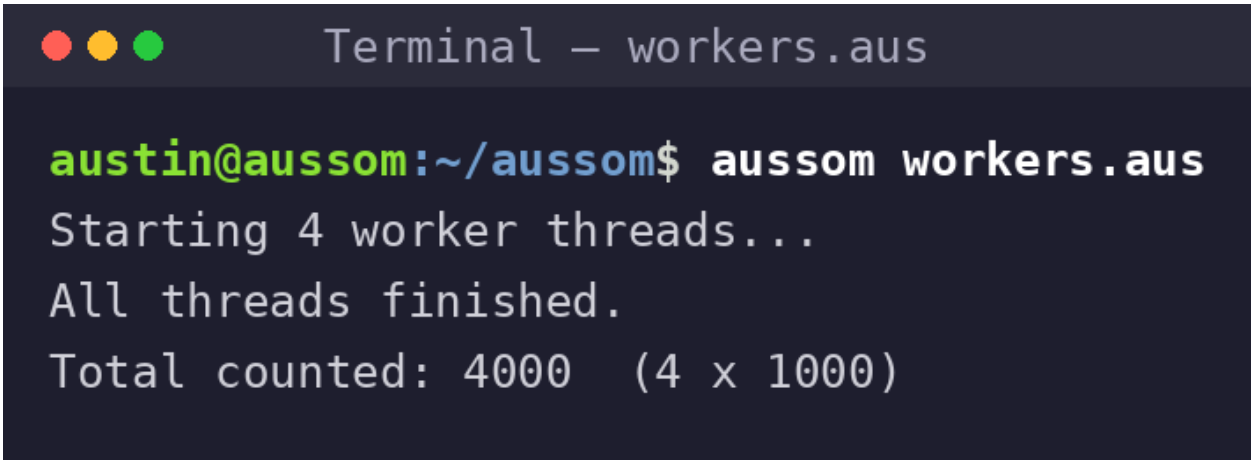


Figure 88: Running workers.aus

indivisible), a **`Lock`** (**`withLock`** guards a block of code), and a **`BlockingQueue`** (**`put`** / **`take`** to hand work between threads).
- Coordinate threads with a **`Latch`** (**`countDown`** / **`await`** — wait until others finish) and a **`Semaphore`** (**`tryAcquire`** / **`release`** — let only N through at once).

Next, in **Chapter 34**, we make our programs trustworthy by **testing** them.

Chapter 34 - Testing Your Code

As a program grows, changing one part can quietly break another. **Tests** are your safety net: small programs that run your code and check it still does what it should. Write them once, and you can run them any time — after every change — to catch mistakes the moment they appear. Aussom has testing built right in through the **`aunit`** module and the `aussom -t` command. This chapter, the last of Part IX, shows how to write and run a test suite.

Writing Tests with `aunit`

A test is just a method that calls your code and **asserts** — states — what the result must be. You group tests in a **class** marked with the **`@Test`** annotation, and mark each test method with `@Test` too. Inside a test method, you call your code and return a **`test.expect`** assertion. Here's a test for a `Calculator` class:

```
include aunit;
include calc;                           // the code being tested

@Test(name = "Calculator Tests")        // marks the class as a test
                                        // suite
class CalculatorTests {

    @Test(name = "add sums two numbers")
    public testAdd() {
        calc = new Calculator();
        return test.expect(calc.add(2, 3), 5);    // assert: add(2,3)
                                                  // must equal 5
    }
}
```

`test.expect(actual, expected)` passes if the two values are equal and **fails** otherwise. Every test method follows this shape: do something, then `return` an assertion about the result. An **`@Test`** annotation takes a friendly `name` that appears in the report, so you can describe *what* each test checks in plain language.

An assertion doesn't just pass or fail silently — when it fails, it records **why**, so a broken test tells you exactly what went wrong instead of leaving you guessing.

Assertions and Test Runners

`test.expect` checks equality, but `aunit` has a whole family of **assertions** for different kinds of check. The ones you'll use most:

- **`test.expect(actual, expected)`** — the two are equal.
- **`test.expectTrue(x)`** / **`test.expectFalse(x)`** — a boolean condition holds.
- **`test.expectNull(x)`** / **`test.expectNotNull(x)`** — a value is (or isn't) `null`.
- **`test.expectThrows(::cb)`** — running the callback *throws* an error (for testing that bad input is rejected).
- **`test.expectClose(a, b)`** — two decimals are equal within a small tolerance (decimals rarely match exactly).
- **`test.expectContains(list, item)`** / **`test.expectKey(map, key)`** / **`test.expectSize(collection, n)`** — checks about collections.

To **run** your tests, use the **`-t`** flag you first saw in Chapter 2:

```
aussom -t calctest.aus
```

This finds every `@Test` class and method in the file, runs them, and prints a **report** — each test with `PASSED` or `FAILED`, then a tally. If every test passes the program exits with code `0`; if any fail, it exits non-zero, so a build script can tell whether the suite is green.

Organizing and Running a Test Suite

Real suites have many tests, and they often share setup — the same object or data each one needs. Rather than repeat that in every method, put it in a **`@BeforeEach`** method, which runs **before every test**. This is the usual way to set up: each test starts with its own **fresh** state, so one test can't affect another. A **`@AfterEach`** method runs after every test, for any per-test cleanup:

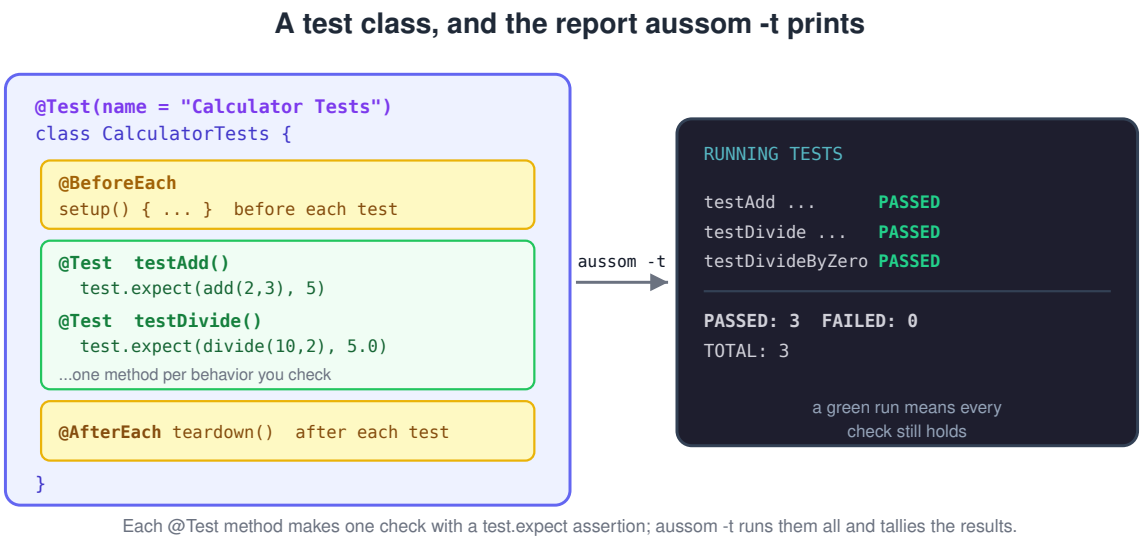


Figure 89: A test class, and the report aussom -t prints

Here's the full suite. `@BeforeEach` builds a fresh `Calculator` for each test, and the three tests check adding, dividing, and that dividing by zero correctly throws:

```
include aunit;
include calc;

@Test(name = "Calculator Tests")
class CalculatorTests {
    private calc = null;

    // give each test a fresh Calculator
    @BeforeEach
    public setup() { this.calc = new Calculator(); }

    @Test(name = "add sums two numbers")
    public testAdd() {
        return test.expect(this.calc.add(2, 3), 5);
    }

    @Test(name = "divide returns the quotient")
    public testDivide() {
        return test.expect(this.calc.divide(10, 2), 5.0);
    }

    @Test(name = "divide by zero throws")
    public testDivideByZero() {
        return test.expectThrows(::divByZero);          // dividing by
                                                        // 0 must throw
    }
    public divByZero() { this.calc.divide(1, 0); }
}
```

Running `aussom -t calctest.aus` prints the report:

Three tests, all green — a summary line of `PASSED: 3 FAILED: 0 TOTAL: 3` tells you the whole suite is healthy. Make a change that breaks add, and the next run flips `testAdd` to `FAILED` with the details, catching the mistake right away.

There's a matching pair of hooks for setup that only needs to happen **once** for the whole suite — say, opening a database connection you reuse: **`@Before`** runs once before any test and **`@After`** runs once after them all. Use `@BeforeEach` for per-test freshness (the common case) and `@Before` for expensive one-time setup. You can find this example here: `34-testing`.

Running Every Suite at Once with `-ta`

`aussom -t file.aus` runs the tests defined **in that one file**. As a project grows you'll have many test files — often one per module — and running them a command at a time gets tedious. The **`-ta`** flag (**t**est-**a**ll) fixes that: it runs the tests in **every class loaded into the engine**, not just the ones in

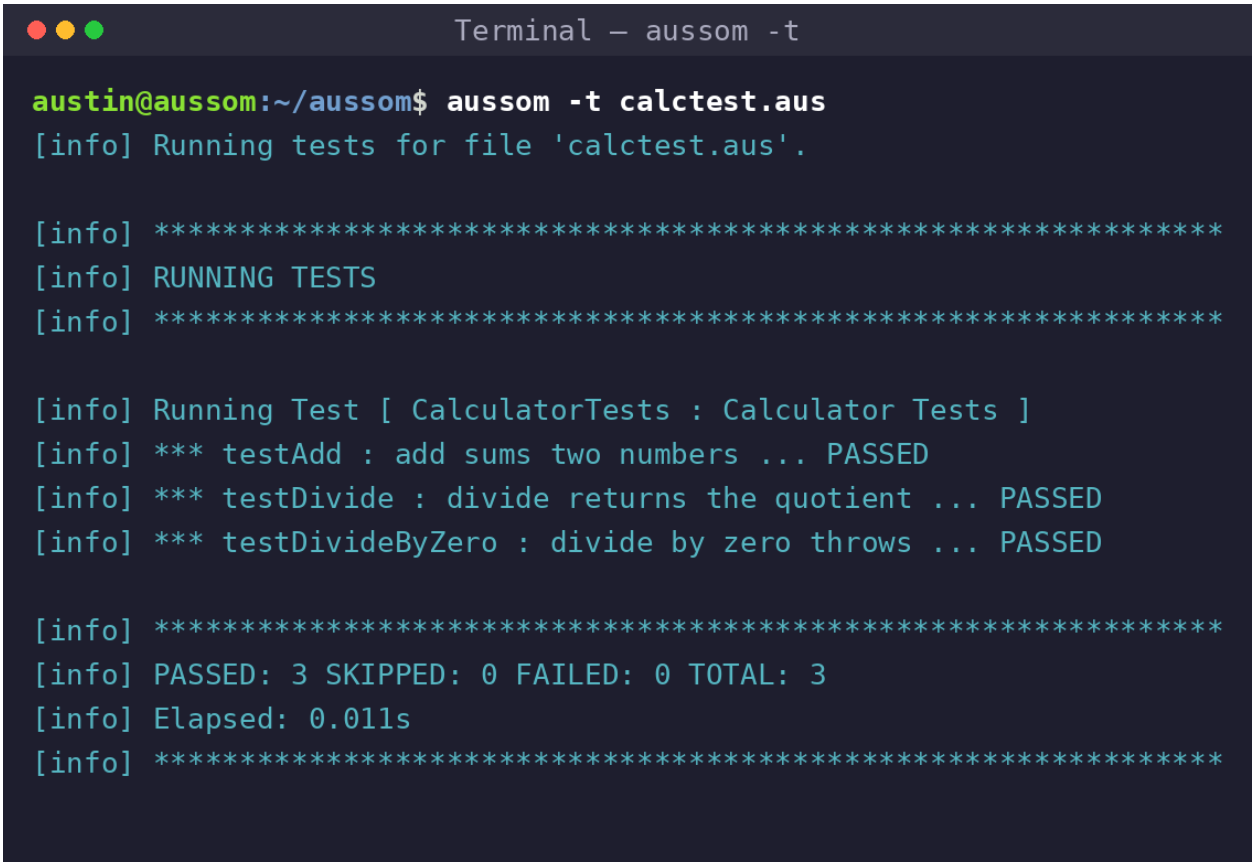


Figure 90: Running the Calculator test suite

the file you name.

The trick is to make a single file that `includes` all your test suites, then point `-ta` at it. Say you have two suites, `calctest.aus` and a second one, `strutiltest.aus`. Create one aggregator file that just pulls them in:

```
// alltests.aus - one file that pulls in every test suite.
include calctest;
include strutiltest;
```

It defines no tests of its own — it only loads the others. Now run them all together:

```
aussom -ta alltests.aus
```

`-ta` loads `alltests.aus`, follows its `includes` to bring in both suites, and runs **every** `@Test` class it finds:

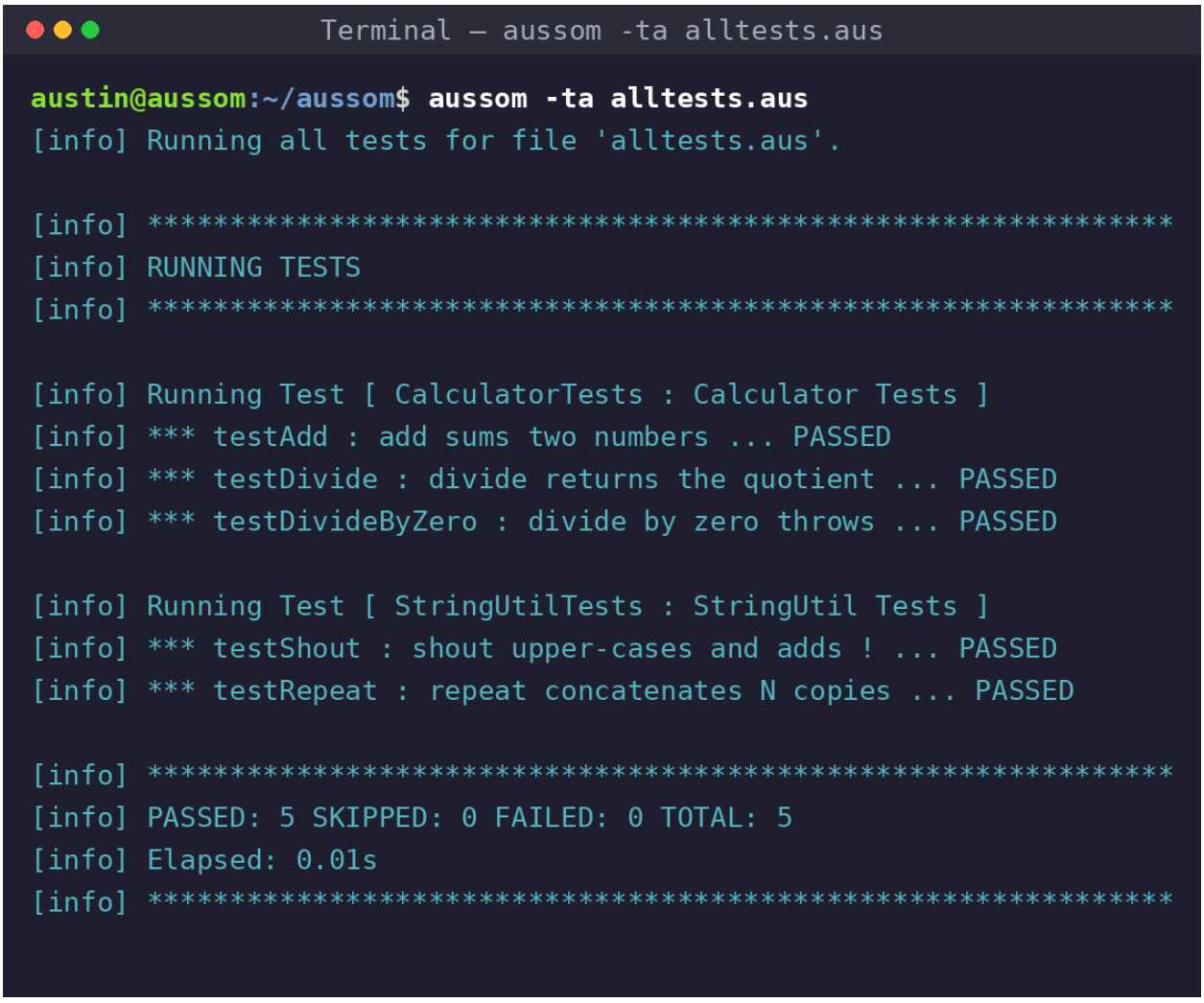


Figure 91: Running every suite at once with aussom -ta

One report, both suites, a combined `PASSED: 5 ... TOTAL: 5` tally. Add a new module later,

give it a test file, drop one more `include` line into `alltests.aus`, and it joins your full test run automatically.

Note the difference: plain `aussom -t alltests.aus` would **not** work here, because `-t` only looks at tests defined *in the named file* — and `alltests.aus` defines none, so it reports "No classes found." Reach for **`-t`** to run one file's tests and **`-ta`** to run a whole aggregated suite.

What You Learned

- Tests are small programs that run your code and **assert** what it should do, so you catch breakage early. Aussom's **`aunit`** module and **`aussom -t`** command run them.
- Mark a **class** and each **test method** with **`@Test(name = "...")`**. A test method does something, then returns a **`test.expect`** assertion.
- Assertions cover many checks: **`expect`** (equal), **`expectTrue`** / **`expectFalse`**, **`expectNull`** / **`expectNotNull`**, **`expectThrows`** (must throw), **`expectClose`** (decimals), and collection checks like **`expectContains`** and **`expectSize`**.
- **`aussom -t file.aus`** runs the suite and prints a **PASSED / FAILED report** with a tally, exiting non-zero if anything fails. Give each test fresh setup with a **`@BeforeEach`** method (and **`@AfterEach`** to clean up); use **`@Before`** / **`@After`** for one-time, whole-suite setup.
- **`aussom -ta file.aus`** (test-all) runs the tests in **every loaded class**, not just the named file. Make an aggregator file that `include`s all your test suites and run it with `-ta` to test the whole project at once.

That completes **Part IX**. Next, in **Part X**, we package and share Aussom code — starting with **Chapter 35, Managing Packages with APAC**.

Part X

Packages and Distribution

Chapter 35 - Managing Packages with APAC

Welcome to **Part X**, where we move from writing code to **sharing and shipping** it. So far every program has used only Aussom's built-in modules or files you wrote yourself. But you rarely have to build everything from scratch — other people have written and published useful **packages** you can pull into your project. Aussom's package manager, **APAC**, finds, installs, and manages those packages. It's a separate command, **apac**, that ships in the same installer as `aussom`.

What APAC Is and How It Fits In

A **package** is a bundle of Aussom code someone has published so others can reuse it — a networking library, a data parser, a UI toolkit. `apac` talks to a central **registry** (a server at `apac.aussom-lang.com`) to search for packages and download them, and it installs them where your programs can find them.

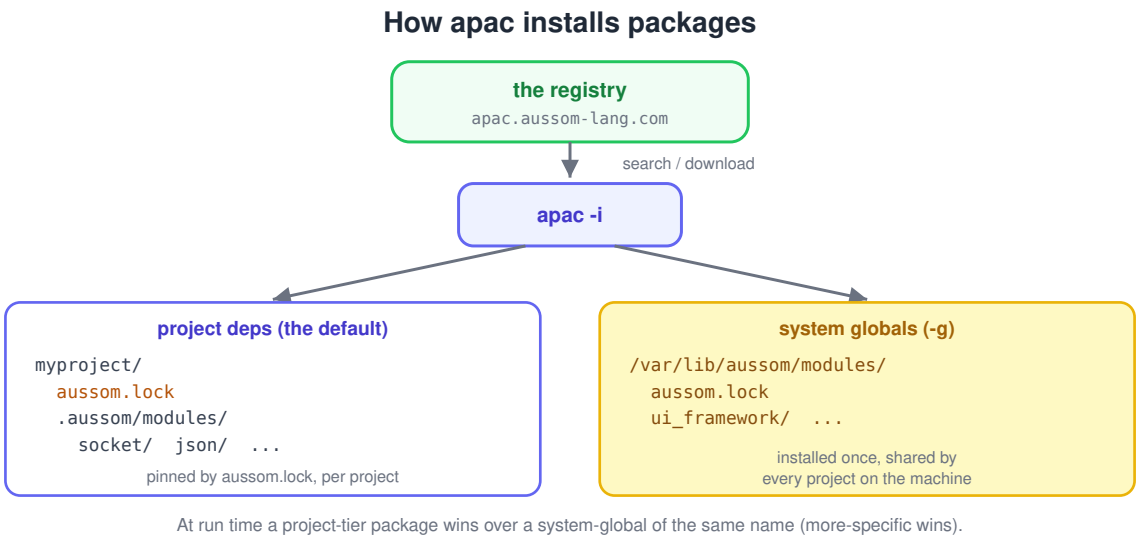


Figure 92: How apac installs packages

There are **two places** a package can be installed:

- **Project dependencies** — the default. Packages land in a `.aussom/modules/` folder inside

your project, and their versions are pinned in a file called `aussom.lock`. Each project gets its own set.
- **System globals** — installed once per machine, shared by every project (with the `-g` flag). Handy for a library every project uses.

Everything below is a `apac` command you run in the terminal. Run **`apac -h`** any time for the full list, and **`apac -v`** for the version.

Searching and Inspecting Packages

Before installing, you'll want to *find* a package and look it over. **`apac -s`** searches the registry, printing a table of matches (name, latest version, type, description):

```
apac -s socket
```

Because it's plain text, you can pipe it through other tools — `apac -s socket | grep tls`. To look closer at one package, **`apac -l`** lists its published **versions**, and adding `@<version>` prints that release's full **manifest** — its description, license, author, dependencies, and more:

```
apac -l socket           # every published version, newest marked
                         # "latest"
apac -l socket@1.0.5     # the full details for one version
```

Both `-s` and `-l` read from the registry, so they always show the latest published information. (The registry is new and growing, so a search may come back empty for now.)

Installing, Listing, and Removing Packages

The everyday command is **`apac -i`**, which installs a package from the registry — fetching it, working out its own dependencies, and unpacking it into your project's `.aussom/modules/`:

```
apac -i socket           # install the latest version
apac -i socket@1.0.5     # pin an exact version
```

If you have a package as a local **zip** file (one a coworker gave you, or one you built yourself — the next chapter), install it with **`-f`**:

```
apac -f -i socket-1.0.5.zip
```

To see what's installed in the current project, **`apac -lf`** prints the lockfile — every package and its pinned version. And **`apac -u`** removes one:

```
apac -lf                 # list installed packages
apac -u socket           # uninstall
```

Here's that lifecycle — installing a package from a zip, listing it, then removing it:

Once a package is installed, its code is available to **`include`** in your programs, exactly like the standard-library modules and your own modules from Chapter 20. The `aussom` CLI adds your project's `.aussom/modules/` to its include path automatically, so — when you run your script from the project folder — the package's files are reachable:

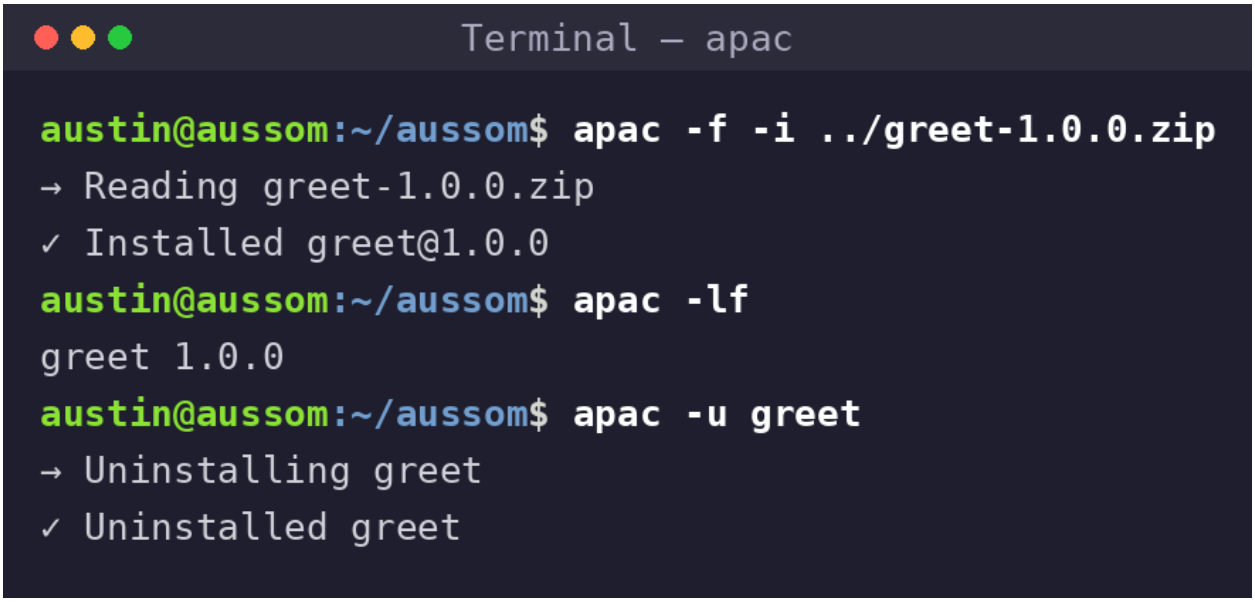


Figure 93: Installing, listing, and removing a package with apac

```
myproject/
    aussom.lock
    .aussom/
        modules/
            socket/
                package.yaml
                socket.aus
                inet.aus
                serverSocket.aus
```

Because the package is a folder, you `include` a file inside it as **`<package>.<file>`** — the same dot-for-a-subfolder form you used with your own modules in Chapter 20 (`include lib.mathutil;`). The `socket` package's main file is `socket.aus`, so:

```
include socket.socket;          // loads socket/socket.aus, the
                                // package's main file
```

The name simply appears twice because the entry file is named after the package. Any other file in the package is reached the same way — `include socket.inet;` loads `socket/inet.aus`, and `include socket.serverSocket;` loads `socket/serverSocket.aus`.

Project Dependencies, System Globals, and the Lockfile

The **lockfile** (`aussom.lock`) is what makes installs *reproducible*. It records the exact version of every package — including the dependencies pulled in behind the scenes — so checking it into source control means a teammate who runs `apac -i` gets the identical set of versions you have. It's the difference between "it works on my machine" and "it works everywhere."

By default everything is a **project dependency**, kept in your project's `.aussom/modules/`. For a

library that *every* project on the machine should share, install it into the **system globals** with `-g` (this writes to a shared system folder, so it needs administrator rights):

```
sudo apac -g -i ui_framework      # install for the whole machine
apac -g -lf                       # list the global packages
```

When the same package name exists in both places, the **project copy wins** — so a project can pin its own version of a shared library without disturbing the global one. That "more-specific wins" rule lets the two tiers coexist cleanly.

Try It Yourself

There's no code file for this chapter — the "example" is the command sequence itself. If you have a package zip (build one with the next chapter's `apac -p`), you can reproduce the whole lifecycle:

```
apac -f -i greet-1.0.0.zip     # install from a local zip
apac -lf                       # confirm it's there
apac -u greet                  # remove it
```

You can find a short walkthrough and the sample package here: `35-apac`.

What You Learned

- **APAC** (apac) is Aussom's package manager. It searches a central **registry**, downloads **packages**, and installs them where your code can `include` them.
- **Find and inspect**: **`apac -s <query>`** searches; **`apac -l <name>`** lists versions; **`apac -l <name>@<ver>`** prints a version's manifest.
- **Install and manage**: **`apac -i <name>[@ver]`** installs from the registry (or **`-f`** from a local zip); **`apac -lf`** lists what's installed; **`apac -u <name>`** removes it.
- Packages install as **project dependencies** (`.aussom/modules/`, pinned in **`aussom.lock`** for reproducible installs) by default, or as **system globals** with **`-g`**. A project copy overrides a global of the same name.

Next, in **Chapter 36**, we cross over to the other side — **creating and publishing** a package of your own.

Chapter 36 - Creating and Publishing a Package

Chapter 35 was about *using* other people's packages. This chapter flips it around: how to turn your own code into a package and share it, so anyone can install it with `apac -i`. A package is just a folder with your `.aus` code, a manifest that describes it, and a license — which `apac` bundles into a zip and uploads to the registry. We'll build a small `string_utils` package from start to publish.

The `package.yaml` Manifest

Every package has a **`package.yaml`** at its root — the **manifest** that names and describes it. You don't write it from scratch; **`apac -ag`** (short for *arcgen*, "archetype generate") scaffolds a whole package for you, manifest included:

```
apac -ag -sm string_utils
```

The **`-sm`** flag stands for *simple*: it produces the flat, Aussom-only layout — just a folder of `.aus` files. (Without `-sm`, `apac -ag` builds the larger Java/Maven layout used for `jvm` and `native` packages, which we'll mention in the next section.) So this command creates a `string_utils/` folder with a `package.yaml`, a `LICENSE.txt`, a `README.md`, and a starter `string_utils.aus`. You then fill in the manifest's details. Here's one for our package:

```yaml
name: string_utils
version: 1.0.0
description: "Small helpers for working with strings."
license: "Apache-2.0"
author: "Ada Lovelace"
authorEmail: "ada@example.com"
type: pure
dependencies: {}
```

The **required** fields:

- **`name`** — lowercase letters, digits, and underscores, starting with a letter (no hyphens). This is what people type in `apac -i`.
- **`version`** — three-part **semver** (`MAJOR.MINOR.PATCH`), like `1.0.0`.
- **`description`** — a one-line blurb (10-200 characters) shown in search results.
- **`license`** — the license *name* (`MIT`, `Apache-2.0`, ...); the full text ships in `LICENSE.txt`.
- **`author`** / **`authorEmail`** — who made it, and a contact address.

- **`type`** — `pure`, `jvm`, or `native` (next section).
- **`dependencies`** — a map of package name to version range (or empty, `{}`).

You can also add optional **`repository`**, **`homepage`**, and **`keywords`** (a short list of search tags). When your package *does* depend on others, each dependency lists a **version range** rather than one exact version:

- **`1.4.3`** — exactly that version.
- **`1.4.*`** — any patch of 1.4; **`1.*`** — any minor of 1.x; **`*`** — any version.
- **`>=1.1.0 <2.0.0`** — a combination, read as *and*.

Package Types: `pure`, `jvm`, and `native`

The manifest's **`type`** field says what *kind* of files the package ships. There are three:

- **`pure`** — Aussom source only. No Java, no native code. The simplest kind, and all you need for most utility packages (like our `string_utils`).
- **`jvm`** — Aussom source plus one or more **Java `.jar`** files, loaded with `app.loadJar(...)` (you saw this in Chapter 32 with the SQLite driver). Use it when your package wraps a Java library.
- **`native`** — Aussom source, JARs, *and* platform **native libraries** (`.so`, `.dll`, `.dylib`). For packages that bind to C libraries.

A `pure` package is a flat folder of `.aus` files; `jvm` and `native` packages start from a Maven project (`apac -ag com.example <name>`) that builds the JAR and assembles the package for you. This chapter sticks with `pure`, which needs no build step.

Building and Publishing to the Registry

Once your code and manifest are ready, **`apac -p`** packages the folder into a publishable zip. It validates the layout, generates a `docs/` folder from your Aussom-Doc comments (the same `/** ... */` comments from Chapter 20), and writes `<name>-<version>.zip`:

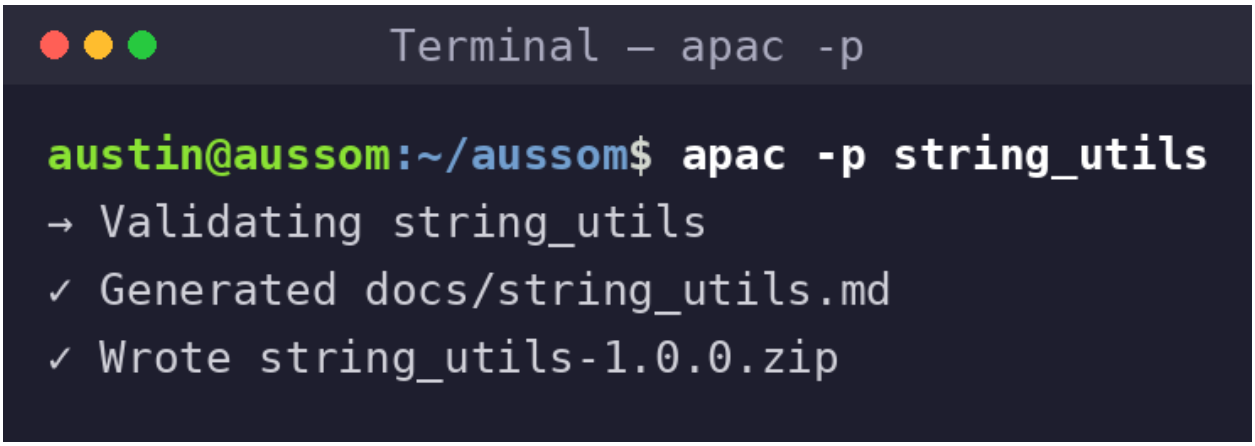


Figure 94: Packaging a source folder with apac -p

The resulting `string_utils-1.0.0.zip` contains everything — manifest, license, README, your `.aus` files, and the generated docs. To **publish** it to the registry, use **`apac -pub`**:

```
apac -pub string_utils-1.0.0.zip
```

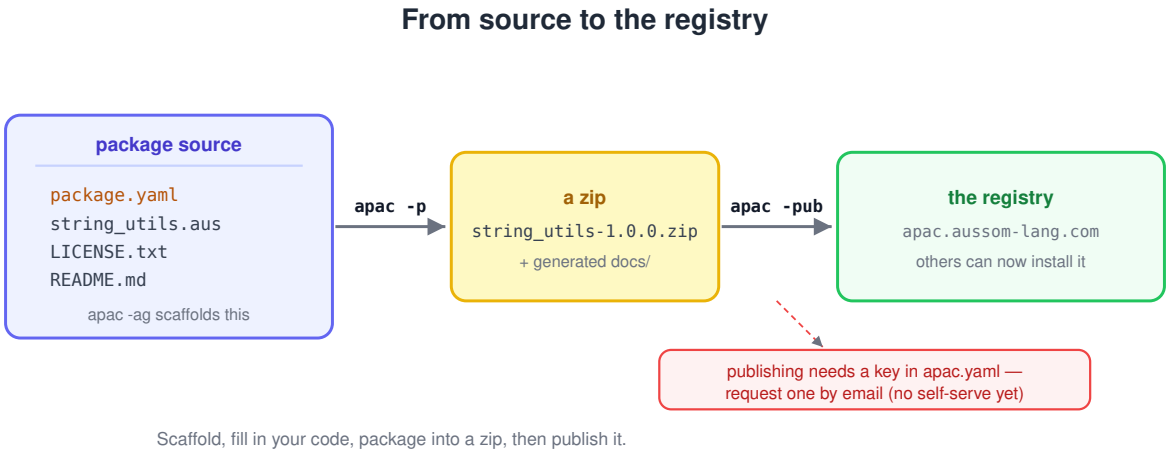


Figure 95: From source to the registry

On success it prints `Published string_utils@1.0.0`, and the package is live for anyone to `apac -i`. Two things to know before you publish:

- **You need a publish key.** Publishing is authenticated. Your key goes in an **`apac.yaml`** config file (in the system-globals folder), under `publish.key`:

```
registry: "https://apac.aussom-lang.com"
publish:
  key: "<your-key-here>"
```

  Getting a key (at the time of writing). There is no self-serve signup yet. To request a publish key, **email `cupofcode@yahoo.com`**. Self-serve signup is planned for a later release. Once you have a key, paste it into `apac.yaml` — `apac` keeps it secret and never prints it back.

- **Versions are permanent.** Once you publish `1.0.0`, you can't overwrite it — publish a new version (`1.0.1`, `1.1.0`, ...) for every change. That's what keeps other people's installs stable.

Try It Yourself

Build the `string_utils` package yourself:

```
apac -ag -sm string_utils      # scaffold the package
# edit string_utils/string_utils.aus and package.yaml
apac -p string_utils           # package it into a zip
```

Publishing needs a key (request one by email, above), so that final `apac -pub` step is one you'll run once you have it. You can find the finished package source here: `36-packages`.

What You Learned

- A **package** is a folder with a **`package.yaml`** manifest, a `LICENSE.txt`, and your `.aus` code. **`apac -ag`** scaffolds one for you.
- The manifest's required fields are **`name`**, **`version`** (semver), **`description`**, **`license`**, **`author`** / **`authorEmail`**, **`type`**, and **`dependencies`** (with version ranges like `1.4.*` or `>=1.1.0 <2.0.0`).
- The **`type`** is **`pure`** (Aussom only), **`jvm`** (adds JARs), or **`native`** (adds native libraries). `pure` needs no build; `jvm`/`native` build from a Maven project.
- **`apac -p`** packages a folder into a `<name>-<version>.zip` (with auto-generated docs); **`apac -pub`** publishes it. Publishing needs a **key in `apac.yaml`** — requested by **email** for now — and published **versions are permanent**.

Next, in **Chapter 37**, we package a whole *application* — building native installers users can double-click.

Chapter 37 - Packaging an Application

Chapter 36 packaged a *library* for other programmers to install. This chapter packages an *application* for ordinary users to run — the kind of thing they **double-click to install**, with no need to know Aussom is even involved. `apac` builds a **native installer** for each operating system — a `.deb` on Linux, a `.msi` on Windows, a `.pkg` on macOS — bundling a Java runtime inside so users install nothing else. We'll turn a small `todo` program into one.

Project Layout and Structure

An application is just a **package** (a `package.yaml` folder, like Chapter 36) with two additions: a **main script** — the `.aus` file with the `main` that runs when the app starts — and an **installer block** in the manifest describing the app to build. You scaffold that installer block with **`apac -ii`** (installer-init), passing the app's display name:

```
apac -ii Todo
```

That does two things. It **extends `package.yaml`** with an `installer:` section, and it creates a **`package-files/`** folder holding per-platform assets — an icons folder, and the small setup scripts each installer format uses (a `deb/` folder for Linux post-install hooks, a `pkg/` folder for macOS, and so on). You fill in the pieces you care about and leave the rest at their defaults.

Bundling Your Code and Resources

The **`installer:`** block is where you describe the app. The top-level fields cover the essentials:

```yaml
installer:
  enabled: true
  appName: Todo
  version: 1.0.0
  vendor: Your Name or Company
  description: A simple to-do app.
  mainScript: todo.aus        # the .aus file with main(args)
  appArgs: [ ]
```

`mainScript` is the important one — it points at the `.aus` file whose `main(args)` runs when someone launches the app. Below the top-level fields, a **`linux`**, **`windows`**, and **`macos`** section lets you tune each platform:

- **`types`** — which installer formats to build (`deb` for Linux, `msi` for Windows, `pkg` for macOS are the defaults).
- **`icon`** — a path under `package-files/icons/` to your app icon (a PNG on Linux, ICO on Windows, ICNS on macOS).
- **`shortcut`** — whether to add a desktop/menu shortcut.
- **`bundle`** — extra pieces to include (JavaFX, GTK4, extra JARs or resources) for apps that need them.

Anything you drop into **`package-files/`** — an icon, a post-install script — gets folded into the installer at build time.

Building Native Installers (deb, msi, pkg)

With the manifest filled in, **`apac -ib`** (installer-build) builds the installers for the operating system you run it on. Under the hood it uses **jpackage**, the Java tool that wraps an app and a runtime into a native installer. Here it is building the Linux `.deb`:

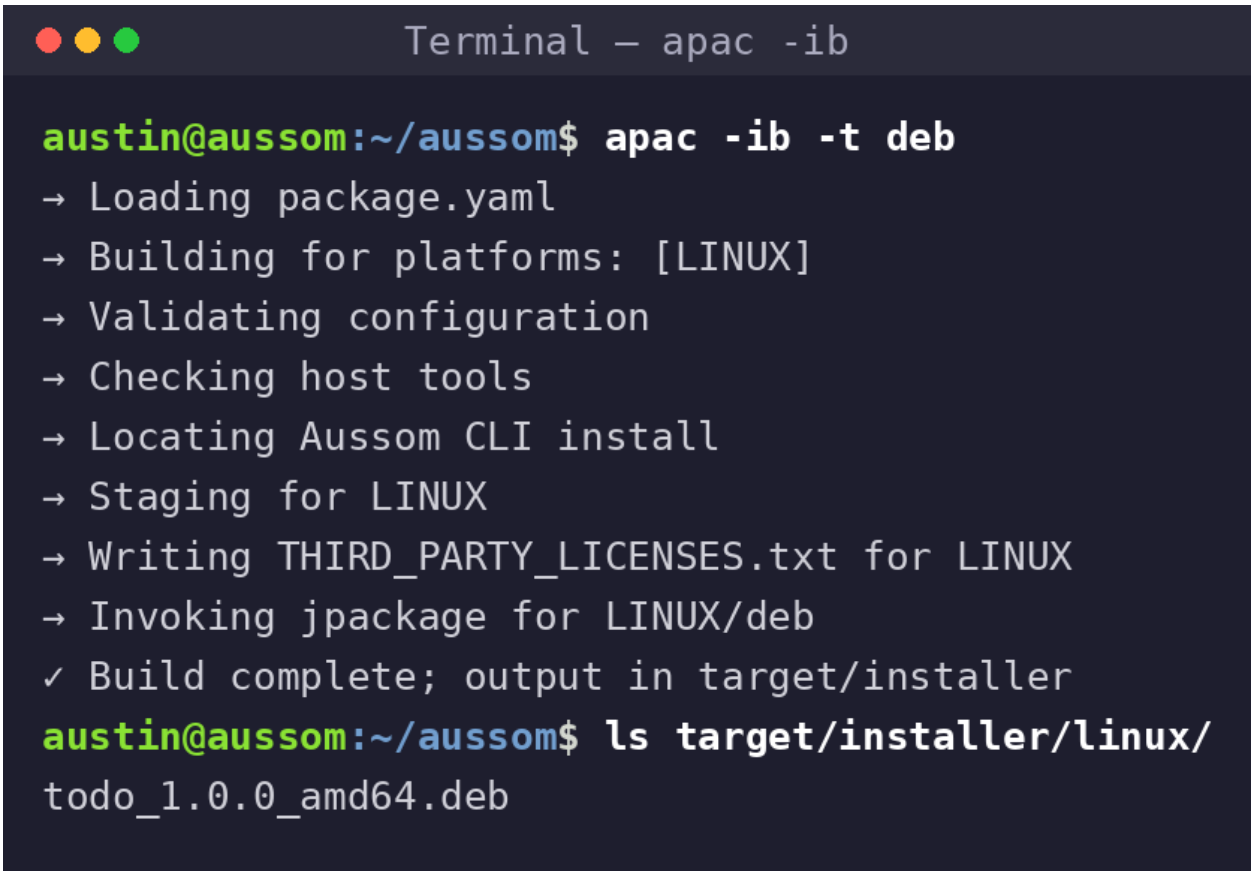


Figure 96: Building a native installer with apac -ib

The result is a real `todo_1.0.0_amd64.deb` a user can install with a double-click or `sudo apt install ./todo_1.0.0_amd64.deb` — and because jpackage bundled a Java runtime inside, they don't need Java (or Aussom) installed at all.

A few useful flags on `apac -ib`:

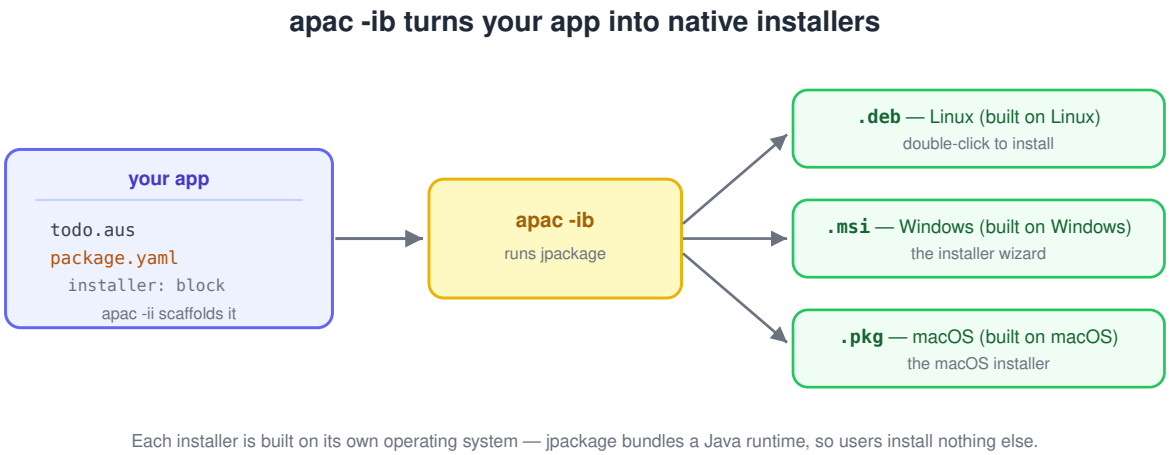


Figure 97: apac -ib turns your app into native installers

- **`-t <type>`** — build just one format (`deb`, `msi`, `pkg`, `rpm`, `dmg`, ...) instead of all of a platform's defaults.
- **`--dry-run`** — print the exact `jpackage` command it *would* run, without building. Handy for seeing what's going on or checking your config.

The one rule: **each installer is built on its own operating system.** jpackage can't cross-compile, so you build the `.deb` on Linux, the `.msi` on Windows, and the `.pkg` on macOS — typically on three machines or a CI pipeline with a runner for each. The manifest and `package-files/` are the same across all three; only the machine you run `apac -ib` on changes.

Try It Yourself

Turn a small program into an installer. Create an app package with a `main`, scaffold the installer config, and build (on Linux, for a `.deb`):

```
apac -ag -sm todo              # a package to hold the app
# put a class with main(args) in todo/todo.aus
cd todo
apac -ii Todo                  # add the installer block +
                               # package-files/
apac -ib -t deb                # build the .deb (on Linux)
```

The finished `.deb` lands in `target/installer/linux/`. Install it and run the app:

```
sudo apt install ./target/installer/linux/todo_1.0.0_amd64.deb
todo "Buy milk"              # prints: Added task: Buy milk
```

The app installs under `/opt/todo/`, and the generated `package-files/deb/postinst` hook symlinks its launcher into `/usr/bin/todo` so the command is on your **PATH** — that's what lets you type `todo` from anywhere. (The matching `prerm` removes the symlink when the package is uninstalled with `sudo apt-get purge todo`.) You can find the sample app here: `37-app-installer`.

What You Learned

- An **application** is a `package.yaml` package with a **`mainScript`** (its `main(args)` entry point) and an **`installer:`** block. **`apac -ii <Name>`** scaffolds that block and a **`package-files/`** folder for icons and setup scripts.
- The installer block sets **`appName`**, **`version`**, **`vendor`**, **`mainScript`**, and per-platform **`linux`** / **`windows`** / **`macos`** options (`types`, `icon`, `shortcut`, `bundle`).
- **`apac -ib`** builds native installers with **jpackage** — a **`.deb`**, **`.msi`**, or **`.pkg`** — each **bundling a Java runtime**, so users install nothing else. Use **`-t`** for one format and **`--dry-run`** to preview.
- Each installer is built on its **own operating system**; jpackage doesn't cross-compile.

That completes **Part X**. Next, in **Part XI**, we bring the whole book together by building a **complete application** from start to finish.

Part XI

Putting It All Together

Chapter 38 - A Complete Application

You've reached the last chapter — and **Part XI**, where everything comes together. Across this book you learned the pieces one at a time: variables, classes, files, JSON, command-line arguments, error handling, packaging. Now we'll use them *together* to build one small but **complete** application: a command-line **task tracker** called `tasks`. It's the kind of real tool you could actually use, and it draws on nearly every idea from the book.

Designing the Program

Before writing code, decide what the program *does*. Our `tasks` tool keeps a to-do list on disk and offers four commands:

- **`tasks add "Buy milk"`** — add a task.
- **`tasks list`** — show every task, with a `[ ]` or `[x]` for its status.
- **`tasks done 1`** — mark task 1 complete.
- **`tasks rm 1`** — remove task 1.

That tells us what we need. Each task is a small record — its text and whether it's done — so a **map** (`{"text": ..., "done": ...}`) fits, and the whole list is a **list of maps**. To remember tasks between runs, we save that list to a file as **JSON**. And we'll split the work into two classes: a **`TaskStore`** that owns the data and the file, and an **`App`** that reads the command line and calls the store.

Building It Step by Step

Start with the data. The `TaskStore` holds the task list and takes care of loading and saving it. Its constructor reads the JSON file if it exists (Chapters 21 and 25), and a private `save` writes it back after every change — so the data is always up to date on disk:

```
include file;

class TaskStore {
    private path = null;
    private tasks = [];

    public TaskStore(P) {
        this.path = P;
        if (file.exists(this.path)) {
```

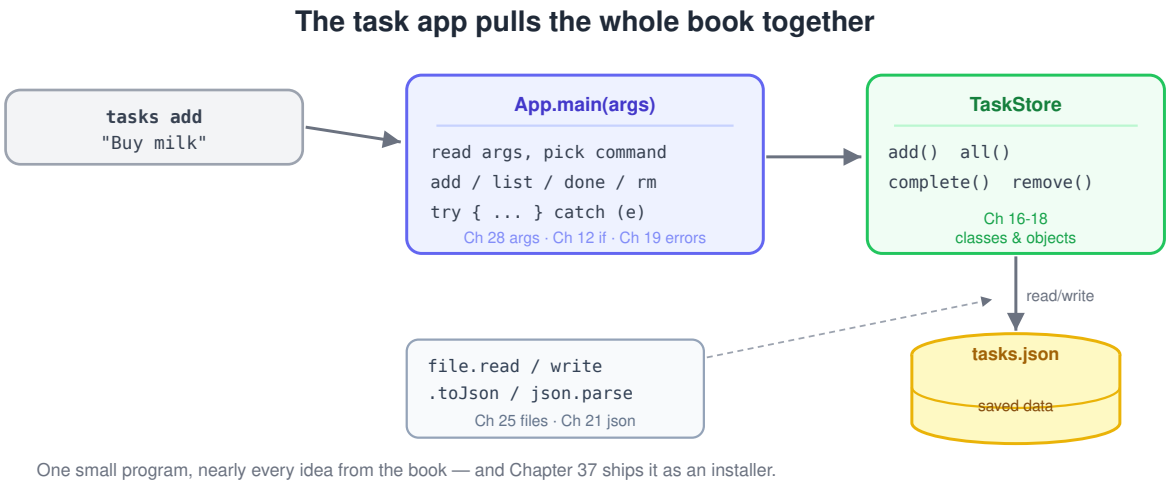


Figure 98: The task app pulls the whole book together

```
            // Load existing tasks from the JSON file.
            this.tasks = json.parse(file.read(this.path));
        }
    }
    private save() { file.write(this.path, this.tasks.toJson()); }

    public add(Text) {
        this.tasks @= {"text": Text, "done": false};
        this.save();
    }
    public all() { return this.tasks; }

    public complete(I) {
        if (I < 0 || I >= #this.tasks) { throw "no task #" + (I + 1); }
        this.tasks[I]["done"] = true;
        this.save();
    }
    public remove(I) {
        if (I < 0 || I >= #this.tasks) { throw "no task #" + (I + 1); }
        this.tasks.removeAt(I);
        this.save();
    }
}
```

Notice how much this one class reuses: **members** and **methods** (Chapter 16), **private** access (the file handling is hidden from the outside), **lists** and **maps** (Chapters 10 and 11), **file I/O** and **JSON** (Chapters 25 and 21), and a **`throw`** when a task number is out of range (Chapter 19).

Then the command line. The `App` class has the `main(args)` entry point. It creates a store, looks at

the first argument to pick a command, and calls the matching `TaskStore` method — all wrapped in a `try/catch` so a bad task number prints a clean message instead of a crash:

```
class App {
    public main(args) {
        store = new TaskStore("tasks.json");
        if (#args == 0) { this.usage(); return; }
        try {
            cmd = args[0];
            if (cmd == "add") {
                store.add(args[1]);
                c.log("Added: " + args[1]);
            } else if (cmd == "list") {
                tasks = store.all();
                if (#tasks == 0) { c.log("No tasks yet."); }
                for (i = 0; i < #tasks; i++) {
                    t = tasks[i];
                    mark = "[ ]";
                    if (t["done"]) { mark = "[x]"; }
                    c.log("  " + (i + 1) + ". " + mark + " " + t["text"]);
                }
            } else if (cmd == "done") {
                store.complete(args[1].parseInt() - 1);
                c.log("Completed task " + args[1] + ".");
            } else if (cmd == "rm") {
                store.remove(args[1].parseInt() - 1);
                c.log("Removed task " + args[1] + ".");
            } else {
                this.usage();
            }
        } catch (e) {
            c.err(e.getText());
        }
    }
    public usage() {
        c.log("usage: tasks <command> [args]");
        c.log("  add <text>   add a task");
        c.log("  list         show all tasks");
        c.log("  done <n>     mark task n complete");
        c.log("  rm <n>       remove task n");
    }
}
```

App adds a few more threads: **`args`** and the `main` entry point (Chapters 27-28), **`if/else`** dispatch and a **`for`** loop (Chapters 12-13), and turning the text argument `"1"` into a number with **`parseInt`** (Chapter 21). Two small classes, and the whole tool is done.

From Source to a Shipped Tool

Save the two classes as **`tasks.aus`** and run it. Here's a real session — adding tasks, listing them, and completing one:

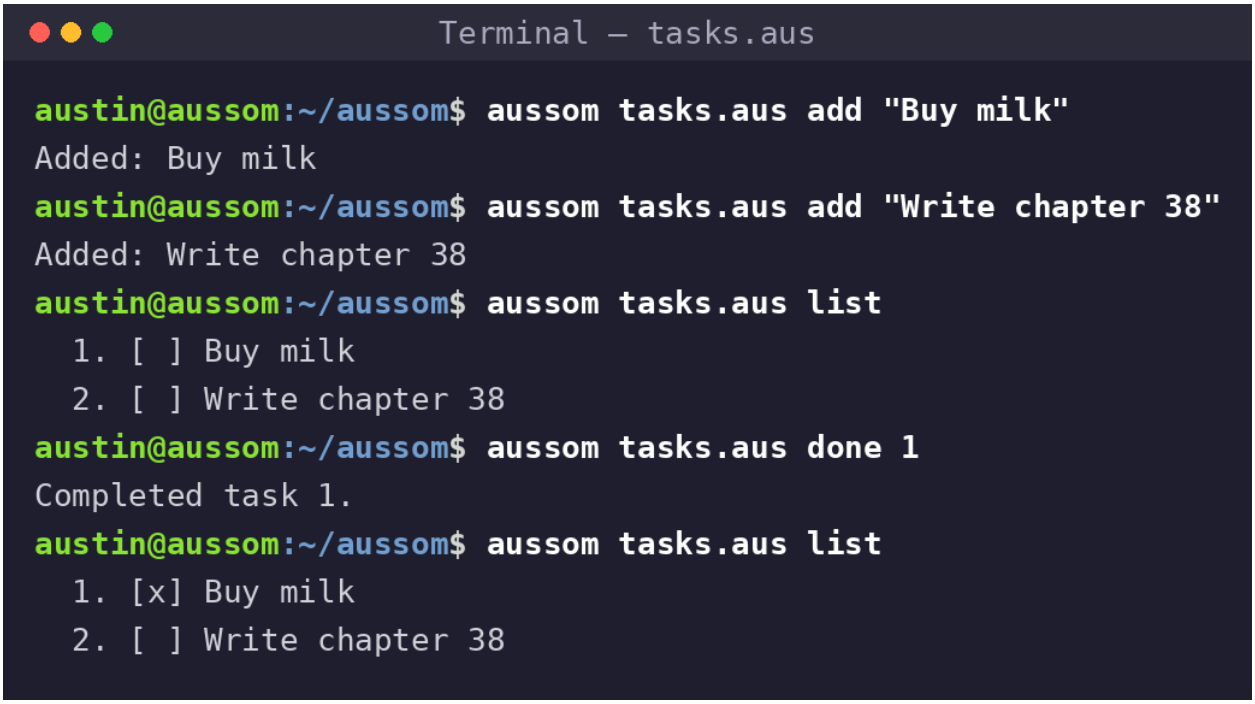


Figure 99: Running the tasks app

Each command saves to `tasks.json` in the current folder, so the list is still there the next time you run it. Try `tasks done 9` and you'll see the friendly `no task #9` from our `catch`, not a stack trace.

Right now you run it with `aussom tasks.aus`. To turn it into a real `tasks` **command** your users can install, package it as an application with the tools from Chapter 37:

```
apac -ii Tasks          # add an installer block; set mainScript to
                        # tasks.aus
apac -ib -t deb         # build the installer (a .deb on Linux)
```

That produces an installer that bundles your code *and* a Java runtime, so anyone can install `tasks` and run it — never knowing, or needing to know, that it's written in Aussom. You've gone the whole way: from a first `c.log("Hello, Aussom!")` to a finished tool you can ship. You can find the complete application here: `38-tasks`.

What You Learned

- A real program is just the book's pieces, **combined**: this task tracker used classes, private members, lists, maps, file I/O, JSON, command-line arguments, loops, and error handling — each introduced in its own chapter, now working together.
- **Design first**: decide what the program does and how its data is shaped, then split the work into focused classes (a `TaskStore` for data, an `App` for the command line).

- **Ship it**: what starts as `aussom tasks.aus` becomes an installable **`tasks`** command with `apac -ii` and `apac -ib` from Chapter 37.

That's the end of the book. You started not knowing Aussom, and you can now write, structure, test, package, and publish real programs in it. The **appendices** that follow are a handy reference — a language cheat sheet, a module index, the style guide in brief, a troubleshooting guide, and where to go next. Congratulations, and happy building.

Part XII

Appendices

Appendix A - Language Quick Reference

A one-page cheat sheet for Aussom's types, operators, and keywords. Each entry points back to the chapter that covers it in full.

Types

Type	What it holds	Example	Chapter
`int`	a whole number	`42`	6-7
`double`	a decimal number	`3.14`	6-7
`string`	text	`"hello"`	6, 8
`bool`	`true` or `false`	`true`	6
`null`	"no value"	`null`	5-6
`list`	an ordered series of values	`[1, 2, 3]`	10
`map`	keyed name-value pairs	`{"a": 1}`	11
object	an instance of a class	`new Dog()`	16
`callback`	a bound method reference	`::greet`	15
`Date`	a moment in time	`new Date().now()`	22
`Buffer`	a fixed-size box of bytes	`new Buffer(16)`	24-25
`exception`	a caught error	`catch (e)`	19

Check a value's type at runtime with **`lang.type(x)`** (Chapter 21).

Operators

Arithmetic: `+` `-` `*` `/` `%` (modulus) `~/` (floor division). Note `/` yields a **double** (`7 / 2` is `3.5`), while `~/` yields the whole part (`7 ~/ 2` is 3).

Comparison: `==` `!=` `<` `>` `<=` `>=`

Logical: `&&` (and) `||` (or) `!` (not)

Assignment: `=` `+=` `-=` `*=` `/=` `%=` `++` `--`

Other:

- `+` — also joins strings (`"a" + "b"`).
- `#` — the size of a list, map, or string (`#items`).

- `[ ]` — index a list or map (`items[0]`, `prices["apple"]`).
- `@=` — append a value to a list (`items @= 4`).
- `.` — reach a member or method (`dog.name`, `dog.speak()`).
- `::` — make a callback to a method (`::onClick`).
- `?` — safe access: `?m["k"]`, `?obj.field`, `?list[i]` return `null` instead of erroring when the key, member, or index is missing.
- `...` — variadic arguments in a function definition (extras arrive as the list `etc`).

Precedence (highest to lowest, Chapter 9):

```
( )   >   ! ?   >   * / ~/ %   >   + -   >   comparisons   >   &&   >   ||
```

When in doubt, add parentheses to make the order explicit.

Keywords

Keyword	Purpose	Chapter
`class`	define a class	14, 16
`new`	create an object	16
`this`	the current object	16
`public` / `private`	member/method access	16
`static`	a single shared class	18
`extern`	a class backed by Java	18
`enum`	a set of named values	—
`return`	return a value from a method	14
`if` / `else`	conditional branching	12
`switch` / `case` / `default`	choose among fixed options	12
`while`	loop while a condition holds	13
`for`	counting and for-each loops	13
`break`	exit a loop early	13
`try` / `catch`	handle errors	19
`throw`	raise an error	19
`include`	load a module or package	20, 35
`instanceof`	test an object's class	17
`null` / `true` / `false`	the literal values	5-6

Aussom has **no** `continue`, `super`, `finally`, or `protected` — see the chapters noted above for what to use instead.

The Shape of a Program

```
// script mode (.auss) - statements run top to bottom
include math;
c.log(math.pi());
```

```
// classical mode (.aus) - the main() of a class runs first
class App {
    public main(args) {
        c.log("Hello, Aussom!");
    }
}
```

See Chapter 28 for when to use each.

Appendix B - Standard Library Module Index

The modules this book covers, grouped by job. **Auto-included** modules are always available; the rest you bring in with `include <module>;`. The last column points to the chapter that covers each one.

Always Available (the `lang` module)

These load automatically — no `include` needed.

Name	What it does	Chapter
`c`	the **console** — `log`, `info`, `warn`, `err`, `print`, `println`	1, 21
`string`	text methods — `toUpper`, `substr`, `split`, `parseInt`, ...	8
`list`	ordered collections — `add`/`@=`, `sort`, `join`, `removeAt`, ...	10
`map`	keyed collections — index by key, `values`, `toJson`, ...	11
`json`	`json.parse` and any value's `.toJson()`	21
`Date`	dates and times — `now`, `format`, `parse`, date math	22
`Buffer`	a fixed-size box of bytes	24-25
`lang`	`lang.type(x)` and other language helpers	21

Core Utilities

Module	What it does	Chapter
`sys`	system info — OS, user, versions	27
`os`	the OS — input, environment, `exec`, `exit`, `printf`/`sprintf`	21, 27
`math`	numbers — rounding, powers, roots, trig, random, constants	23

Module	What it does	Chapter
`util`	`regex`, `base64`, `hex`, `uuid`	24
`file`	read and write files and folders	25
`app`	load a Java `.jar` at runtime (`app.loadJar`)	32, 36

Data Formats

Module	What it does	Chapter
`json`	JSON (auto-included)	21, 26
`xml`	parse and produce XML	26
`yaml`	parse and produce YAML	26

Networking and Data

Module	What it does	Chapter
`http`	make HTTP requests (`get`, `post`, ...)	31
`jdbc`	connect to SQL databases	32

Concurrency

Module	What it does	Chapter
`thread`	run work on threads (`Thread`, `Timer`)	33
`concurrent`	thread-safe types — `AtomicLong`, `Lock`, `BlockingQueue`, `Latch`, `Semaphore`	33

Command-Line and Terminal UI

Module	What it does	Chapter
`cliparser`	parse command-line options and flags	29
`jansi`	color, styling, and cursor control	30
`curses`	full text user interfaces (TUIs)	30

Testing

Module	What it does	Chapter
`aunit`	write tests (`@Test`, `test.expect`); run with `aussom -t`	34

Java and Native Interop (advanced)

Module	What it does	Chapter
`aji`	call Java classes and objects directly at runtime	Appendix F
`panama`	call native C library functions (Foreign Function & Memory API)	Appendix F

And More

Aussom ships additional modules this beginner book doesn't cover — for sockets, compression, codecs, Markdown, templates, reflection, and desktop UIs (JavaFX), among others. Browse the full API reference at aussom-lang.com, and remember you can install still more from the package registry with **APAC** (Chapter 35).

Appendix C - The Aussom Style Guide in Brief

Consistent style makes code easier to read, maintain, and document. This is a short summary of Aussom's official conventions; the full guide lives at aussom-lang.com.

Naming

Thing	Style	Example
Regular class	**PascalCase**	`BankAccount`, `TaskStore`
Static class / enum	**lowercase**	`sys`, `math`, `dateunit`
Method (function)	**camelCase**	`getBalance`, `addTask`
Method argument	**PascalCase**	`Name`, `Amount`, `Data`
Local variable	**camelCase**	`total`, `rows`
Member variable	**camelCase**	`firstName`, `balance`

The lowercase names for **static classes and enums** signal that they're single, shared things rather than blueprints you make objects from. Capitalizing **method arguments** keeps them visually distinct from local variables and from members accessed through `this`. The examples throughout this book follow these conventions — you'll see `add(A, B)` and `greet(Name)`, with camelCase locals and members inside.

Indentation and Bracing

- Indent with **tabs** (matching the standard library).
- **Opening braces** go on the **same line** as the declaration:

```
class Dog {
    public speak() {
        return "woof";
    }
}
```

- **Closing braces** go on their own line. For a short, single-statement body, the whole thing may sit on one line: `public area() { return this.w * this.h; }`.

Documentation Comments

Document classes and methods with **Aussom-Doc** comments — a block comment that opens with `/**`. Two tags add structure to a method: **`@p`** documents a parameter (using the parameter's name), and **`@r`** describes the return value:

```
/**
 * Builds a friendly greeting.
 * @p Name is the name to greet.
 * @r string - the finished greeting.
 */
public greet(Name) {
    return "Hello, " + Name + "!";
}
```

Running `aussom -d file.aus` turns these comments into Markdown documentation (Chapter 20), and `apac -p` includes them automatically when it packages your code (Chapter 36). Use plain `//` comments for ordinary in-code notes, and reserve `/** */` for documentation.

A Few More Habits

- Keep one clear job per class; split large files into modules (Chapter 20).
- Prefer `?`-placeholder queries and validated input over hand-built strings (Chapters 24, 32).
- Wrap risky work in `try/catch` and give the user a clear message (Chapter 19).
- Name things for what they *mean*, not how they're built — future you will thank you.

Appendix D - Troubleshooting and Lessons Learned

The errors and surprises you're most likely to hit, and how to fix them. Each points to the chapter with the full story.

Common Errors

`command not found: aussom` — the terminal can't find Aussom. It was probably installed into a shell that's still open. Close the terminal and open a new one so it picks up the new command. (Chapter 2)

`PARSE_ERROR: Top-level statements are not allowed in this parse context` — you ran a file full of top-level statements in *classical* mode, which expects a class with a `main`. Give the file the **`.auss`** extension, or run it with **`aussom -s file.aus`**, to use script mode. (Chapter 28)

`Division by 0 exception` — a `/` (or `~/` or `%`) with a zero on the right. Check the denominator first, or wrap the calculation in `try/catch`. (Chapter 19)

`Index out of bounds` — you asked for a list position that doesn't exist. Remember lists are **0-based**: a list of 3 items has indexes 0, 1, 2. Guard with `#list` before indexing. (Chapter 10)

Numbers glued together instead of added — `"3" + 1` gives `"31"`, not 4, because the left side is a string (often from user input or a file). Convert it first with **`parseInt()`** or **`parseDouble()`**. (Chapter 21)

`Object 'X' has no overload of 'Y'` — you called a method `Y` that the value doesn't have — a typo, the wrong type, or a `null` where you expected an object. Check the name, and make sure the object was actually created. (Chapters 8, 16)

A callback "has no object" when called — you tried `cb(args)`. Invoke a callback with **`cb.call(args)`** instead. (Chapter 15)

Gotchas (Surprising, but Correct)

These aren't bugs — just behavior worth remembering.

- **`new Date()` is the epoch (1970), not now.** Use **`new Date().now()`** for the current time. (Chapter 22)

- **`math` functions return decimals.** `math.sqrt(144)` is `12.0`, not 12. Call **`.toInt()`** when you need a whole number. (Chapter 23)
- **Maps don't keep their order.** The keys may come back in a different order than you added them. (Chapter 11)
- **`/` always gives a double.** `7 / 2` is `3.5`; use **`~/`** for whole-number (floor) division. (Chapters 7, Appendix A)
- **Regex backslashes must be doubled.** Write `"\\d+"` in an Aussom string to mean the regex `\d+`. (Chapter 24)
- **A `Buffer` has a fixed size.** Size it to your data (`new Buffer(#text)`) before encoding, or you'll encode the empty padding too. (Chapter 24)
- **Aussom has no anonymous functions.** A callback is always a method reference, `::method`. (Chapters 15, 33)
- **Including an installed package uses `<package>.<file>`.** After `apac -i socket`, write **`include socket.socket;`** (the entry file), not `include socket;`. (Chapters 20, 35)

Things Aussom Doesn't Have

If you're coming from another language, don't reach for these — they don't exist:

- **`continue`** in loops — wrap the body in an `if` instead. (Chapter 13)
- **`super`** — give a parent method a distinct name to call it. (Chapter 17)
- **`finally`** — put "always run" code after the `try/catch`. (Chapter 19)
- **`protected`** — only `public` and `private`. (Chapter 16)

When You're Stuck

- **Read the whole error.** Aussom's messages name the file, line, and what it expected — and a **stack trace** shows the chain of calls that led there (Chapter 19).
- **Print your way in.** A few `c.log(...)` calls to show what a value actually holds solves most mysteries.
- **Check the type.** `c.log(lang.type(x))` confirms whether `x` is the `int`, `list`, or object you think it is (Chapter 21).
- **Test small pieces.** Write a quick `aunit` test around the part you suspect (Chapter 34).

Appendix E - Further Resources

Where to go once you've finished the book — to look things up, get help, and keep building.

The Aussom Website

Everything official starts at **aussom-lang.com**. The top navigation leads to:

- **Docs** — the full documentation, including the **Quick Start**, the **Language Guide**, and the complete **API Reference** for every module and type. When this book says "see the API docs," this is where to look.
- **Download** — installers for all three platforms (a `.deb` for Linux, a `.pkg` for macOS, a `.msi` for Windows), plus the editor plugins below.
- **Playground** — an in-browser Aussom environment at **playground.aussom-lang.com** where you can try code without installing anything. Great for quick experiments.
- **APAC**, **News**, and **Books** round out the menu.

Aussom comes in a few **editions** — the **CLI** you used in this book, **Aussom Base** (an embeddable interpreter for the JVM), **Aussom-Script** (Aussom in the browser), and **Aussom Server** (for hosting web apps). The docs site has a section for each.

Editor Plugins

Writing Aussom is much nicer with editor support — syntax highlighting, validation, code completion, and run/debug. Two plugins are on the **Download** page (Chapter 3):

- **IntelliJ IDEA** plugin.
- **VS Code** plugin.

Install the one for your editor and reopen your Aussom files.

The Package Registry

APAC (Chapters 35-36) pulls in published packages from the registry at **apac.aussom-lang.com**. Use `apac -s <query>` to search it, and — once you have a publish key — `apac -pub` to share your own packages with everyone else.

Built-In Help

You don't always need the web. The tools carry their own reference:

- **`aussom -h`** and **`apac -h`** — the full list of command-line options.
- **`aussom -od`** — open the documentation bundled with your install, offline.
- **`aussom -d file.aus`** — generate Markdown docs from your own code's comments (Chapter 20).

Keep Building

The best next step is a project of your own. Take the task tracker from Chapter 38 and add a feature — due dates, priorities, a `search` command. Or start something new: a small web client with `http` (Chapter 31), a data tool with `jdbc` (Chapter 32), a colorful CLI with `jansi` (Chapter 30). You have all the pieces now. Build something, share it, and welcome to the Aussom community.

Appendix F - Calling Java and Native Code (AJI and Panama)

Aussom runs on the **Java Virtual Machine (JVM)**, and it can reach *outside* the language in two directions: into the enormous world of existing **Java** libraries, and down into the **native** C libraries your operating system already ships. You reach Java with **AJI** and native code with **Panama**. Both are **advanced, optional** features — most programs never need them — but you'll run across them, so here's what they are and how a basic call looks. For the full API, see the docs at aussom-lang.com.

AJI - the Aussom Java Interface

Chapter 18 showed **`extern` classes**, which wrap a Java class so it looks like an Aussom class. **AJI** (the **A**ussom **J**ava **I**nterface) is the lighter-weight alternative: it lets you create and call Java objects **directly at runtime**, with **no Java code and no compile step**. It's ideal for a one-off call into a Java library you don't want to write a whole wrapper for.

Bring it in with `include aji;`. The static `aji` class is your entry point:

```
include aji;

// Call a static Java method (class name, method name, then args).
// Simple results convert straight to Aussom values.
biggest = aji.invokeStatic("java.lang.Math", "max", 3, 9);
c.log("Math.max(3, 9) = " + biggest);  // 9

// Create a Java object, then call its instance methods with .invoke.
sb = aji.newObj("java.lang.StringBuilder");
sb.invoke("append", "Hello, ");
sb.invoke("append", "Java!");
c.log(sb.invoke("toString"));  // Hello, Java!

// Read a static field.
pi = aji.getStaticMember("java.lang.Math", "PI");
c.log("Math.PI = " + pi);
```

You always give a class's **full package path** (`java.lang.Math`, not `Math`). A call like `invokeStatic` or `.invoke` returns a plain Aussom value when the result is simple (an int, a string, and so on); when Java hands back an *object*, AJI wraps it in an **AJO** (Aussom Java Object) that you keep working with

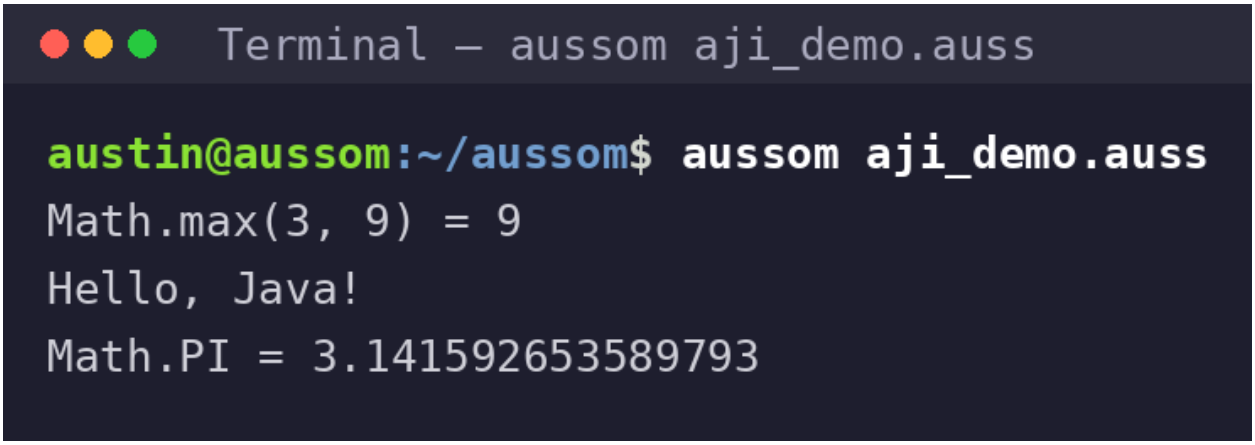


Figure 100: Running aji_demo.auss

through `.invoke`.

The `aji` class and its returned AJOs offer a handful of related calls:

- **`invokeStatic`** / **`invokeStaticRaw`** — call a static method; the Raw form keeps the result wrapped as an AJO instead of converting it to an Aussom value.
- **`newObj`** — construct a new Java object.
- **`getStaticMember`** / **`setStaticMember`** — read or write a static field.
- On an AJO: **`invoke`** / **`invokeRaw`** (instance methods), **`getMember`** / **`setMember`** (instance fields), and **`getName`** (its Java class name).
- **`closure`** — bridge an Aussom callback to a Java functional interface, so Java code can call *back* into your Aussom function.

Panama - Calling Native C Libraries

Panama reaches past the JVM entirely to call functions in **native libraries** — the C `.so`, `.dll`, and `.dylib` files your operating system already has. It's built on Java's **Foreign Function & Memory (FFM) API**, and it's how Aussom talks to system and C libraries with no glue code in another language.

Include it with `include panama;`. The pattern is always the same: open a library, describe a function's **signature**, **bind** the symbol, then **call** it:

```
include panama;

// Open the C standard library.
lib = panama.defaultLibrary();

// int getpid(void) - takes no arguments.
// funcType(returnType, argTypeList) describes the signature.
getpid = lib.bind("getpid", panama.funcType("int", []));
c.log("This process id is " + getpid.call());

// size_t strlen(const char *) - needs a native C string.
```

```
arena = panama.arena();  // a scope that owns native memory
strlen = lib.bind("strlen", panama.funcType("size_t", ["pointer"]));
// cstring copies an Aussom string into the arena, returning a pointer.
text = panama.cstring("native world", arena);
c.log("strlen = " + strlen.call(text));  // 12
```

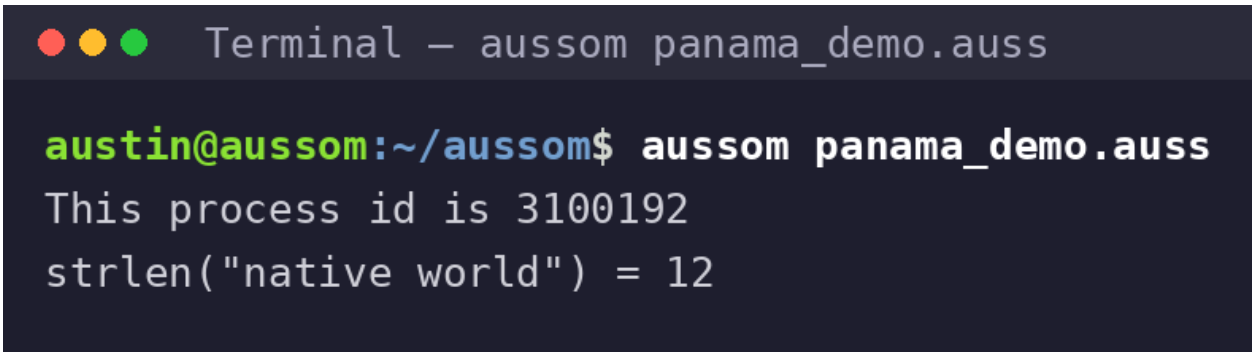


Figure 101: Running panama_demo.auss

Native types are named with **type tokens** — `"int"`, `"size_t"`, `"double"`, `"pointer"`, `"void"`, and so on — and `funcType(returnType, [argTypes])` describes the signature. Anything that isn't a plain number (a string, a struct, an array) lives in native memory you allocate from an **`arena`**; `panama.cstring` above copies an Aussom string into the arena and returns a pointer to it.

A few things to keep in mind:

- **`defaultLibrary`** / **`openLibrary`** — get the C standard library, or open a specific one by name or path.
- **`funcType`** / **`variadicFuncType`** — describe a fixed or variadic signature; **`bind`** turns a symbol into something you **`call`**.
- **`arena`** — owns native memory; helpers like **`cstring`**, **`allocStruct`**, and **`array`** place data there for a call.
- **`callback`** — wrap an Aussom callback so native code can call back into it.
- Panama is **low-level**: you manage native memory and match C types yourself, and a mistake can crash the whole process (not just throw a catchable error). It's also gated by the security manager. Reach for it only when you genuinely need a native library, and lean on the API docs.

You can find both examples here: `appendix-f-interop`.