

ANDROID COURSE

Android Studio 2: The New UI

12 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	The New UI at a Glance	2
Chapter 2	A Settings Tour	9
Chapter 3	New Projects	18
Chapter 4	Gradle in Kotlin DSL	25
Chapter 5	Compose Preview	37
Chapter 6	Devices	43
Chapter 7	Logcat and Debugger	47
Chapter 8	Profiler and Inspection	51
Chapter 9	The Code Menu	55
Chapter 10	Git in the IDE	58
Chapter 11	Agent and Labs	62
Chapter 12	Capstone	65

Android Studio 2: The New UI

Course 2 · Chapter 1 · Finding Your Way Around

You open Android Studio and the window does not look like the one in older books, videos and in *Android Studio: The IDE Itself*. There is no row of menus, the left and right edges carry strips of small icons, and a magnifying glass sits in the top corner. This is the **New UI**, and it is what **Android Studio Rabbit 2** shows by default. This chapter is a map of it: where the menus went, what the icon strips are, what is in the toolbar and how to find anything with one shortcut. Everything later in the course assumes you can find your way around with this map.

> **WHAT WAS CHECKED, AND HOW**
>
> **Checked on a real install.** Every menu name, setting name, shortcut and layout detail below that is marked as seen comes from screenshots of Android Studio Rabbit 2 (stable, 2026.2.2) taken on 2026-11-20 and 2026-11-21: the File menu, the toolbar, the left, right and bottom strips, the Appearance settings page and the Search Everywhere window.
> **Checked in the docs.** One web search against Google's *New UI in Android Studio* page (see the Not Yet Verified box at the end for what that search did and did not show).
> **Not run by me.** I did not use the IDE myself. Where a name below could not be read in a screenshot, the chapter says so instead of guessing.

The Window, Region by Region

Think of the window as a frame with six regions. The names below are the ones you will see used through the rest of the course.

REGION	WHERE	WHAT IT HOLDS
Main toolbar	Top row	The **hamburger button** (three lines) on the far left, then the **project widget** (project name with a drop-down), the **Version Control** widget, the **device selector** (for example *Pixel 10 Pro*), the **run configuration** drop-down (*app*), the green **Run** button, the **Debug** button, a **more** (three dots) menu, and on the right a row of action icons, the magnifying glass (**Search Everywhere**), the gear (**Settings** menu) and your account picture.
Left tool window bar	Left edge	Icons for **Project** and **Resource Manager**, and a three-dots **More tool windows** button that opens the rest.

REGION	WHERE	WHAT IT HOLDS
Right tool window bar	Right edge	Icons for **Notifications**, **Gradle**, **Device Manager**, **Running Devices** and **Agent**.
Bottom tool window bar	Lower left	Icons for windows that open along the bottom: **Run**, **Find**, **Build**, **Problems**, **Terminal** and **Version Control** among others.
Editor area	Centre	Your open files. With nothing open it shows a hint list of shortcuts (below).
Navigation bar	Bottom edge	A path of clickable parts that shows where the open file sits, for example `catalogue > app > src > main > res > drawable`, with the status icons at the far right.

A tool window that is open takes up one of these edges: the **Resource Manager** on the left, the **Agent** on the right and **Build** along the bottom can all be open at once, which is how the window looked in the screenshots this chapter is taken from.

> **COMPARED WITH THE CLASSIC UI**
>
> Android Studio: The IDE Itself 1 describes a window with a menu row across the top, wider tool window tabs on the edges and widgets in the status bar. In the New UI the menu row is folded into one button, the edge tabs are compact icons and project, version control and device choices live in the main toolbar. The editor, the debugger and the refactoring tools are the same ones underneath: only the controls moved.

The Hamburger Menu, and Getting the Menus Back

By default the main menu is **hidden under the hamburger button**: click the three-line icon at the left of the toolbar and the same menus you remember appear as a list. If you prefer them in a row, there is a setting with three choices. Open **File > Settings** (`Ctrl+Alt+S`), then **Appearance & Behavior > Appearance**, and find **Main menu** under **UI Options**:

MAIN MENU SETTING	WHAT YOU GET
Hide under Hamburger Button	The default: one button, menus open from it.
Merge with Main Toolbar	The menu row and the toolbar share one line.

MAIN MENU SETTING	WHAT YOU GET
Show above Main Toolbar	A classic menu row on its own line above the toolbar. This is the setting used in the screenshot of the File menu behind this chapter.

The menu bar then reads **File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS, Window, Help**. Two details are easy to miss. There is **no Analyze menu** any more: its commands moved under **Code**, which Chapter 9 of this course covers. And once a project is a Git repository the **VCS** menu is replaced by one called **Git**, which Chapter 10 covers.

What is in the File menu

The File menu is a good first exploration because it holds several commands you will use constantly. In the screenshot it listed these, in this order:

```
// File menu on Android Studio Rabbit 2 (shortcuts as shown)
New  >                          Open...
Profile or Debug APK            Recent Projects  >
Close Project
Settings...                     Ctrl+Alt+S
Project Structure...            Ctrl+Alt+Shift+S
Plugins...                      File Properties  >
Local History  >
Save All                        Ctrl+S
Sync Project with Gradle Files  Ctrl+Shift+O
Refresh Linked C++ Projects     (greyed out without C++)
Reload All from Disk            Ctrl+Alt+Y
Repair IDE                      Invalidate Caches...
Manage IDE Settings  >          New Projects Setup  >
Export  >                       Print...
Power Save Mode                 Exit
```

Two of these deserve a note now. **Sync Project with Gradle Files** is the one to press after you edit a build file (Chapter 4). **Invalidate Caches...** is the repair command to try when the IDE behaves strangely; it clears its own caches and restarts, and it does not touch your code.

The Tool Window Bars

A **tool window** is any panel around the editor: Project, Build, Logcat, Terminal and so on. In the New UI the bars show only a selection of icons, and the rest sit behind the three-dots **More tool windows** button, so the bars stay short. Click an icon to open its window, click it again to hide it. Every tool window is also listed under **View > Tool Windows**, and the most common have a shortcut.

TOOL WINDOW	SHORTCUT	WHERE IT OPENS
Commit	`Alt+0`	Left
Project	`Alt+1`	Left
Bookmarks	`Alt+2`	Left
Find	`Alt+3`	Bottom
Run	`Alt+4`	Bottom
Debug	`Alt+5`	Bottom
Problems	`Alt+6`	Bottom
Structure	`Alt+7`	Left
Services	`Alt+8`	Bottom
Version Control	`Alt+9`	Bottom
Terminal	`Alt+F12`	Bottom
Logcat	none by default	Bottom

The "Where it opens" column is a note of where each window sat in the screenshots, not a rule: you can drag any tool window to another edge. The same View menu list also contains Profiler, App Inspection, App Links Assistant, App Quality Insights, Backup and Sync History, Build, Build Variants, Coverage, Device Explorer, Device Manager, Gradle, Hierarchy, Layout Validation, Merge Requests, Notifications, Pull Requests, Resource Manager, Running Devices, Agent, TODO and Upgrade Assistant. Later chapters take the important ones in turn.

BARS MISSING, OR NO NAMES ON THE ICONS?

Two switches under **Settings > Appearance & Behavior > Appearance > Tool Windows** change this: **Show tool window bars** (on by default) shows or hides the strips, and **Show tool window names** (off by default) adds a text label beside each icon, which helps while you learn which icon is

> which. The same section has options for side-by-side layout, a widescreen layout and remembering the size of each window. If an icon is a mystery, hover over it for its tooltip.

Search Everywhere

The fastest way to find almost anything is **Search Everywhere**: press `Shift` twice quickly (the editor's empty-state hint calls it `Shift Shift`), or click the magnifying glass at the right of the toolbar. A box opens with a row of tabs: **All, Classes, Files, Symbols, Actions, Text**, a tick-box **Include non-project items** and a note, *Type / to see commands*. Type a few letters and results from every tab appear together: files, code symbols, menu actions and even individual settings pages.

For example, typing `Font` listed settings pages named Font (under Editor and under Console Font and Terminal) next to actions such as *Increase Font Size* and *Reset Font Size in All Editors*. That is the trick for the whole course: **if you cannot remember where a setting or command lives, search for it by name.**

The empty editor also lists the shortcuts worth learning first:

ACTION	SHORTCUT
Search Everywhere	`Shift Shift`
Project view	`Alt+1`
Go to File	`Ctrl+Shift+N`
Recent Files	`Ctrl+E`
Navigation Bar	`Alt+Home`

Go to File searches only file names, so it is the one to use when you know the file you want, and **Recent Files** brings back something you had open a minute ago. **Navigation Bar** puts the keyboard focus on the path at the bottom of the window so you can walk up and down the folders with the arrow keys.

The Welcome Screen

Before a project is open, Android Studio shows the **Welcome screen**. It has the usual project actions (new project, open, get from version control), and, as the next chapters need, it is also the place to reach **More Actions > Virtual Device Manager** and the Settings page without a project open. Once a project is open, the same things live in the menus described above.

Try It Yourself

EXERCISE 1

Make the main menu permanently visible: open **Settings**, change **Main menu** to **Show above Main Toolbar** and apply it. Then open **File** and write down the menu path and the shortcut for *Sync Project with Gradle Files*, and for *Project Structure*.

View solution

EXERCISE 2

Draw (on paper is fine) your own window map: list every icon on your left, right and bottom tool window bars with its name, using hover tooltips or **Show tool window names**. Then open **View > Tool Windows** and note five windows that have *no* icon on the bars and how you would reach them.

View solution

EXERCISE 3

Open the same page, **Settings > Appearance & Behavior > Appearance**, three ways: from the **File** menu, with its keyboard shortcut and with **Search Everywhere** (search for *Appearance*). Then use **Go to File** to open `MainActivity.kt` and say which of the four ways was fastest for you.

View solution

NOT YET VERIFIED

Checked on the install (screenshots of Android Studio Rabbit 2, 2026-11-20 and 2026-11-21): the menu bar names and the File menu list with its shortcuts; the three *Main menu* choices and the *Tool Windows* options on the Appearance page; the left, right and bottom bar contents; the toolbar widgets; the navigation bar path; Search Everywhere's tabs and results; the shortcut hints on the empty

editor; the View > Tool Windows list with shortcuts; no Analyze menu.

Checked in the docs (one web search): the search found Google's *New UI in Android Studio* page. Its summary says the main toolbar prioritises the most-used actions (device, version control, project switching) and can be customised under Menus and Toolbars, and that the tool window bar shows only a selection of icons with the rest behind a *More tool windows* button. The general *Get to know the Android Studio UI* page says Search Everywhere opens with a double Shift or the magnifying glass. I read search summaries, not the full pages, and the search did not confirm the hamburger menu (that comes from the screenshots).

Not verified: the names of the unlabelled action icons on the right of the toolbar; the exact left-to-right order of icons on the bars (it may vary with your layout and what you have opened); the "Where it opens" column (it records the screenshots, not a rule); what *Type / to see commands* offers; whether the Welcome screen layout differs between releases. Defaults can change between Android Studio versions, so trust your own window over this text where they differ.

COMING UP

Chapter 2 is a settings tour: themes (including Islands Dark and Sync with OS), zoom and fonts, editor behaviour, auto import, code style and Backup and Sync.

Android Studio 2: The New UI

Course 2 · Chapter 2 · Making the IDE Yours

Chapter 1 mapped the window. This chapter maps the **Settings** dialog, because almost every complaint about an IDE ("the text is too small", "it keeps adding imports I did not ask for", "the lines wrap at the wrong place") is a setting somewhere in it. The dialog is large, with a tree of dozens of pages, so the aim is not to read every page. It is to learn how the tree is laid out, how to find a page fast, and which handful of pages are worth visiting in your first week.

> **WHAT WAS CHECKED, AND HOW**
>
> **Checked on a real install.** The page names, check-boxes, drop-downs and default values below come from screenshots of Settings on Android Studio Rabbit 2 (stable, 2026.2.2) taken on 2026-11-20 and 2026-11-21: Appearance, Editor > General, Editor > General > Appearance, Auto Import, Code Folding, Font and Code Style.
>
> **Checked in the docs.** One web search, which returned JetBrains documentation for the IntelliJ platform that Android Studio is built on (not an Android Studio page). It covers *Sync with OS* and the Islands themes.
>
> **Not run by me.** Where a page was only partly visible in a screenshot, the chapter says so.

Opening Settings and Finding Things

Open it with **File > Settings...** or `Ctrl+Alt+S` (both seen in the File menu in Chapter 1), or with the gear in the main toolbar, or from the Welcome screen without a project. The dialog has a **tree on the left**, the **selected page on the right**, a **search box** at the top of the tree, and three buttons at the bottom: **OK** (apply and close), **Apply** (apply and stay) and **Cancel** (throw the changes away). A page title at the top shows the path as breadcrumbs, for example `Editor > General > Auto Import`; the arrows at the top right step back and forward through the pages you have visited.

The search box filters the whole tree as you type. Type `Font` and the tree collapses to the pages that mention it (in one screenshot: Appearance, Notifications, Data Editor and Viewer, Editor > Font, Color Scheme, Inspections, Reader Mode, Markdown, Logcat, Terminal and Advanced Settings), and a matching setting on a page is highlighted. The same trick works from outside the dialog: Search Everywhere (`Shift Shift`) lists individual settings pages next to files and actions. **When you do not know which page holds a setting, search; do not browse.**

The Shape of the Tree

The top-level groups in the tree on Rabbit 2, with the pages inside the two you will use most, are below. A `●` beside a page in the dialog just marks a page you have changed from its default.

```
Settings
├── Appearance & Behavior
│   ├── Appearance
│   ├── Menus and Toolbars
│   ├── System Settings
│   ├── File Colors, Scopes, Notifications
│   ├── Data Editor and Viewer, Quick Lists
│   ├── Required Plugins, Trusted Locations, Path Variables
│   └── Presentation Assistant
├── Keymap
├── Editor
│   ├── General  (Auto Import, Appearance, Breadcrumbs, Code Completion, Code Folding,
│   │             Console, Editor Tabs, Gutter Icons, Postfix Completion, Smart Keys, St
│   ├── Code Editing
│   ├── Font
│   ├── Color Scheme
│   ├── Code Style  (Java, Kotlin, C/C++, CMake, EditorConfig, Groovy, HTML, JSON, Markd
│   │              Properties, Shell Script, TOML, XML, YAML ...)
│   ├── Inspections, File and Code Templates, File Encodings
│   ├── Live Templates, File Types, UI Tools, Copyright
│   └── Inlay Hints, Emmet, Intentions, Language Injections, Live Edit,
│       Natural Languages, Reader Mode, TextMate Bundles, TODO
├── Plugins
├── Version Control
├── Build, Execution, Deployment
├── Languages & Frameworks
├── Tools
├── Backup and Sync
├── Advanced Settings
├── Other Settings
├── Experimental
└── Studio Labs
```

Two of those groups have a chapter of their own later: **Version Control** belongs with Chapter 10 and **Studio Labs** with Chapter 11. **Build, Execution, Deployment** is where Gradle settings live

(Chapter 4), and **Tools** holds Logcat and Terminal (Chapters 7 and 1).

Appearance: Theme, Menus and Window Chrome

Appearance & Behavior > Appearance is the first page to visit. The **Theme** drop-down at the top chooses the look of the whole IDE. Two things from the docs and your install are worth knowing: the **Islands** themes (for example **Islands Dark**) are the New UI's themes, with clear separation between the panels, and **Sync with OS** makes the IDE follow your operating system's light or dark mode, switching on its own if Windows switches day and night. The docs also describe a "different tool window background" option that appears only for Islands themes.

Further down, under **UI Options**, the screenshot showed these (with the state on that install):

SETTING	STATE SEEN	WHAT IT DOES
Compact mode	off	Smaller spacing, so UI elements take up less screen space.
Always show full path in window header	off	Puts the full project path in the title bar.
Use project colors in main toolbar	on	Gives each project a toolbar tint, so you can tell windows apart at a glance (the purple "C" next to the project name).
Keep popups open for toggle items	on	Lets you tick several items in a menu without it closing.
Main menu	Hide under Hamburger Button	The three-way choice from Chapter 1.
Smooth scrolling, mnemonics, icons in menu items	on	Cosmetic and keyboard conveniences; the underlined letters in menu names are the mnemonics.

Below that sit the **Tool Windows** options from Chapter 1 and **Presentation Mode**, whose **Zoom** setting (175% on the install checked) is the zoom used when you switch to Presentation Mode for a talk or a screen recording. It is not the zoom for everyday work: for that, use the font settings next.

> **COMPARED WITH THE CLASSIC UI**

> Android Studio: The IDE Itself 1 shows the older layout and themes. The page is in the same place (**Appearance & Behavior > Appearance**); what changed is the list of themes and the extras that belong to the New UI, such as the main menu choice and the tool window options. Older Android Studio's default dark theme was called Darcula (from general knowledge, not rechecked this chapter).

Fonts and Zoom

Text size is under **Editor > Font**. The screenshot showed a **Size** of `13.0` and a **Line height** of `1.2`, an **Enable ligatures** check-box (off), a **Typography Settings** section with **Main weight** (Regular), **Bold weight** (Bold, marked Recommended) and a **Fallback font** (None, used for symbols the main font does not have), and a long list of OpenType **Character variants** such as Slashed Zero. A preview pane on the right shows a paragraph of sample text in the font and the settings you have chosen, so you see the result before you press OK. The font family drop-down sits just above the part of the page in the screenshot.

For quick changes without opening Settings there are three routes:

ROUTE	HOW
Mouse wheel	Tick **Change font size with Ctrl+Mouse Wheel in** (**Editor > General > Mouse Control**; it was off on the install checked), then choose **Active editor** or **All editors**.
Actions	Search Everywhere listed **Increase Font Size**, **Decrease Font Size** and **Reset Font Size** (under the View menu, Active Editor), and the "in All Editors" versions.
Settings page	**Editor > Font**, for a permanent change with the live preview.

Editor > General: How Typing and Scrolling Feel

Editor > General is a long page of small behaviours. The sections visible in the screenshot were:

SECTION	WHAT IT CONTROLS (STATE SEEN)
Mouse Control	Font size with Ctrl+Mouse Wheel (off); move code fragments by drag-and-drop (on; hold Ctrl to copy).
Soft Wraps	Wrap long lines on screen without changing the file, per file pattern such as `*.md; *.txt; *.rst; *.adoc` (off). Wrapped lines keep their original indent (on).

SECTION	WHAT IT CONTROLS (STATE SEEN)
Virtual Space	Whether the caret may sit after the end of a line or inside tabs (both off).
Scroll Offset	How many lines or columns stay visible around the caret as you scroll (vertical 1, horizontal 3).
Caret Movement	How Ctrl+Arrow jumps between words and what Home and End do on a wrapped line (IDE defaults).
Scrolling	Enable smooth scrolling (on).

Editor > General > Appearance

This sub-page decides what the editor shows around your code. The defaults on the install checked are the ones most people keep, but two are worth a look:

SETTING	STATE SEEN	WHY YOU MIGHT CHANGE IT
Show line numbers	on, *Absolute*	The drop-down offers other numbering styles; absolute numbers match error messages and Logcat stack traces.
Show indent guides	on	The faint vertical lines that help in deeply nested Compose code.
Show whitespaces	off	Turn it on, with its Leading, Inner, Trailing and Selection options, when a mix of tabs and spaces is causing trouble.
Highlight occurrences of selected text	on	Marks every other use of the word you have selected.
Render documentation comments	off	Shows comment blocks as formatted text instead of raw `/** */`.
Show intention bulb	on	The lightbulb that offers quick fixes (Chapter 9).
Caret blinking, smooth caret blinking and movement	500 ms; smooth options off and tagged *New*	Pure taste; the "New" tags show these are recent additions.

Auto Import

Editor > General > Auto Import matters more to Android developers than any other editor page, because Compose code needs dozens of imports. The page has a block per language. The Java and Kotlin blocks on the install checked read:

LANGUAGE	SETTING	STATE SEEN
Java	Show auto-import tooltip for Classes, and for Static methods and fields	both on
Java	Insert imports on paste	*Always*
Java	Add unambiguous imports on the fly	off
Java	Optimize imports on the fly	off
Kotlin	Add unambiguous imports on the fly	off
Kotlin	Optimize imports on the fly	off

"Unambiguous" means only one class of that name could be meant, so the IDE can add the `import` line itself without asking. With the Kotlin option off (the default here), you press `Alt+Enter` on a red name and choose the import. Turning **Add unambiguous imports on the fly** on for Kotlin saves a great many of those key presses; turning **Optimize imports on the fly** on also removes unused imports as you go, which keeps files tidy but can surprise you when a temporarily unused import disappears. Both are personal taste, and neither changes what the code does. The same page has lists to force-include or exclude particular classes or packages from auto-import, which is useful if a class you never want (a second `Date` class, say) keeps appearing at the top of the list.

Code Folding

Editor > General > Code Folding controls the small arrows in the margin that collapse a block, and what starts collapsed when you open a file. The screenshot showed **Show code folding arrows** set to *On mouse hover*, and under **Fold by default**: **File header** and **Imports** on (which is why the long import list at the top of a file is hidden behind a single line), **Documentation comments**, **Method bodies** and **Custom folding regions** off, and under Android, **String references** on (a reference to a string resource is shown as the string's text). The Java, Kotlin and C/C++ sections continue below. Tick **Method bodies** if you like to open a file and see only an outline of it.

Code Style

Editor > Code Style decides how the IDE formats code when you ask it to. The top of the page shows the **Scheme** (here *Project*, a scheme that belongs to this project; a gear beside it manages schemes). Under it is a warning, *Settings may be overridden by Indents Detection* with a **Disable** link: when the IDE sees that a file already uses a particular indent, it follows the file. The **General** tab then showed:

SETTING	STATE SEEN	NOTE
Line separator	System-Dependent	Applies to new files only (a Windows machine writes `\r\n`).
Hard wrap at	100 columns	The column the formatter wraps at, and where a visual guide can be drawn.
Wrap on typing	off	Whether the editor breaks a line for you as you type past the column.
Visual guides	empty (optional)	Vertical lines at the columns you list, one (`80`) or several (`80, 120`).
Detect and use existing file indents for editing	on	The cause of the Indents Detection warning above.
Enable EditorConfig support	on	An `.editorconfig` file in the project can override these IDE settings.

Beneath **Code Style** sits one page per language: Java, Kotlin, C/C++, CMake, EditorConfig, Gradle Declarative Configuration Language, Groovy, HTML, JSON, Markdown, Mermaid, Properties, Shell Script, TOML, XML, YAML and Other File Types. For Android work the page you will actually open is **Kotlin**. Because the scheme is *Project*, your choices travel with the project instead of staying on your machine.

Backup and Sync, and the Rest of the Tree

Backup and Sync has its own top-level page in the tree, and the View > Tool Windows list (Chapter 1) has a matching **Backup and Sync History** window. The pages are meant for saving your IDE settings to an account so a second machine can share them. **I could not check what the page offers on this install**, so treat that as the name of a feature to explore, not as

something this chapter has described. The same caution applies to **Experimental**, **Advanced Settings** and **Studio Labs**, which are listed in the tree: Studio Labs gets its own chapter (Chapter 11).

CHANGE ONE THING AT A TIME

A Settings change takes effect when you press **Apply** or **OK**. If a change makes things worse, **Cancel** before you apply it, or change it back and apply again. The pages keep a mark beside any page you have edited, so you can find your own changes later by scanning the tree for the dot.

Try It Yourself

EXERCISE 1

Use the Settings search box (not the tree) to find the page **Editor > Font**. Increase the font **Size** by two points, watch the preview pane change, apply it, then set it back. Then find the **Change font size with Ctrl+Mouse Wheel** option, turn it on for **Active editor**, and use it in a file.

View solution

EXERCISE 2

Open **Editor > General > Auto Import** and write down the state of the four Kotlin and Java *on the fly* options. Turn on **Add unambiguous imports on the fly** for Kotlin, then in a scratch Kotlin file type a line that uses `LocalDate.now()` and say whether the `import` line appeared by itself. Then decide whether to keep the setting.

View solution

EXERCISE 3

In **Editor > Code Style**, note the **Scheme** and the **Hard wrap at** value. Set **Visual guides** to `80, 100`, apply, and look at a long line in the editor. Then open **Editor > Code Style > Kotlin** and list the tabs you see there.

View solution

NOT YET VERIFIED

Checked on the install (screenshots of Android Studio Rabbit 2, 2026-11-20 and 2026-11-21): the Settings tree and its group names; the Appearance page's UI Options, Main menu choices, Tool Windows options and Presentation Mode zoom; the Editor > General page and its Appearance sub-page; Auto Import; Code Folding (the top part); Font (the top part); Code Style General tab and its language list; the Settings search result for *Font*.

Checked in the docs (one web search): the result was JetBrains documentation for IntelliJ IDEA and sister IDEs, not an Android Studio page. It says *Sync with OS* follows the operating system's light or dark mode and can switch with the day and night schedule, describes Islands Dark as the default dark theme with clear separation of UI sections, and describes a tool window background option for Islands themes only. I read search summaries, not the full pages.

Not verified: the contents of the Theme drop-down and the theme names (taken from my earlier audit notes of your screenshots, not re-seen this chapter); the font family drop-down; the lower parts of Code Folding (Java, Kotlin) and Code Style (Formatter tab and the per-language pages); what Backup and Sync, Experimental and Advanced Settings offer; whether Code Style's *Project* scheme is stored in a project file; the behaviour of the exercises (I did not run them). The Darcula remark is from general knowledge. Defaults change between releases, so trust your own Settings over this text where they differ.

COMING UP

Chapter 3 covers new projects: the two-screen New Project wizard, Empty Activity (Compose) against the Views templates, the Android and Project views, Resource Manager and why there is no `res/layout` folder.

Android Studio 2: The New UI

Course 2 · Chapter 3 · Templates and Project Views

Every project starts at the **New Project** wizard, and the choice you make on its first screen decides what the rest of your project looks like: Jetpack Compose or the older XML Views, whether there is a `res/layout` folder, whether C++ is involved. The wizard is shorter than it used to be, so this chapter is quick to read, then spends its second half on what you see *after* Finish: the **Project** tool window and its many views, and the **Resource Manager**.

> **WHAT WAS CHECKED, AND HOW**
>
> **Checked on a real install.** The wizard's two screens, the full template list, the Compose and Views details pages, the Project view drop-down, the Android view and the Resource Manager come from screenshots of Android Studio Rabbit 2 (stable, 2026.2.2) taken on 2026-11-20 and 2026-11-21, plus the folder layout of a generated Compose project.
>
> **Checked in the docs.** One web search against Google's *Create a project* page (see the Not Yet Verified box).
>
> **Not run by me.** The values in the exercises are examples; names such as `com.example.scratch` are placeholders, not your own.

Starting the Wizard

From the Welcome screen choose **New Project**. With a project already open, use **File > New > New Project...** (the File menu's **New** entry opens a sub-menu, as seen in Chapter 1). The wizard is a dialog with **two screens**, driven by **Previous**, **Next**, **Cancel** and **Finish** buttons: a template page, then one page of details. Next is available on the first screen; Finish only on the second.

Screen 1: Choose a Template

The left of the dialog lists categories: **Create with AI...** (a separate entry at the top), then **Phone and large screens** (selected by default), **Wear OS**, **Television**, **Car** and **XR**. The right shows the templates for the selected category as a grid of phone-shaped pictures; the list scrolls. For **Phone and large screens** the install checked offered ten:

TEMPLATE	WHAT THE SCREENSHOTS SHOW
No Activity	A dashed empty box: a project with no screen at all.
Empty Activity	The default, with the Compose logo. Its details page reads *Create a new empty activity with Jetpack Compose*. **This is the template this course and the Compose-first Android courses use.**
Gemini API	Tagged **Preview**: a starter for apps that call Google's Gemini models.
Navigation UI Activity	Shows the Compose logo above a bottom navigation bar. I did not open it to confirm it is Compose.
Basic Views Activity	Name says Views: the XML-based path, with a floating action button in its picture.
Empty Views Activity	Details page reads *Creates a new empty activity*, with a **Language** field.
Responsive Views Activity	A picture of a layout that adapts between small and large screens.
Game Activity (C++)	A game-controller picture: a native C++ game starter.
Native C++	A C++ logo: an app with native code.
Kotlin Multiplatform (Install Plugin)	The label says a plugin must be installed first.

The point to take from this list is the naming. **The XML-based templates now say "Views" in their names**, so choosing XML is an explicit decision rather than the default. Everything built later in this course series assumes **Empty Activity**.

Screen 2: Name It and Set the Basics

The second screen is headed with the template you chose and a one-line description, then four fields:

FIELD	WHAT IT IS
Name	The app's name, shown under its icon. Also used for the project folder.
Package name	The app's unique identifier, written like a reversed web address (for example `com.example.scratch`). It is the ID the Play Store and the phone use for your app, so it is hard to change after you publish.

FIELD	WHAT IT IS
Save location	The folder for the project, with a folder-picker button.
Minimum SDK	The oldest Android version the app will run on. The install checked showed **API 24 ("Nougat"; Android 7.0)** and, under the field, *Your app will run on approximately 99.5% of devices.*

Below the fields is a **Help me choose** link: per Google's docs it opens a dialog with the cumulative distribution of Android versions, so you can see how many devices each minimum level reaches. The trade-off is simple: a lower minimum reaches more devices but gives you fewer modern APIs, a higher one the opposite. Leave the default unless you have a reason.

> **THE WHITESPACE WARNING**
>
> Both details pages showed a warning: *Save location should not contain whitespace, as this can cause problems with the NDK tools.* The NDK is the toolkit for native C++ code. A Kotlin and Compose app does not use it, so a space in the folder name (for example `Projects 2026`) is probably harmless today, but a path with no spaces costs nothing and keeps you safe if the project ever adds C++. I could not test whether a space causes any problem in a Compose-only project.

What is missing from the page

Two choices that older Android Studio versions asked about do **not** appear on the Compose page. There is **no Language field** (a Compose app is Kotlin) and there is **no build configuration choice** between Kotlin and Groovy for the Gradle files. The generated project uses Kotlin and the Kotlin DSL (Chapter 4). The *Views* template's details page, however, **does** have a **Language** drop-down (it showed Kotlin); neither template page had a build-language choice. That the older wizard asked for both is from general knowledge and was not rechecked.

Press **Finish**. Android Studio creates the files, then runs a **Gradle sync** and indexes the project. Progress shows in the **Build** tool window at the bottom (its **Sync** tab) and the first sync can take a while because it downloads what the project needs. Chapter 4 explains what sync is doing.

The Project Tool Window and Its Views

The **Project** tool window (`Alt+1`, the folder icon at the top of the left bar) shows your files, but not always in the same way. At the top-left of the panel is a drop-down that names the current **view**, and there are ten to choose from. The install checked offered:

VIEW	WHAT IT SHOWS
Android	The default for an Android project: a tidied view grouping the manifest, source code, resources and build scripts (below).
Project	The real folder structure on disk, including the `build` output folder and hidden configuration. Use it to see exactly where a file lives.
Packages	Source files grouped by package only.
Project Files	Files that belong to the project, grouped by location.
Production	Only the code that goes into the shipped app.
Tests	Only test code.
Project Source Files	Only source files, without other project files.
Project Non-Source Files	Everything that is not source code.
Open Files	The files you currently have open in the editor.
Scratches and Consoles	Scratch files and consoles that are not part of the project.

Only the **names** of the views were visible in the screenshots; the one-line descriptions are my reading of those names and of how the IntelliJ platform works, not something I checked on the install. For daily work you will use only two: **Android** for writing code and **Project** for looking at the truth on disk.

The Android view

In the Android view the top level of a project looks like this (names as seen on the install checked):

```
// Project tool window, Android view
app
├── manifests
├── kotlin+java
│   ├── com.example.scratch
│   ├── com.example.scratch (androidTest)
│   └── com.example.scratch (test)
├── res
```

```
    └── keepRules
  Gradle Scripts
```

The three entries under `kotlin+java` are not three projects. They are three **source sets** with the same package name: the app's code, the `androidTest` code that runs on a device or emulator, and the `test` code that runs on your computer. The Android view labels the test ones in brackets and greys them. **Gradle Scripts** collects the build files, Chapter 4's subject.

> **WHERE DID RES/LAYOUT GO?**
>
> A project from the Compose template has **no** `res/layout` **folder**: the screen is written in Kotlin, not XML. The `res` folder of the generated project held only `drawable`, `mipmap-*` (the launcher icons), `values` and `xml`. A Views project starts with layout files, which is the second big difference between the two template families. If you open an old tutorial and it tells you to edit a layout XML file, it was written for a Views project.

The Resource Manager

The second icon on the left bar is the **Resource Manager**, a tool window for the app's images, colours and strings as a gallery instead of a folder tree. It has a **Module** drop-down at the top (for example *Module: Catalogue*), a search box with a filter button, and **plus** and **refresh** buttons. A row of tabs filters by kind: **Drawable, Color, Layout, Mip Map, String**. Under them, resources are grouped by module and source set (for example *Catalogue.app.main (2)*, the two resources it holds) and shown as thumbnails with their name and the number of *versions* (alternates for different screen densities or modes). A new project's Drawable tab showed `ic_launcher_background` and `ic_launcher_foreground`, the two halves of the launcher icon. Buttons at the bottom switch between a list and a grid and zoom the thumbnails.

The Resource Manager is the quickest way to *see* what resources you have and to import a new image (the plus button), and it is the New UI's tidier replacement for hunting through `res` by hand. In a Compose app you will use it far less than in a Views app, because much of the visual design is code.

> **COMPARED WITH THE CLASSIC UI**
>
> Android Studio: The IDE Itself 3 teaches the Layout Editor and XML layouts as the main design path, and the older New Project wizard asked you for a Language and a build language. In the New UI path used here the wizard is two screens, the default template is Compose and there is no layout folder to

edit. Chapter 5 of this course covers **Compose Preview**, which replaces the Layout Editor for Compose code.

Try It Yourself

EXERCISE 1

Create a scratch project: choose **Empty Activity**, name it `Scratch`, use the package `com.example.scratch`, a **Save location with no spaces** and the default **Minimum SDK**. Before you click Finish, press **Help me choose** and write down what it shows. Afterwards, note the **Name**, **Package name** and **Minimum SDK** you used and what the Android view shows at its top level.

View solution

EXERCISE 2

In the scratch project, switch the Project tool window through the **Android**, **Project**, **Packages** and **Project Files** views. For each, find `MainActivity.kt` and the manifest and write down the path of folders you had to open. Then say which of the two views (Android or Project) shows the `build` folder.

View solution

EXERCISE 3

Open the **Resource Manager**, list the five tabs and the two resources on the **Drawable** tab, then find the same files in the **Project** view under `app/src/main/res`. Confirm there is no `res/layout` folder. Optionally, create a second scratch project from **Empty Views Activity** and compare: does it have a **Language** field, and does it have a `res/layout` folder?

View solution

NOT YET VERIFIED

Checked on the install (screenshots of Android Studio Rabbit 2, 2026-11-20 and 2026-11-21, and the generated project's folders): the wizard's two screens and buttons; the category list; all ten templates and the two details pages (Empty Activity, Empty Views Activity), including the Name, Package name, Save location, Minimum SDK and Language fields, the 99.5% line and the whitespace

warning; the ten Project view names; the top level of the Android view; the Resource Manager's controls, tabs and contents; the folders under `res`.

Checked in the docs (one web search): Google's English *Create a project* page says Empty Activity is the recommended starting point for new Compose projects, that Help me choose opens a dialog showing the cumulative distribution for the selected API level, and that a lower minimum API level means fewer modern APIs but more devices. It also describes Kotlin as the recommended language. I read search summaries, not the whole page, and did not check whether the default minimum API level in other releases differs from 24: older and third-party tutorials show different examples.

Not verified: what each Project view other than Android actually lists (the descriptions are my reading of the names); what the other templates generate; whether Navigation UI Activity is Compose; the exact contents of the Gemini API template; what Create with AI... does; the Wear OS, Television, Car and XR template lists; whether a space in the save location causes any problem for a Compose-only project; and the exercises (not run by me). The statement that older wizards asked for a language and a build language is from general knowledge. Defaults change between releases, so trust your own wizard where it differs.

Source: Create a project (developer.android.com).

COMING UP

Chapter 4 opens the Gradle Scripts group: Kotlin DSL build files, the version catalog, the Compose BOM, the Android Gradle Plugin and Gradle versions, the Project Structure dialog and what sync actually does.

Android Studio 2: The New UI

Course 2 · Chapter 4 · Reading the Build Files

Every Android project is built by **Gradle**, a build tool that reads a handful of script files, downloads the libraries you ask for and turns your code into an app. You will never get far in Android without opening those files, and they are the part of the IDE that has changed most since older tutorials were written: the scripts are now **Kotlin** files (`.kts`), versions live in a separate **version catalog**, and Android's Gradle plugin is at version 9. This chapter reads a freshly generated Compose project's build files line by line, then shows the two dialogs that show the same information graphically: **Project Structure** and **Run/Debug Configurations**.

> **WHAT WAS CHECKED, AND HOW**
>
> **Checked on a real install.** The files below are those of a project generated by Android Studio Rabbit 2 (stable, 2026.2.2) from the Empty Activity template, read from disk on 2026-11-21, and the screenshots are of the Gradle Scripts group, the Project Structure dialog (every page except Suggestions) and the Run/Debug Configurations dialog. I replaced the project's own name and package with `MyApp` and `com.example.myapp`; the version numbers are exactly as generated.
>
> **Checked in the docs.** One web search against Google's *Android Gradle plugin 9.0* release notes (see the Not Yet Verified box).
>
> **Not run by me.** I did not edit a build file or sync a project myself; the exercises are untested.

What Gradle, AGP and Sync Are

Three names appear everywhere in this chapter, so fix them now:

NAME	WHAT IT IS
Gradle	The general build tool. It runs **tasks** (compile, test, package) described in build scripts. The version used is set by the *wrapper* (below): 9.6.0 in the project checked.
AGP (Android Gradle Plugin)	A plugin for Gradle that knows how to build an Android app: resources, manifest, packaging, signing. Its version in this project is **9.5.1**.
Sync	Android Studio asking Gradle to read the build files and report the project's shape (modules, dependencies, source folders), so the IDE can offer correct code completion and error

NAME	WHAT IT IS
	checking. Press it after you edit a build file: **File > Sync Project with Gradle Files**, `Ctrl+Shift+O` in Chapter 1's File menu list. Its progress shows on the **Sync** tab of the **Build** tool window.

The Gradle Scripts Group

In the Android view (Chapter 3) the bottom entry is **Gradle Scripts**. It collects seven files, each with a plain-English label in brackets, which is the quickest way to learn which is which:

FILE	LABEL SHOWN	ITS JOB
`build.gradle.kts`	(Project: MyApp)	The top-level build file. In a new project it only declares which plugins the modules may use.
`build.gradle.kts`	(Module :app)	The app's own build file: SDK versions, app ID, build types and dependencies. This is the file you edit most.
`gradle.properties`	(Project Properties)	Project-wide switches and the Gradle process's memory.
`gradle-wrapper.properties`	(Gradle Version)	Which Gradle version the project downloads and uses.
`libs.versions.toml`	(Version Catalog "libs")	The one list of library and plugin versions.
`local.properties`	(SDK Location)	Where the Android SDK is on *this* computer. Machine-specific, so it is normally left out of version control.
`settings.gradle.kts`	(Project Settings)	The project's name, its modules and where Gradle looks for plugins and libraries.

COMPARED WITH THE CLASSIC UI

Android Studio: The IDE Itself 2 shows Groovy files named `build.gradle` with plugin versions such as `8.1.0` written straight into the project-level file, and library lines in quotes. On a new project today the files are Kotlin (`.kts`), plugin and library versions are in `libs.versions.toml`, and the project-level file contains only `alias(...) apply false` lines. The old lesson of "project-level or module-level file?" has a third answer: **the catalog**.

The Version Catalog

`gradle/libs.versions.toml` has three tables. This is the whole file as generated:

```
# gradle/libs.versions.toml
[versions]
agp = "9.5.1"
coreKtx = "1.19.1"
junit = "4.13.2"
junitVersion = "1.3.0"
espressoCore = "3.7.0"
lifecycleRuntimeKtx = "2.11.1"
activityCompose = "1.13.0"
kotlin = "2.2.10"
composeBom = "2026.02.01"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVers
androidx-espresso-core = { group = "androidx.test.espresso", name = "espresso-core", ver
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle", name = "lifecycle-runti
androidx-activity-compose = { group = "androidx.activity", name = "activity-compose", ve
androidx-compose-bom = { group = "androidx.compose", name = "compose-bom", version.ref =
androidx-compose-ui = { group = "androidx.compose.ui", name = "ui" }
androidx-compose-ui-graphics = { group = "androidx.compose.ui", name = "ui-graphics" }
androidx-compose-ui-tooling = { group = "androidx.compose.ui", name = "ui-tooling" }
androidx-compose-ui-tooling-preview = { group = "androidx.compose.ui", name = "ui-toolin
androidx-compose-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-test-man
androidx-compose-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-test-junit
androidx-compose-material3 = { group = "androidx.compose.material3", name = "material3"

[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "kotlin" }
```

Read it in three steps:

1. `[versions]` gives each version a name. Change a number here and every library or plugin that refers to it follows.

2. `[libraries]` names each library: a `group` and a `name` (together, its Maven address) plus `version.ref`, which points at a line in `[versions]`.
3. `[plugins]` does the same for Gradle plugins, using an `id`.

Notice that the Compose libraries (`ui`, `ui-graphics`, `material3` and the rest) have **no version at all**. Their versions are supplied by `androidx-compose-bom`, the **Compose BOM** ("bill of materials"): one version number (`composeBom = "2026.02.01"`) that fixes a matching version for every Compose library, so they always fit together.

The naming rule. A dash in a catalog name becomes a dot in the build file. `androidx-core-ktx` is written `libs.androidx.core.ktx`, and the plugin `kotlin-compose` is `libs.plugins.kotlin.compose`. Android Studio completes these names for you once you type `libs.`.

The Three Kotlin Build Files

settings.gradle.kts

```
// settings.gradle.kts
pluginManagement {
    repositories {
        google {
            content {
                includeGroupByRegex("com\\.android.*")
                includeGroupByRegex("com\\.google.*")
                includeGroupByRegex("androidx.*")
            }
        }
        mavenCentral()
        gradlePluginPortal()
    }
}
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention") version "1.0.0"
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
```

```
    }
}

rootProject.name = "MyApp"
include(":app")
```

`pluginManagement` says where Gradle finds *plugins* (Google's repository, but only for groups starting `com.android`, `com.google` or `androidx`, then Maven Central and the Gradle plugin portal). The `foojay-resolver-convention` plugin lets Gradle download a Java toolchain it is missing. `dependencyResolutionManagement` says where modules find *libraries* (Google and Maven Central), and `FAIL_ON_PROJECT_REPOS` makes the build fail if a module tries to declare its own repositories, which keeps every module on the same list. `rootProject.name` is the project name and `include(":app")` registers the one module.

build.gradle.kts (Project)

```
// build.gradle.kts (Project: MyApp)
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.kotlin.compose) apply false
}
```

Two lines. `alias(...)` refers to a plugin in the catalog, and `apply false` means "make this plugin available to the modules, but do not apply it to the root project itself". The versions are not here any more: they are in the catalog.

build.gradle.kts (Module :app)

```
// app/build.gradle.kts
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.compose)
}

android {
    namespace = "com.example.myapp"
    compileSdk {
        version = release(37)
    }
```

```
    defaultConfig {
        applicationId = "com.example.myapp"
        minSdk = 24
        targetSdk = 37
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            optimization {
                enable = true
                packageScope = setOf("androidx.**", "kotlin.**", "kotlinx.**")
            }
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_11
        targetCompatibility = JavaVersion.VERSION_11
    }
    buildFeatures {
        compose = true
    }
}

dependencies {
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.activity.compose)
    implementation(libs.androidx.compose.material3)
    implementation(libs.androidx.compose.ui)
    implementation(libs.androidx.compose.ui.graphics)
    implementation(libs.androidx.compose.ui.tooling.preview)
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    testImplementation(libs.junit)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.compose.ui.test.junit4)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(libs.androidx.junit)
    debugImplementation(libs.androidx.compose.ui.test.manifest)
```

```
    debugImplementation(libs.androidx.compose.ui.tooling)
}
```

BLOCK OR LINE	WHAT IT DOES
`plugins { alias(...) }`	Applies the two plugins to this module. **There is no `kotlin-android` plugin**: see the next section.
`namespace`	The package name used for the generated `R` class and the code.
`compileSdk { version = release(37) }`	The Android version the code is compiled against. A new form: it replaces the older `compileSdk = 34`.
`applicationId`, `minSdk`, `targetSdk`	The app's identity on a device (what you chose in the wizard), the oldest Android it supports (24) and the version it is tested for (37).
`versionCode`, `versionName`	The number the store compares and the label users see. You raise these for each release.
`buildTypes { release { optimization { ... } } }`	Settings for the `release` build: `enable = true` turns on code shrinking and optimisation for the listed packages. This replaces the older `isMinifyEnabled` line plus ProGuard file names.
`compileOptions`	Java source and target level: Java 11.
`buildFeatures { compose = true }`	Switches on Compose for this module.
`implementation(platform(libs.androidx.compose.bom))`	Brings in the Compose BOM so the Compose libraries below it need no versions.
`implementation` / `testImplementation` / `androidTestImplementation` / `debugImplementation`	The **configuration**: *implementation* for the app, *test* for tests that run on your computer, *androidTest* for tests that run on a device, *debug* only in debug builds (the Compose tooling lives here).

AGP 9: Kotlin Is Built In

Older Android projects applied a second plugin, `org.jetbrains.kotlin.android`, to compile Kotlin. Look back at the files: it is not there. Google's AGP 9.0 release notes say that **Kotlin support is now built into AGP and on by default**, so an Android app module no longer applies the Kotlin Android plugin. AGP 9.0 brings Kotlin Gradle plugin 2.2.10 with it as a runtime dependency (the same number as `kotlin = "2.2.10"` in the catalog above), and the notes describe a switch, `android.builtInKotlin=false`, to opt out. The one Kotlin plugin the project still lists is `kotlin-compose`, which Compose needs. If you open an older tutorial that adds `id("org.jetbrains.kotlin.android")`, that line is for the pre-9 way of working.

The odd new-looking forms in the module file (`compileSdk { version = release(37) }` and the `optimization { }` block) appear to belong to the **new DSL** that AGP 9 introduced. The docs summary I found says the old Kotlin plugin is not compatible with it; I did not read the DSL page itself, so treat that link between the forms and the new DSL as my inference.

gradle.properties and the Wrapper

`gradle.properties` in the generated project had four live lines (everything else was comments):

```
# gradle.properties (comments removed)
org.gradle.jvmargs=-Xmx2048m -Dfile.encoding=UTF-8
# org.gradle.parallel=true
org.gradle.configuration-cache=true
kotlin.code.style=official
```

LINE	MEANING
`org.gradle.jvmargs=-Xmx2048m ...`	The memory limit (2048 MB) and text encoding of the **Gradle daemon**, the background process that does the building. This is *not* the memory of the IDE itself, which is a separate setting.
`org.gradle.parallel=true`	Commented out (a leading `#`), so off. It lets independent modules build at the same time; the generated comment says to use it only with decoupled projects.
`org.gradle.configuration-cache=true`	On. Gradle can skip its configuration phase when nothing that affects the build configuration has changed, which makes repeat builds faster.

LINE	MEANING
`kotlin.code.style=official`	Tells the IDE to use Kotlin's official code style for this project.

The **wrapper** file, `gradle/wrapper/gradle-wrapper.properties`, names the exact Gradle version. In this project: `distributionUrl=https\://services.gradle.org/distributions/gradle-9.6.0-bin.zip`, with a checksum line and `validateDistributionUrl=true`. The wrapper means anyone who opens the project, on any computer, downloads and uses the same Gradle, so you do not have to install Gradle yourself. `local.properties`, the seventh file, holds only the path to the Android SDK on your machine.

The Project Structure Dialog

Everything above can be edited as text, but Android Studio also offers a form view of the same settings: **File > Project Structure...**, `Ctrl+Alt+Shift+S`. It has three groups of pages in a list on the left, with **Suggestions** below them (a number badge on it showed 2 items in the project checked):

GROUP	PAGE	WHAT YOU SEE
1	Project	**Android Gradle Plugin Version**, shown as `$versions.agp : 9.5.1` (it reads the catalog entry, not a loose number), with a note to use the **AGP Upgrade Assistant** to change it, and **Gradle Version**, here 9.6.0.
1	SDK Location	The **Android SDK location** with a folder button. The text says the location is used for new projects and for existing ones with no `local.properties` containing an `sdk.dir` line.
1	Variables	A tree of variables by source: the build script, the **Default version catalog: libs (libs.versions.toml)** with every `[versions]` entry and its value, the project, and `:app`. A `+New catalog version` row adds one.
2	Modules	The `app` module with tabs **Properties**, **Default Config** and **Signing Configs**. Properties lists Compile Sdk Version (37), Build Tools Version (blank), NDK Version (blank), Source and Target Compatibility (`$JavaVersion.VERSION_11 : 11 (Java 11)`), Enable View Binding (blank) and two "Retain information about dependencies" options.
2	Dependencies	Declared Dependencies for `<All Modules>` or `app`, each with its Configuration (`implementation`, `androidTestImplementation` and so on), a Details pane showing the group, artifact and requested version (`/*not specified*/` for BOM-

GROUP	PAGE	WHAT YOU SEE
		supplied ones), and on the right **Resolved Dependencies** grouped by variant: `debug`, `debugAndroidTest`, `debugUnitTest`, `release`.
2	Build Variants	Tabs **Build Types** (`release` and `debug`) and **Flavors**; for a build type, fields such as Application ID Suffix, Version Name Suffix, Debuggable, Signing Config, Minify Enabled, ProGuard Files, Manifest Placeholders, Multi Dex Enabled and Matching Fallbacks.
3	Suggestions	Not photographed: the badge showed that two suggestions were waiting.

ONE THING THAT DOES NOT MATCH

The Build Variants page showed **Minify Enabled: false** for the `release` build type, while the module file has `optimization { enable = true }` for release. The dialog's older field names (Minify Enabled, ProGuard Files, Renderscript) look like the pre-9 DSL, so I suspect it does not yet reflect the new `optimization` block. I have not confirmed that. **When the dialog and the file disagree, believe the file**, and be careful about editing a setting in the dialog that your build file now sets in a different way.

The Run Configuration

In the main toolbar (Chapter 1) are a **device selector** (on the install checked, *Medium Phone API 37.0*), a **run configuration** drop-down reading *app*, and the green **Run** and **Debug** buttons. A **run configuration** is a saved recipe for starting something: which module, how to install it, which screen to launch. Open the drop-down and look for **Edit Configurations...** (I did not photograph that menu; the dialog it opens is titled **Run/Debug Configurations**) to see them. The dialog has a tree on the left with buttons to add, remove, copy, make a folder and sort, and an **Edit configuration templates...** link. The project checked had an **Android App** group containing four entries: `androidTest`, `app`, `main` and `unitTest`.

The `app` configuration, the one Run uses, showed:

FIELD	VALUE SEEN
Name	`app`, with **Allow multiple instances** ticked and **Store as project file** unticked
Tabs	General, Miscellaneous, Debugger
Module	the app's module

FIELD	VALUE SEEN
Deploy	Default APK
Install options (all unticked)	Install for all users (if already installed, will only update for existing users); Always install with package manager (disables deploy optimizations on Android 11 and later); Clear app storage before deployment
Install Flags	empty: *Options to 'pm install' command*
Launch	Default Activity
Launch Flags	empty: *Options to 'am start' command*

A **Before launch** section sits below, listing steps to do first (normally a Gradle build). The defaults are right for almost every project: you change a run configuration when you need to launch a different screen, clear stored data on each run or pass flags to the installer.

A BROKEN CONFIGURATION

The `unitTest` entry showed a red error marker and the message *Error: Module not specified*: its Module field was empty. It was listed under **Android App**, not a test group, so it was probably created by accident. A run configuration like that shows up in the toolbar drop-down and does nothing useful: delete it with the minus button, or select it and set a module. How it was created is not known.

Try It Yourself

EXERCISE 1

In a project (the scratch project from Chapter 3 will do) open each of the seven files in **Gradle Scripts** and write down its name, the label in brackets and its job in your own words. Then open `libs.versions.toml` and say how many tables it has, which libraries have *no* version, and why.

View solution

EXERCISE 2

Open **Project Structure** (`Ctrl+Alt+Shift+S`). From the **Project** page read the AGP and Gradle versions; from **Variables** read the value of `composeBom`; from **Dependencies** pick one dependency and read its configuration. Then find the same three facts in the files. Change nothing, and press **Cancel**.

View solution

EXERCISE 3

Add a dependency the catalog way: put a new library line in `libs.versions.toml`, use it in the module file, sync, and find it on the **Dependencies** page. Then remove it again and sync. Use the Compose extended icons library (a good candidate because the Compose BOM supplies its version).

View solution

NOT YET VERIFIED

Checked on the install (Android Studio Rabbit 2, files read from disk on 2026-11-21, screenshots on 2026-11-20 and 2026-11-21): all file contents and version numbers shown; the Gradle Scripts group and its labels; the Project Structure pages Project, SDK Location, Variables, Modules, Dependencies and Build Variants; the Run/Debug Configurations dialog for `app`; the `unitTest` error; the device selector and run configuration in the toolbar.

Checked in the docs (one web search): Google's *Android Gradle plugin 9.0* release notes say Kotlin support is built in and on by default so the Kotlin Android plugin is no longer applied, that AGP 9.0 brings Kotlin Gradle plugin 2.2.10 as a runtime dependency, that `android.builtInKotlin=false` opts out, and that the Kotlin Android plugin is not compatible with the new DSL. I read search summaries, not the full pages, and the search said nothing about version catalogs.

Not verified: that the `compileSdk { ... }` and `optimization { ... }` forms are the new DSL (my inference); why the Build Variants page shows Minify Enabled false; the Suggestions page; what the two Suggestions are; the Debugger and Miscellaneous tabs of the run configuration; the `main` and `androidTest` configurations; how the `unitTest` configuration was created; the Compose extended icons coordinates in the exercise (general knowledge); the exercises (not run by me); and the IDE's own memory setting, which is separate from `org.gradle.jvmargs` and not covered. Versions in these files change with every Android Studio release, so trust your own project over the numbers here.

Sources: Android Gradle plugin 9.0 release notes (developer.android.com).

COMING UP

Chapter 5 turns to Compose Preview and the Compose-first editor: `@Preview`, the preview pane, interactive and on-device previews, and what replaced the Layout Editor.

Android Studio 2: The New UI

Course 2 · Chapter 5 · Seeing Your UI Without Running It

In a Compose project there is no `res/layout` folder (Chapter 3), so there is no XML file to open in the old **Layout Editor**. The screen is Kotlin code, and the way you see it without installing the app on a phone is **Compose Preview**: you mark a function with `@Preview` and Android Studio draws it beside your code. This chapter explains how to write previews, the tools that go with them (several previews from one line, interactive mode, running a preview on a device, UI Check) and how this differs from the Layout Editor you may have read about.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **This is the least verified chapter so far.** I have **no screenshots** of Compose Preview on Rabbit 2, so I could not check any button label, menu name or layout of the preview pane on the install. **Checked on the install:** only the facts that appear elsewhere in this course's files and screenshots: the project's dependencies, the `@Preview` import in a generated `MainActivity.kt` and the existence of a **Layout Validation** tool window in the View menu.
> **Checked in the docs:** two web searches against Google's *Preview your UI with composable previews* page and its debugging page. What they said is marked *(docs)* below.
> **General knowledge, not checked:** marked *(general knowledge)*. When you reach a screenshot-based version of this chapter, trust the screenshots over these notes.

Writing a Preview

A preview is a Kotlin function with two annotations: `@Composable` (it draws UI) and `@Preview` (draw it in the IDE). It takes no parameters, and it normally wraps your screen in your app's theme so the colours and fonts match. A generated project contains one:

```
// MainActivity.kt (shape of the generated preview)
import androidx.compose.ui.tooling.preview.Preview

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MyAppTheme {
        Greeting("Android")
```

```
    }
}
```

`showBackground = true` paints a white background behind the preview; without it the preview is transparent, which can make light text invisible. `MyAppTheme` is whatever theme function the template created for your project (in the project checked it was in `ui/theme`, next to `Color.kt`, `Theme.kt` and `Type.kt`). Two details connect back to Chapter 4:

- The `Preview` annotation lives in `androidx.compose.ui.tooling.preview`, which comes from the `ui-tooling-preview` library, declared as `implementation(libs.androidx.compose.ui.tooling.preview)`.
- The `ui-tooling` library is declared as `debugImplementation`. My understanding is that this is the library the IDE uses to render and inspect previews and that it is kept out of release builds *(general knowledge)*.

You can have as many `@Preview` functions as you like in a file, and several annotations on one function to see it in different set-ups. Common `@Preview` parameters include `name`, `group`, `widthDp`, `heightDp`, `uiMode` (for dark mode), `fontScale` and `device` *(general knowledge: only `showBackground` appears in the project I looked at)*.

The Preview Pane

When a file contains an `@Preview`, the editor can show the rendered preview next to the code. Android Studio normally gives you a way to switch between code only, preview only and both side by side, and to build and refresh the preview when it goes out of date *(general knowledge: I could not check the names or places of those controls on Rabbit 2)*. Two practical points hold whatever the buttons are called:

- The preview needs the project to **build successfully**. If the file has a compile error (a missing import, a function that no longer exists) the preview cannot draw, so fix the errors in the editor first, or the preview shows an error instead of your UI *(general knowledge)*.
- A preview is a picture of *one function with fixed inputs*. It does not run your app, your database or your network code. Give a screen's composable its data as plain parameters so a preview can pass in sample values, and keep database or view-model work outside the composable you preview *(general knowledge)*.

Several Previews from One Line

(docs) Compose's tooling provides ready-made **multipreview annotations** that show one composable in several configurations at once, from UI tooling 1.6.0-alpha01 onward:

ANNOTATION	WHAT IT ADDS
`@PreviewScreenSizes`	The same UI at several screen sizes.
`@PreviewFontScales`	The same UI at several font scales (small to large text).
`@PreviewLightDark`	The same UI in light and dark mode.
`@PreviewDynamicColors`	The same UI with different dynamic colour schemes.

You can also define your own: an annotation class that carries several `@Preview` annotations with different settings, which then renders every variant wherever you apply it. *(docs)* One catch noted by the docs: stacking several multipreview annotations does **not** produce every combination (for example every screen size in both light and dark); each annotation renders only its own variants.

```
// One composable, previewed in light and dark mode
@PreviewLightDark
@Composable
fun GreetingPreview() {
    MyAppTheme {
        Greeting("Android")
    }
}
```

Interactive Mode and Running a Preview on a Device

A normal preview is a still picture. *(docs)* Two tools make it behave more like the real app:

TOOL	WHAT THE DOCS SAY
Interactive mode	Runs a preview much as it would run on a phone or tablet, in its own isolated sandbox, so you can click elements and type input without affecting other previews.

TOOL	WHAT THE DOCS SAY
Run Preview (deploy to device)	Click the **Run Preview** icon next to the `@Preview` annotation, or at the top of the preview, and Android Studio installs that preview on a connected device or emulator. It launches as a new activity in the same app, so it shares the app's manifest and permissions.

Use interactive mode when you want to try a button or a text field; use the device route when the thing you are checking depends on the real device (a permission, a camera, real hardware behaviour). Chapter 6 covers devices and emulators.

UI Check, and What Happened to Layout Validation

(docs) **UI Check** automatically previews your UI in different configurations and highlights issues found, such as text that is cut off at a large font size or an element that does not fit a small screen. You turn it on from the **UI Check** icon on the preview. It is the quickest way to find accessibility and layout problems before you run anything.

Chapter 1's View > Tool Windows list also contains **Layout Validation**. The docs I found describe that tool for *View-based* (XML) layouts: it shows a layout in many device sizes at once. I found no Compose-specific description of it, so I cannot say what it does for a Compose project. For Compose, UI Check is the documented equivalent. The Problems panel (the **Problems** tool window, `Alt+6`) also reports design-tool issues *(docs)*.

What Replaced the Layout Editor

LAYOUT EDITOR (XML VIEWS)	COMPOSE PREVIEW
You open a layout XML file in a design surface.	You open a Kotlin file containing `@Preview` functions. There is no layout file.
You can drag widgets from a palette onto the screen *(general knowledge)*.	You write or edit code; the preview redraws. The picture is output, not the thing you edit *(general knowledge)*.
One layout file, one screen.	Any composable can have a preview, including small pieces such as a single card or button.
Resizing for different devices through the editor.	Different sizes, font scales and modes through parameters and multipreview annotations *(docs)*.

LAYOUT EDITOR (XML VIEWS)	COMPOSE PREVIEW
Still available for XML projects (Chapter 3's Views templates).	The default for the Empty Activity template.

COMPARED WITH THE CLASSIC UI

Android Studio: The IDE Itself 3 teaches the Layout Editor and XML as the main way to design a screen. For a Compose project that chapter does not apply: the Layout Editor is for XML layouts, which a Compose project does not have. The concepts it taught (constraints, sizing, reading what a screen will look like on several devices) now live in Kotlin code and in the preview annotations above.

Try It Yourself

These exercises are written from the docs, not from screenshots, so each asks you to **report what you actually see**. Your answers are the evidence this chapter lacks.

EXERCISE 1

In the scratch project open `MainActivity.kt` and find the generated `@Preview`. Open the preview pane (write down the name or icon of every control you use to show code, preview or both, and to refresh the preview). Change the greeting text and say how the preview updates. Then remove `showBackground = true` and describe the difference.

View solution

EXERCISE 2

Replace `@Preview(showBackground = true)` with `@PreviewLightDark`, then try `@PreviewFontScales`. Count how many previews each one produces and say what differs between them. Then add both annotations to the same function and check whether you get every combination.

View solution

EXERCISE 3

Add a small composable with a `Button` that counts clicks (a `remember { mutableIntStateOf(0) }` counter), give it an `@Preview`, then try **interactive mode**. Afterwards run the same preview on an

emulator or device with **Run Preview**. Say what each route showed and when you would choose one over the other.

View solution

NOT YET VERIFIED

Checked on the install: only these, and only from other chapters' evidence: the `ui-tooling-preview` (implementation) and `ui-tooling` (debugImplementation) dependencies; the `androidx.compose.ui.tooling.preview.Preview` import and a generated `@Preview(showBackground = true)` function in `MainActivity.kt`; the `ui/theme` files; the Layout Validation and Problems entries in the View menu list.

Checked in the docs (two web searches): Google's *Preview your UI with composable previews* page says multipreview annotations (`@PreviewScreenSizes`, `@PreviewFontScales`, `@PreviewLightDark`, `@PreviewDynamicColors`) exist from UI tooling 1.6.0-alpha01, that custom multipreviews are annotation classes carrying several previews, that stacked multipreview annotations do not combine, that interactive mode runs in an isolated sandbox, that Run Preview installs the preview on a device or emulator as a new activity in the same app, and that UI Check previews the UI in different configurations and highlights issues. The Problems-panel page describes Layout Validation for View-based layouts. I read search summaries, not the full pages.

Not verified at all: the names, icons and positions of the preview pane's controls (code, split and design modes, build and refresh, interactive, run preview, UI Check) on Rabbit 2; the Gallery mode; the full list of `@Preview` parameters; what Layout Validation does in a Compose project; what `ui-tooling` is used for in detail; the "general knowledge" advice on building previews with plain data; and the exercises (not run by me). **A screenshot of a `@Preview` with the preview pane open, and of the preview toolbar, would turn this chapter into a verified one.**

Sources: Preview your UI with composable previews (developer.android.com), Debug your Compose UI (developer.android.com), View issues for your design tools in the Problems panel (developer.android.com).

COMING UP

Chapter 6 covers devices: creating a virtual device in the Device Manager, where that window appears, mirroring a phone or emulator in Running Devices, Device Explorer and wireless pairing.

Android Studio 2: The New UI

Course 2 · Chapter 6 · Device Manager and Running Devices

An app has to run somewhere: on an **emulator** (a virtual phone on your computer) or on a **physical device**. The Classic UI had an "AVD Manager" for the first and a plain device drop-down for the second. Rabbit 2 has two tool windows for the job: **Device Manager** (which devices exist, and what you can do with them) and **Running Devices** (the device's screen, shown inside the IDE). This chapter explains both, plus **Device Explorer** and pairing a phone over Wi-Fi.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (your screenshots and notes, Rabbit 2):** the names and locations of the two tool windows, and that the system images on offer include API 37.0 ("CinnamonBun"). **Checked in the docs (one web search):** mirroring and Wi-Fi pairing, marked *(docs)*. **General knowledge, not checked:** the steps for creating a virtual device and the contents of each window, marked *(general knowledge)*. I have **no screenshot** of a Running Devices window with a device in it, so its controls are not described.

Where Things Are

TOOL	HOW TO OPEN IT	WHAT IT IS FOR
Device Manager	Icon on the right-hand tool-window bar; **Tools > Device Manager**; **View > Tool Windows > Device Manager**. On the Welcome screen: **More Actions > Virtual Device Manager**.	List, create, start and stop virtual devices; see connected physical devices.
Running Devices	Icon on the right-hand bar; **View > Tool Windows > Running Devices** only (it is not under Tools). It opens by itself when an emulator or a mirrored phone starts.	Show and control a running device's screen inside the IDE.
Device Explorer	**View > Tool Windows > Device Explorer** (still in the list on Rabbit 2).	Browse a device's file system *(general knowledge)*.

Creating a Virtual Device

(general knowledge: I did not re-check these steps on Rabbit 2) In Device Manager, use the **+** (create device) button, choose a hardware profile (a phone size such as the Medium Phone the

toolbar shows), choose a **system image** (an Android version, which is downloaded the first time you pick it), then name it and finish. Your installation offers API 37.0 images. Two habits help:

- Match the image to what you are testing. A recent API level shows current behaviour; an older one near your project's `minSdk` (24 in a generated project, Chapter 3) shows the oldest phones you support.
- Prefer an image that matches your computer's processor type, since a mismatched one is far slower *(general knowledge)*.

The toolbar's device selector (it showed "Medium Phone API 37.0" in the project checked) chooses which device **Run** installs the app on.

Running Devices: the Screen in the IDE

Start an emulator from Device Manager and the Running Devices window opens with its screen, so the phone does not float in a separate window. You can show a *physical* phone the same way: *(docs)* **device mirroring** is available whenever a device with USB or wireless debugging turned on is connected, and you can start and stop it from Running Devices or from Device Manager. The mirroring settings are under **Settings > Tools > Device Mirroring**. The docs also warn that Android Studio can start mirroring automatically for any connected, paired device, so turn that off there if you do not want your phone's screen appearing whenever you plug it in.

Pairing a Phone over Wi-Fi

(docs) Wireless debugging needs a phone on **Android 11 or higher**, on the same network as your computer. The outline:

1. On the phone, turn on **Developer options**, then **Wireless debugging**.
2. In Android Studio open the **Pair Devices Using Wi-Fi** dialog from the run-configuration menu in the toolbar, or pair from Device Manager.
3. Pair with the **QR code** or a **six-digit pairing code** shown on the phone.

The docs' blog describes a newer "ADB Wi-Fi 2.0" flow with a pair-over-Wi-Fi icon in Device Manager, but says it needs Android 17 and Android Studio Quail 3 or later, so I have not assumed it applies to your install. If the phone does not appear, the old fallback is USB: connect

with a cable, accept the debugging prompt on the phone, and it appears in the device selector *(general knowledge)*.

```
# In the Terminal tool window (Alt+F12): list the devices the IDE can see
adb devices
```

COMPARED WITH THE CLASSIC UI

Android Studio: The IDE Itself 6 calls the tool the "AVD Manager" and does not mention Running Devices or Wi-Fi pairing. On Rabbit 2 the AVD Manager is now Device Manager, the emulator and a mirrored phone share the Running Devices window, and the ADB and architecture material in that chapter is still correct.

Try It Yourself

EXERCISE 1

Open Device Manager three ways (right-hand bar, Tools menu, View > Tool Windows) and Running Devices once. Say which entries are missing from the Tools menu.

View solution

EXERCISE 2

Create a virtual device with a system image of your choice, start it, and run the scratch project on it. Report which window opened when the emulator started and what the toolbar's device selector shows.

View solution

EXERCISE 3

If you own an Android phone, enable debugging, connect it by USB or Wi-Fi, and mirror it in Running Devices. Then find the Device Mirroring settings and decide whether you want mirroring to start automatically. Without a phone, find that settings page and describe what it offers.

View solution

NOT YET VERIFIED

Checked on the install: Device Manager's three entry points and the Welcome-screen route; Running Devices as a View-menu-only tool window that opens when a device starts; Device Explorer in the View list; API 37.0 images; the "Medium Phone API 37.0" toolbar selector.
Checked in the docs (one web search, summaries only): mirroring availability and start/stop, **Settings > Tools > Device Mirroring**, automatic mirroring, Android 11 for wireless debugging, the QR and six-digit pairing options, the Pair Devices Using Wi-Fi dialog, and the ADB Wi-Fi 2.0 caveat.
Not verified: the create-device steps; what Device Manager and Running Devices look like with a device; what Device Explorer shows on Rabbit 2; the exercises (not run by me). **A screenshot of Running Devices with a mirrored phone or emulator would verify this chapter.**
Sources: Run apps on a hardware device (developer.android.com), ADB Wi-Fi 2.0 (Android Developers' Blog).

COMING UP

Chapter 7 covers Logcat and the debugger: the redesigned Logcat window, its filter box, breakpoints and the Debug window.

Android Studio 2: The New UI

Course 2 · Chapter 7 · Logcat and the Debugger

When an app misbehaves you have two tools: read what it says (**Logcat**) and stop it mid-run to look inside (the **debugger**). Both work on a running device from Chapter 6. This chapter covers the Logcat window and its query-style filter box, then breakpoints, stepping and the Debug window.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (a screenshot, Rabbit 2):** the Logcat window's layout and its default filter. **Checked in the docs (two web searches, summaries only):** the query syntax and the debugger features, marked *(docs)*. **General knowledge, not checked:** marked *(general knowledge)*. I have no screenshot of the Debug window, so its layout is not described beyond what the docs say.

The Logcat Window

Logcat shows the messages the device's system and your app write. On the install it is a tool window (**View > Tool Windows > Logcat**; it has no shortcut) with these parts:

PART	WHAT IT DOES
Tab strip, with a + button	Several Logcat tabs, each with its own device and filter.
Device drop-down	Which device's log to show; it names the emulator and its Android version.
Filter box	Defaults to `package:mine`: only your project's app.
Cc toggle	Match case in the filter.
Star, help icon	Save the current filter as a favourite; open help.
Vertical button strip	Clear, pause, restart, wrap lines, scroll to end and others *(I could not read every icon's name)*.

The Query Filter

(docs) The filter takes **key:value** terms for `tag`, `package`, `process`, `message` and `level`. `package:mine` is special: it matches any package in the open project. `level:INFO` matches INFO and everything more serious (WARN, ERROR, ASSERT). A leading minus negates a term, and terms combine with `&`, `|` and parentheses. A `~` after the key (`tag~:`) switches that term to a regular expression.

```
# Only my app (the default)
package:mine

# Warnings and errors from my app
package:mine & level:WARN

# Hide a noisy tag
package:mine -tag:Choreographer

# Either my tag or any error
(tag:Catalogue | level:ERROR) & package:mine
```

The docs' own example is `(tag:foo | level:ERROR) & package:mine`; the other lines apply the same rules (`Choreographer` and `Catalogue` are example tags). To write a message yourself, use `Log.d("Catalogue", "saved item")` in Kotlin *(general knowledge)* and filter on that tag.

The Debugger

(docs) Run the app with **Debug** instead of Run. When execution reaches a **breakpoint** (click in the editor's left gutter next to a line) the app pauses, and you can examine variables, evaluate expressions and continue line by line. The docs also describe **conditional breakpoints** (stop only when a condition is true) and **logging breakpoints** (write a message to Logcat without stopping).

ACTION	WHAT IT DOES
Step over	Run the current line, stop at the next one *(general knowledge)*.
Step into	Go inside the function the line calls *(general knowledge)*.
Step out	Finish the current function and stop in its caller *(general knowledge)*.
Resume	Run until the next breakpoint *(general knowledge)*.

ACTION	WHAT IT DOES
Evaluate expression	Run a Kotlin expression in the paused frame; `Alt+F8` on Windows and Linux *(docs)*. The Variables pane also lets you evaluate ad hoc expressions.

The Debug tool window is `Alt+5` and Run is `Alt+4` (Chapter 1's tool-window list, checked on the install).

COMPARED WITH THE CLASSIC UI

Android Studio: The IDE Itself 4 describes the older Logcat, where you chose a level from a drop-down and built filters through the interface. On Rabbit 2 the filter is a typed query, and `package:mine` replaces the old "Show only selected application" idea *(my reading of the two, general knowledge)*. The debugger's core ideas are unchanged.

Try It Yourself

EXERCISE 1

Run the scratch project on a device, open Logcat and describe each control you can see. Clear the log, then switch the filter from `package:mine` to `level:ERROR` and say what changes.

View solution

EXERCISE 2

Add `Log.d("Demo", "hello")` to `onCreate` in `MainActivity.kt` and run. Write filters that show only that message by tag, and by message text. Save one as a favourite with the star.

View solution

EXERCISE 3

Set a breakpoint on a line that runs at start-up, start the app with Debug, and when it pauses step over twice, then evaluate an expression with `Alt+F8`. Finally change the breakpoint into a logging breakpoint and check the message reaches Logcat.

View solution

NOT YET VERIFIED

Checked on the install: the Logcat window parts in the table above, the `package:mine` default, Logcat having no shortcut, and the Run and Debug shortcuts (Alt+4, Alt+5).
Checked in the docs (summaries only): the key names, `package:mine`, `level:` semantics, negation, combining, the `~` regex form, conditional and logging breakpoints, and Alt+F8.
Not verified: the exact names of the Logcat button-strip icons; the Debug window's layout and button names; the step actions' shortcuts; how Logcat's old and new filters map to each other; my example queries (not run); the exercises (not run by me). **A screenshot of the Debug window while paused would verify the debugger half.**
Sources: View logs with Logcat (developer.android.com), Debug your app (developer.android.com).

COMING UP

Chapter 8 covers the task-based Profiler, App Inspection with the Network Inspector and Layout Inspector, and the Build Analyzer.

Android Studio 2: The New UI

Course 2 · Chapter 8 · The Profiler and App Inspection

Logcat and the debugger (Chapter 7) answer "what did the code do?". This chapter is about "how well did it do it?" and "what is it showing and sending?". Three tool windows cover it: the **Profiler** (CPU and memory), **App Inspection** (network traffic and the live layout) and the **Build Analyzer** (why the build is slow).

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (screenshots, Rabbit 2):** the Profiler's task-based start screen and that App Inspection and Profiler are in the View > Tool Windows list. **Checked in the docs (one web search, summaries only):** the Network Inspector and the Build Analyzer, marked *(docs)*. **General knowledge, not checked:** marked *(general knowledge)*. The search returned **nothing on the Layout Inspector or the Profiler tasks in detail**, so those parts are thin and labelled.

The Profiler: Start from a Task

The Classic UI Profiler opened on a timeline with CPU, memory, network and energy rows. On Rabbit 2 the start screen asks *what you want to find out* and offers **tasks** (checked on the install):

TASK ON THE START SCREEN	BEHIND IT
Capture System Activities	System Trace.
View Live Telemetry	A live view of the app's resource use.
Find CPU Hotspots	Java/Kotlin Method Recording, or Callstack Sample.
Analyze Memory Usage	Heap Dump.
Find Memory Leaks with LeakCanary	Uses the LeakCanary library's leak detection *(general knowledge)*.
Track Memory Consumption	Java/Kotlin Allocations, or Native Allocations.

There is **no Network task** in that screenshot, and no Power or Energy task was visible (I was told one might exist; that is unconfirmed). Network moved to App Inspection, below. A heap dump is a snapshot of the objects in memory, used to find what is holding on to it; sampling and method

recording show which functions take the time *(general knowledge)*. You profile a **running** app, so start a device first (Chapter 6). A blog item I found says Android Performance Analyzer is the updated System Trace viewer in recent builds *(docs, unconfirmed for Rabbit 2)*.

App Inspection and the Network Inspector

(docs) The **Network Inspector** lives in the **App Inspection** tool window: **View > Tool Windows > App Inspection**, then its Network Inspector tab. It shows real-time network activity on a timeline (data sent and received), with two views of a selected stretch of time:

- **Connection View**: the files sent and received.
- **Thread View**: which of your app's threads made each request.

(docs) It supports `HttpsURLConnection` and **OkHttp**, which means libraries built on OkHttp such as Retrofit can be inspected too. A **Rules** tab can change responses (status code, headers, body) so you can see how the app copes with, say, an error. I could not check the Rules tab on the install because the app was not running when I looked.

The Layout Inspector

(general knowledge, not found by my search) The Layout Inspector is also an App Inspection tool. It shows the *running* app's UI as a tree and a picture, so you can click an element and read its size and properties. That is the right tool for "why does this look wrong on the device?", and it works for Compose as well as XML. Treat the details as unchecked.

The Build Analyzer

(docs) After each build the **Build Analyzer** produces a report. Open **View > Tool Windows > Build** and click the **Build Analyzer** tab. It can point out configuration problems, for example dynamic dependency versions that cause unexpected downloads. Chapter 4's version catalog (fixed version numbers) is the usual cure for that one.

> **COMPARED WITH THE CLASSIC UI**
>
> Android Studio: The IDE Itself 5 teaches the Profiler's timeline and its Network Profiler. On Rabbit 2 the Profiler starts from tasks, and network traffic is inspected in the **Network Inspector** inside App

Inspection (confirmed on the install: no Network task exists).

Try It Yourself

EXERCISE 1

Open the Profiler with the app running and list every task it offers. Compare your list with the table above and note differences (especially anything about Power or Energy).

View solution

EXERCISE 2

Open App Inspection and find the Network Inspector and the Layout Inspector. In the Layout Inspector click an element of your running app and report which properties you can read. If your app makes no network calls, say what the Network Inspector shows instead.

View solution

EXERCISE 3

Run **Build > Rebuild Project**, then open the Build Analyzer tab in the Build window. Write down what its report says about your build, and whether it lists any warnings.

View solution

NOT YET VERIFIED

Checked on the install: the Profiler's six task entries and the absence of a Network task; App Inspection and Profiler in the View > Tool Windows list.
Checked in the docs (one web search, summaries only): the Network Inspector's location, timeline, Connection View, Thread View, OkHttp support and Rules tab; the Build Analyzer's location and purpose; the Android Performance Analyzer note.
Not verified: the Layout Inspector (nothing found); any Power or Energy task; what each Profiler task records and shows; what the Rules tab looks like; the Build menu path used in Exercise 3 (general knowledge); the exercises (not run by me). **A screenshot of the Network Inspector with a running app that makes requests, and one of the Layout Inspector, would verify this chapter.**

Sources: Inspect network traffic with the Network Inspector (developer.android.com), Troubleshoot build performance with Build Analyzer (developer.android.com), Introducing Android Performance Analyzer (Android Developers' Blog).

COMING UP

Chapter 9 covers the Code menu: Inspect Code, Code Cleanup, Analyze Code, Analyze Stack Trace and the Refactor menu, and why there is no Analyze menu any more.

Android Studio 2: The New UI

Course 2 · Chapter 9 · Inspect, Analyze and Refactor

The IDE can check your code without running it (**inspections** and **lint**), tidy it in bulk (**Code Cleanup**) and restructure it safely (**refactoring**). Older guides send you to an **Analyze** menu for the first group. On Rabbit 2 that menu is gone, and the commands live in the **Code** menu. This chapter shows where everything went.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (a screenshot of the Code menu, and your confirmation):** the entries of the Code menu and the absence of an Analyze menu. **Checked in the docs (one web search, summaries only):** Code > Inspect Code and Code Cleanup, marked *(docs)*. **General knowledge, not checked:** marked *(general knowledge)*. The search found **nothing** on Analyze Stack Trace, Play Policy Insights or the Refactor menu, so those are described only by name and by what I know of the IDE.

Where the Analyze Menu Went

The main menu (Chapter 1) has: File, Edit, View, Navigate, **Code**, **Refactor**, Build, Run, Tools, VCS (or Git), Window, Help. There is no Analyze. The Code menu now contains (checked on the install):

CODE MENU ENTRY	WHAT IT IS FOR
Inspect Code	*(docs)* Manually runs the configured lint and other IDE inspections; results appear in the Inspection Results window.
Inspect for Play Policy Insights	Checks the project against Google Play policy *(general knowledge from the name; not checked)*.
Code Cleanup	*(docs, from JetBrains' IntelliJ documentation, which Android Studio is built on)* Applies quick-fixes in bulk to a chosen scope without asking about each problem.
Analyze Code >	A submenu of further analysis tools *(I did not open it)*.
Analyze Stack Trace or Thread Dump	Paste a crash's stack trace and get clickable links into your source *(general knowledge)*.

Android Studio's lint page names the path as **Code > Inspect Code**; an older Android page still says **Analyze > Inspect Code** *(docs)*. Same command, older location. The Code menu also holds the editing commands you use daily (reformat, optimise imports and so on, *general knowledge*).

Lint and the Problems Window

Lint is Android's static checker: it finds likely bugs, missing translations, accessibility and performance problems. Inspect Code runs it together with the IDE's own inspections. Issues also appear as you type (underlines in the editor) and in the **Problems** tool window, `Alt+6` (checked on the install). Use Inspect Code when you want the whole project checked at once rather than the open file.

```
# Same lint check from the command line, in the Terminal tool window (Alt+F12)
./gradlew lint
```

(general knowledge: the Gradle task name is standard, but I have not run it on this install; on Windows use `gradlew lint`*.)*

The Refactor Menu

Refactoring has its own top-level menu now. *(general knowledge, the menu's contents were not captured)* The ones to learn first: **Rename** (changes every use of a name, `Shift+F6`), **Extract** (turn selected code into a function, variable or constant) and **Safe Delete** (delete only if nothing uses it). They check the whole project, which is why they beat find-and-replace.

> **COMPARED WITH THE CLASSIC UI**
>
> Android Studio: The IDE Itself 7 teaches "Analyze > Inspect Code". On Rabbit 2 use **Code > Inspect Code**. The Rename, Extract and Safe Delete material in that chapter is still correct.

Try It Yourself

EXERCISE 1

Open the Code menu and write down every entry. Check there is no Analyze menu. Open the **Analyze Code** submenu and list what it contains.

View solution

EXERCISE 2

Add an unused variable and a hard-coded string to a Kotlin file, then run **Code > Inspect Code** on the whole project. Find your two problems in the results and say how the results are grouped.

View solution

EXERCISE 3

Rename a function with **Refactor > Rename** and check that every call changed. Then copy a stack trace (invent one if needed) and try **Analyze Stack Trace or Thread Dump**. Report what each did.

View solution

NOT YET VERIFIED

Checked on the install: the five Code menu entries above, the absence of an Analyze menu, the Refactor menu's existence and the Problems shortcut (Alt+6).
Checked in the docs (one web search, summaries only): Code > Inspect Code and the Inspection Results window; the older Analyze > Inspect Code path; Code Cleanup as described for IntelliJ and Rider (not Android Studio's own page).
Not verified: the Analyze Code submenu; what Play Policy Insights and Analyze Stack Trace actually do; the Refactor menu's contents and shortcuts; the `gradlew lint` command; the exercises (not run by me). **A screenshot of the Analyze Code submenu and the Refactor menu would verify this chapter.**
Sources: Improve your code with lint checks (developer.android.com), Run inspections (jetbrains.com).

COMING UP

Chapter 10 covers Git in the New UI: the Version Control widget, the branch widget, the Commit window (Alt+0) and the Git window (Alt+9).

Android Studio 2: The New UI

Course 2 · Chapter 10 · Git in the New UI

Git saves the history of your project so you can undo, compare and share it. Android Studio has Git built in, and the New UI changes what you see depending on whether the project *has* a repository yet. This chapter shows both states, the two tool windows you will use most, and a few commit habits.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (your screenshots, Rabbit 2):** the toolbar widget and menu name before and after a repository exists, the Commit window and the Git window. **Checked in the docs (one web search, summaries only):** enabling version control from the VCS menu, marked *(docs)*. **From other JetBrains IDEs, not Android Studio:** committing and pushing steps, marked *(JetBrains docs, not checked here)*. **Share Project on GitHub is not verified.**

Before a Repository: VCS

In a project with no repository the menu bar says **VCS**, and the top toolbar shows a **Version Control** widget beside the project name (checked on the install). *(docs)* From the VCS menu choose **Enable Version Control Integration**, pick Git as the system for the project root and click OK. Version-control settings are also under **File > Settings > Version Control**.

After a Repository: Git

ITEM	BEFORE A REPOSITORY	AFTER A REPOSITORY
Menu bar	VCS	Git (replaces VCS)
Toolbar widget	Version Control	The current branch name (`master` in the project checked)

The Commit Window (Alt+0)

On the left bar. It lists the changed files with checkboxes, has an **Amend last commit** option, a message box, **Commit** and **Commit and Push…** buttons, and a gear for options (all checked on the install). Tick the files that belong together, write the message and commit. *(JetBrains docs, not checked here)* New files start under Unversioned Files and are added by ticking them.

Habit: in the screenshot the commit message was typed as `"initial commit"` *with the quote marks*. The IDE does not need them; they become part of the message. Write a short plain sentence, say what changed and why, and keep one idea per commit.

The Git Window (Alt+9)

At the bottom. It has **Log** and **Console** tabs, a branch tree (HEAD, Local, Remote), a search for text or a hash, and filters for **Branch, User, Date and Paths** (checked on the install). In that project the log showed `origin & master`, meaning the project already pointed at a remote. The Log tab is where you read history; the Console shows the Git commands the IDE ran.

Remotes and Pushing

A **remote** is a copy on a server such as GitHub, usually named `origin`. Two routes are possible: make the repository in the IDE first and add a remote later, or create the repository on GitHub first and clone or connect it. *(JetBrains docs, not checked here)* Other JetBrains IDEs push with **Git > Push** or `Ctrl+Shift+K`, which lists the commits before sending them; **Commit and Push…** does both in one step. Some JetBrains IDEs can also create a GitHub repository from the VCS or Git menu; I could not confirm that for Android Studio, so check the menu yourself. If you prefer the command line, the Terminal window (`Alt+F12`) runs the same commands:

```
# Same actions in the Terminal tool window
git status
git add .
git commit -m "Add catalogue list screen"
git push
```

KEEP SECRETS OUT

A generated project's `.gitignore` normally excludes `local.properties` and build folders *(general knowledge)*. Check it before the first push, and never commit passwords or API keys.

COMPARED WITH THE CLASSIC UI

Android Studio: The IDE Itself 8 describes a Git widget in the status bar and the VCS menu throughout. On Rabbit 2 the branch widget is in the **top toolbar** and the menu is **Git** once a repository exists. The Git concepts in that chapter still apply.

Try It Yourself

EXERCISE 1

In a scratch project with no repository, note the menu name and toolbar widget. Use **VCS > Enable Version Control Integration** with Git, then note what changed in the menu bar and the widget.

View solution

EXERCISE 2

Open the Commit window (Alt+0), tick your files, commit with a message that has no quote marks, then open the Git window (Alt+9) and find your commit in the Log tab.

View solution

EXERCISE 3

Search the Git menu for any command that creates or shares a repository on GitHub and report its exact name, or say there is none. Then create a branch and switch back to `master` using the toolbar widget.

View solution

NOT YET VERIFIED

Checked on the install: VCS menu and Version Control widget before a repository; Git menu and branch widget after; the Commit window's contents and Alt+0; the Git window's tabs, tree, search, filters and Alt+9; the quoted commit message.
Checked in the docs (one web search): Android Studio's *Enable Version Control Integration* and the Settings path.

From other JetBrains IDEs only: unversioned-file handling, Git > Push and Ctrl+Shift+K.
Not verified: Share Project on GitHub or any GitHub menu item; the toolbar widget's branch menu (Exercise 3); the `.gitignore` contents; the exercises (not run by me). **A screenshot of the Git menu open (after a repository exists) and the branch widget's drop-down would verify this chapter.**
Sources: Version control basics (developer.android.com), Tutorial: Getting started with Git in GoLand (jetbrains.com).

COMING UP

Chapter 11 covers the Agent window and Studio Labs. Chapter 12 is the capstone.

Android Studio 2: The New UI

Course 2 · Chapter 11 · The Agent Window and Studio Labs

Two things on your install are new since the Classic UI: an **Agent** tool window on the right-hand bar, and a **Studio Labs** area in Settings. This is a short chapter, because I have not used either on your machine, so everything below comes from Google's documentation and blog, not from what I saw.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install:** only that **Agent** appears on the right-hand tool-window bar and in **View > Tool Windows** (Chapter 1). **Checked in the docs (one extended web search, summaries only):** everything else, marked *(docs)*. **Not verified:** what the window looks like, what it can do on your account, and where Studio Labs sits in your Settings tree. The AI features change quickly, so expect some of this to be out of date soon.

The Agent Window

(docs) Current Android documentation says you open it by clicking **Agent** in the sidebar or choosing **View > Tool Windows > Agent**. Older 2025 posts call the same place "Gemini" and its Agent tab, so older guides name it differently. Google's blog describes **Agent Mode** as an AI assistant that can take a task you describe in plain language and work across your project, rather than only answer one question. The 2026 Otter 3 post adds that you can choose between language models.

What it can do depends on your version, your sign-in and (for businesses) an administrator's settings *(docs: there is a separate "Gemini in Android Studio for businesses" page; I did not read it)*.

Studio Labs

(docs) Studio Labs is a part of **Settings** where experimental features are switched on. A 2025 Google I/O post says to open the **Studio Labs** tab in Settings and enable the features you want; in *canary* releases those AI features are on automatically. **Journeys** (AI-run tests described in plain language) is one example, and the docs say running journeys with the agent needs Android Studio Rabbit 3 or later, so it may not be on your Rabbit 2. I could not find a page that

says exactly where Studio Labs sits in the current Settings tree, so use the Settings search box (Chapter 2) and type "Labs".

What to Trust and What to Check

This part is advice, not documentation *(general knowledge)*:

- **Read every change before you keep it.** The agent's output is a suggestion. You are still the one whose name is on the commit.
- **Commit first.** With a clean Git state (Chapter 10) you can see exactly what the agent changed in the Commit window, and throw it away if it is wrong.
- **Build, run and test.** A change that looks right can still fail to compile or break behaviour. Chapters 6 and 7 are how you find out.
- **Be careful what you share.** Anything in your project may be sent to a remote service; do not let secrets (API keys, passwords) sit in files the agent reads.
- **Check version-specific advice.** AI tools sometimes describe old menus (a stale "Analyze" menu, say). Compare with what Chapters 1 and 9 show on your screen.

> **COMPARED WITH THE CLASSIC UI**
>
> Nothing in Android Studio: The IDE Itself covers an agent or Studio Labs; both are new to this edition.

Try It Yourself

EXERCISE 1

Open the Agent window from the right-hand bar, then close it and reopen it with **View > Tool Windows > Agent**. Describe what it shows before you sign in or type anything.

View solution

EXERCISE 2

Open Settings, search for "Labs" and report what you find and which features can be switched on or off. Do not enable anything you do not understand.

View solution

EXERCISE 3

In a scratch project with a clean Git state, ask the agent (if you have access) for one small change. Review it in the Commit window, build and run the app, then keep or discard it, and write down your reasons.

View solution

NOT YET VERIFIED

Checked on the install: the Agent entry on the tool-window bar and in View > Tool Windows.
Checked in the docs (one extended search, summaries only): the two ways to open the Agent window; its older name (Gemini); Agent Mode as a task-taking assistant; model choice (Otter 3 post); the Studio Labs tab in Settings for enabling experimental features; Journeys needing Rabbit 3 for agent runs.
Not verified: the window's appearance and sign-in steps; what Studio Labs offers on your install and where it sits; availability on your account; the exercises (not run by me). **Screenshots of the Agent window and the Studio Labs settings page would verify this chapter.**
Sources: AI in Android Studio features (developer.android.com), Google I/O 2025: What's new in Android development tools (Android Developers Blog), Otter 3 Feature Drop (Android Developers Blog), Journeys for Android Studio (developer.android.com).

COMING UP

Chapter 12 is the capstone: start a project, read its Gradle files, preview a screen, run it on a device, read Logcat, set a breakpoint, take a profiler task and commit.

Android Studio 2: The New UI

Course 2 · Chapter 12 · Capstone: Build, Run and Inspect a Compose App

Eleven chapters taught the tools one at a time. This one uses them together, in the order you would on a real day: make a project, read its build files, preview a screen, run it, read the log, stop it with a breakpoint, profile it and commit. Nothing here is new; each step names the chapter it comes from. Do it in a scratch project.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> This chapter adds no new facts. Every step repeats something from Chapters 1 to 11, and carries that chapter's evidence level: the **menu names, tool-window locations, Gradle files and Profiler tasks** were checked on the install; **Compose Preview, the debugger, Network and Layout Inspectors, Code-menu details and Git push** come from the docs or general knowledge. I have not run this walkthrough. Where a step fails, trust your screen over the book, and tell me what you saw.

The Walkthrough

#	STEP	CHAPTER
1	Open the Welcome screen or **File > New Project**. Choose **Empty Activity** (Compose), name the project, accept minimum SDK 24, Finish.	3
2	In the Android view open **Gradle Scripts**. Find `build.gradle.kts (Module :app)` and `libs.versions.toml`; note the AGP version and the Compose BOM line.	4
3	Open `MainActivity.kt`. Show the preview pane next to the `@Preview`; change the greeting text.	5
4	In Device Manager start (or create) an emulator. Check Running Devices opens and the toolbar selector shows it. Press Run.	6
5	Add `Log.d("Capstone", "started")` in `onCreate`, run again, and filter Logcat with `package:mine & tag:Capstone`.	7
6	Set a breakpoint on that line, start with **Debug**, step over once and evaluate an expression with `Alt+F8`. Resume.	7
7	Open the Profiler with the app running and take one task (for example **View Live Telemetry**). Open App Inspection and look at the Layout Inspector.	8

#	STEP	CHAPTER
8	Run **Code > Inspect Code** on the whole project and read the results.	9
9	**VCS > Enable Version Control Integration** (Git), then Commit (`Alt+0`) with a plain message and find it in the Git window (`Alt+9`).	10

```
// MainActivity.kt: the one line the capstone adds
import android.util.Log

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("Capstone", "started")
    // ...the generated setContent { ... } stays as it is
}
```

If the Agent window (Chapter 11) is available to you, you may use it in step 8 to explain a finding, but review what it suggests before keeping it.

Checklist

- I can open Device Manager, Running Devices, Logcat and the Profiler without searching.
- I know which files in **Gradle Scripts** hold versions, and which hold settings.
- I can write a Logcat query by package, tag and level.
- I can set, step through and remove a breakpoint.
- I know the Code menu holds Inspect Code and that there is no Analyze menu.
- I commit with a plain message and can find the commit in the Git window.
- I know which parts of this course were checked on a real install and which were not.

Where Things Live: One-Page Map

YOU WANT TO...	GO TO
Reach any command	Hamburger menu, or Search Everywhere (Chapter 1)
Change a setting	Settings (Chapter 2)

YOU WANT TO...	GO TO
See project files	Project tool window, Android or Project view (Chapter 3)
Change a version	`libs.versions.toml`, then sync (Chapter 4)
See a screen without running	`@Preview` (Chapter 5)
Start a device	Device Manager, then Running Devices (Chapter 6)
Read the app's messages	Logcat, filter `package:mine` (Chapter 7)
Pause and inspect	Breakpoint and Debug, `Alt+5` (Chapter 7)
Find slowness, memory use, network traffic	Profiler tasks, App Inspection (Chapter 8)
Check code quality	Code > Inspect Code, Problems `Alt+6` (Chapter 9)
Save history	Commit `Alt+0`, Git `Alt+9` (Chapter 10)
Ask the AI agent	Agent tool window (Chapter 11)

COMPARED WITH THE CLASSIC UI

Android Studio: The IDE Itself 9 is the Classic UI capstone: it used the Layout Editor, the AVD Manager, the Network Profiler and the Analyze menu. Every one of those steps has changed here, which is why this course exists.

Try It Yourself

EXERCISE 1

Do steps 1 to 4 of the walkthrough and write down, for each, anything that differed from this course's description.

View solution

EXERCISE 2

Do steps 5 to 7: log a line, catch it with a breakpoint and take a Profiler task. Write the Logcat query you used and the task you chose.

View solution

EXERCISE 3

Do steps 8 and 9, then send me your list of differences from all three exercises. That list is what turns the unverified parts of this course into verified ones.

View solution

COURSE COMPLETE

You have covered the New UI (1), settings (2), projects and views (3), Gradle in Kotlin DSL (4), Compose Preview (5), devices (6), Logcat and the debugger (7), the Profiler and App Inspection (8), the Code menu (9), Git (10) and the Agent window and Studio Labs (11). Next in the Android track are the Compose-first app-building courses.

NOT YET VERIFIED

Checked on the install: the items listed as checked in Chapters 1 to 4 and the names and locations in 6 to 10.
Not verified: this walkthrough as a whole (not run by me), and all items each chapter lists as unverified. See each chapter's own Not Yet Verified box and the screenshots-wanted list.
Sources: those listed in Chapters 5 to 11.