OFFICE & PRODUCTIVITY SOFTWARE

Google Workspace

Cloud-Native, Not File-Based · Real-Time Collaboration

Version History · Sheets, Slides & Apps Script

File Format Interoperability

Capstone: Real-Time Team Collaboration on a Shared Report

Philip Osztromok

Generated with Claude

Table of Contents

Google Workspace — 9 Chapters

This course assumes you already know Word, Excel, PowerPoint, and their LibreOffice equivalents. Rather than touring Docs/Sheets/Slides feature by feature the way the LibreOffice course toured Writer/Calc/Impress, this course focuses on what's genuinely, structurally different about Google Workspace — starting with the single biggest one: there's no local file at all, by default.

Every Prior Course Assumed a File

Word saves a `.docx` file. Writer saves a `.odt` file. Both are genuine files sitting on a disk somewhere, which you explicitly save, and which you could lose entirely if you forget to save before a crash — AutoRecover and background autosave features soften that risk, but the fundamental unit both applications work with is still **a file you explicitly save**.

A Google Doc Is Not a File — It's a Cloud Object

Opening a new document at docs.google.com (or the Sheets/Slides equivalent) creates something that lives on Google's own servers from the moment it's created — not a file on your own computer at all. There's no "Save As" step needed at the start, and normally no manual save step at any point afterward. Every keystroke syncs to the server automatically and continuously, typically within a second or two of typing it.

No "Save" Button, By Design

Unlike Word's or Writer's own `Ctrl+S`, Google Workspace documents genuinely don't have a primary Save command in the traditional sense — there's simply nothing to manually trigger, because there's no separate unsaved state to commit. **File** → **Download** exists, but that's a fundamentally different action: exporting a standalone copy, not saving the live document.

What "Autosave" Actually Means Here

This is not the same mechanism as Word's own AutoRecover, which quietly writes to a separate temporary recovery file you'd restore from after a crash, alongside whatever you last manually saved. Google Workspace's autosave writes directly to the single, canonical, live version of the document on Google's own servers, continuously — there is no separate "recovery" step needed after a crash, because there was never a separate, only-locally-held working copy to lose in the first place.

The Practical Implications

- **Working offline** requires an explicit setting (offline mode via the Google Docs offline extension/app) rather than being the assumed default the way a desktop application always works without an internet connection.
- **Sharing a document** means sharing a link to the one live cloud object — not attaching a file, and not sending a copy at all.
- **Multiple browser tabs or devices** viewing the same document all see the exact same live version — there are no separate local copies that could quietly drift apart from each other.

	WORD / WRITER	GOOGLE WORKSPACE
Where the document lives	A file on your own disk	A cloud object on Google's own servers
Saving	Manual (Ctrl+S), plus background AutoRecover	Automatic, continuous, direct to the live version — no manual step
"Download"	Opening the actual working file	Exporting a separate, disconnected copy of the live document
Working offline	The default	An explicit opt-in mode

THIS ISN'T A SMALL UI DIFFERENCE — IT'S THIS COURSE'S FOUNDATION

Every other chapter in this course builds directly on this one architectural choice: Chapter 2's real-time collaboration, Chapter 4's Version History, and Chapter 8's import/export behaviour all exist precisely because a Google document is a live cloud object rather than a file. Getting this chapter genuinely solid is worth more time than any single feature-by-feature comparison could be.

"DOWNLOAD" IS NOT "SAVE" — A GENUINE TRAP COMING FROM A FILE-BASED BACKGROUND

Someone used to Word or Writer's own mental model can easily assume File → Download is the Google Workspace equivalent of saving — it isn't. Downloading creates a completely separate, disconnected local file that does not stay in sync with the live cloud document going forward. Editing the downloaded copy afterward does nothing to the original online document, and vice versa — the two immediately become two entirely independent things the moment the download happens.

Hands-On Exercises

EXERCISE 1

Create a new blank Google Doc and type a paragraph. Without pressing any save shortcut, close the browser tab entirely and reopen the document from Google Drive. Describe what you'd expect to find, and explain why no explicit save step was needed for this to work correctly.

View solution

EXERCISE 2

Use File > Download to save a Google Doc as a .docx file to your computer. Edit a sentence in the downloaded .docx file using a different application, then go back and check the original Google Doc online. Explain exactly what you find, and why.

View solution

EXERCISE 3

A colleague coming from years of Word experience asks, "so what's the Ctrl+S equivalent in Google Docs, since I don't want to lose my work?" Explain why this question is based on a mental model that doesn't quite apply here, and what actually protects their work instead.

View solution

CHAPTER 1 QUICK REFERENCE

- A Google Doc/Sheet/Slide is a **cloud object** on Google's own servers, not a file on your disk
- There is **no manual Save command** — every change syncs automatically and continuously to the live version
- **File → Download** exports a separate, disconnected copy — it is not the same action as saving
- Offline access is an explicit opt-in mode, not the assumed default
- Every later chapter in this course — real-time collaboration, Version History, import/export — builds directly on this one architectural fact

Chapter 1 established that a Google Doc is one live cloud object, not a file living on one person's disk. This chapter is the direct payoff of that fact: because there's only ever one live copy, multiple people can edit it at the exact same instant, watching each other's cursors move in real time — something a file-based model genuinely cannot do.

Recap: How Word Solves Collaboration

Word & PowerPoint Chapter 6 covered Track Changes — a fundamentally **asynchronous** model. One person edits, their changes are recorded as tracked insertions and deletions, and another person reviews and accepts or rejects them *later*, typically not at the same moment the original edits were made.

What "Real-Time" Actually Means

Because Chapter 1 established that a Google Doc has only one live copy, multiple people opening the same document are genuinely editing the exact same object at the exact same instant — not separate copies that get reconciled afterward. Each active editor's cursor appears as its own coloured marker, labelled with their name, visible to everyone else, moving and typing live as they work.

Why This Wasn't Possible in Word or Writer

A Word or Writer file lives on one disk (or a shared network location) at a time — two people typing directly into the literal same file simultaneously risks conflicts or outright corruption, which is exactly why Track Changes exists as an asynchronous workaround: record each person's edits separately, reconcile them afterward. Google Workspace's cloud-native architecture from Chapter 1 sidesteps this problem entirely, since every session simply connects to the one existing live copy rather than each holding its own separate copy that needs reconciling.

Seeing Who's Editing

Every active viewer or editor shows as a small avatar in the top-right corner of the interface, and as a coloured, name-labelled cursor wherever they're currently working in the document. Hovering over any cursor confirms exactly who it belongs to — a live, always-current picture of who's doing what, rather than something you'd only discover after the fact the way Track Changes reveals who made an edit.

Editing the Same Paragraph Simultaneously

If two people type in the same paragraph, or even the same word, at the same moment, Google's own conflict-resolution merges both people's keystrokes automatically and near-instantly in almost all cases — neither person needs to explicitly resolve a conflict the way you might in some other collaborative-editing contexts. The document simply reflects both sets of changes, interleaved correctly, continuing the same live-sync behaviour Chapter 1 already established for a single editor.

Sharing and Permission Levels

The **Share** button is the actual mechanism that lets more than one person access the same cloud object in the first place — Viewer, Commenter, or Editor roles, assigned per person or via a shareable link. This is precisely Chapter 1's own "sharing a Doc means sharing a link to the live cloud object, not a file" point, now made concrete: sharing IS how real-time collaboration actually begins.

	WORD & POWERPOINT (TRACK CHANGES)	GOOGLE WORKSPACE
Timing	Asynchronous — edit now, review later	Synchronous — everyone edits the same live document at once
Seeing others while they work	Not possible — only see edits after they're made	Live, named, coloured cursors visible in real time
Why this model exists	A file can't be safely edited by two people at once	There's only one copy in the first place — nothing to reconcile
Granting access	Sending or sharing a copy of the file	Share button — Viewer/Commenter/Editor roles on the one live object

REAL-TIME COLLABORATION ISN'T A BOLTED-ON FEATURE — IT'S A CONSEQUENCE OF CHAPTER 1

Google Workspace didn't add live collaboration as an extra capability on top of an otherwise-ordinary document format — it falls out naturally from the architectural choice Chapter 1 described. Once there's genuinely only one live copy of a document, and any number of sessions can connect to that same copy, simultaneous multi-person editing is simply what happens, not a separate feature someone had to design in on top of a file-based starting point.

THERE'S NO PRIVATE STAGING AREA — EVERY EDIT IS INSTANTLY VISIBLE TO EVERYONE CURRENTLY VIEWING

Unlike Track Changes, where you can freely experiment in your own copy before anyone else sees anything, an edit made directly in a shared Google Doc is visible to every other current viewer the instant it's typed, with no draft or private buffer by default. This is worth remembering before making a large structural change while colleagues are actively viewing the document — Chapter 3's own Suggesting mode is the tool built specifically to address this concern, letting you propose changes without directly altering what everyone else currently sees.

Hands-On Exercises

EXERCISE 1

Share a Google Doc with a second account (or a colleague), both open it simultaneously, and have both people type in different parts of the document at the same time. Describe exactly what each person sees happening on their own screen while the other person types.

View solution

EXERCISE 2

Explain, in your own words, why two people editing the same Word .docx file saved on a shared network drive at the exact same moment is a genuinely different (and riskier) situation than two people editing the same Google Doc at the exact same moment.

View solution

EXERCISE 3

A colleague wants to try reorganizing an entire section of a shared report while two other people are actively viewing it live, and is worried about disrupting their work-in-progress reading. Explain why this concern is genuinely valid given this chapter's own warning box, and name the tool (covered in the next chapter) built specifically to address it.

View solution

CHAPTER 2 QUICK REFERENCE

- Google Workspace collaboration is **synchronous** — everyone edits the same live object at once; Word's Track Changes is **asynchronous**
- Each active editor shows as a **named, coloured cursor**, live, plus an avatar in the interface's top-right corner
- Simultaneous edits to the same text merge automatically — no manual conflict resolution needed in ordinary use
- The **Share** button (Viewer/Commenter/Editor) is what grants access to the one live cloud object in the first place
- This entire chapter is a direct consequence of Chapter 1's cloud-native architecture, not a separate feature
- There is no private staging area by default — every edit is instantly visible to every current viewer; Chapter 3's Suggesting mode addresses this

Chapter 2 closed with a genuine concern: editing directly changes what every current viewer sees, instantly, with no private staging area. This chapter is the tool that solves it — Suggesting mode gives you Track-Changes-style safety without giving up Chapter 2's own real-time visibility.

The Three Editing Modes

A mode switcher in the top-right area of the interface offers three options: **Editing** (direct changes, visible instantly to everyone, exactly as Chapter 2 described), **Suggesting** (proposed changes shown distinctly, never applied to the actual text until explicitly accepted), and **Viewing** (read-only).

Suggesting Mode: Google's Own Live Equivalent of Track Changes

Switch to Suggesting mode, then type or delete text as normal. Insertions appear as coloured, underlined text; deletions appear as coloured strikethrough — the exact same visual convention Word & PowerPoint Chapter 6 used for tracked changes. The genuine difference: this is happening inside a live, shared document. Anyone else currently viewing sees your suggestion appear in real time as you type it, clearly marked as a proposal rather than an actual change to the underlying text.

Accepting and Rejecting Suggestions

Small checkmark and X icons appear directly beside each suggestion — the document's owner, or any Editor, can accept or reject each one individually. The underlying concept is identical to Word's own Accept/Reject from Chapter 6; only the mechanism (inline icons rather than a separate Review tab) differs.

Comments: Discussion Without Changing the Text

Selecting text and clicking the comment icon (or Insert → Comment) attaches a note to that specific selection, without altering the document's actual content at all — the identical concept to Word & PowerPoint Chapter 6's own Comments. The genuine difference here: comments can be replied to in real time by anyone currently viewing, and typing `@` followed by someone's email address notifies them directly, tying back to Chapter 2's own live, multi-person nature.

Comments vs. Suggestions: The Same Distinction as Word's Own

A **Comment** is a question or note *about* the content, never touching the text itself. A **Suggestion** is an actual proposed edit *to* the text, that can subsequently be accepted or rejected. This maps directly onto Word & PowerPoint Chapter 6's own Comment-vs-Tracked-Change distinction — nothing new to learn here conceptually, only new names for the same two roles.

Resolving a Comment

Marking a comment thread **Resolved** greys it out and collapses it without deleting the underlying discussion — the same idea as Word's own Resolve feature, preserving the conversation for later reference rather than erasing it outright.

	WORD & POWERPOINT (CH.6)	GOOGLE WORKSPACE
Timing of proposed edits	Recorded, reviewed later, asynchronously	Visible live to every current viewer as they're proposed
Visual convention	Underlined insertions, struck-through deletions	Identical — underlined insertions, struck-through deletions
Accepting/rejecting	A dedicated Review tab	Small inline checkmark/X icons per suggestion
Notifying a specific person	No built-in equivalent	@mention by email inside a Comment

THIS IS EXACTLY THE TOOL CHAPTER 2 PROMISED

Suggesting mode doesn't sacrifice Chapter 2's live, multi-person visibility to get Track-Changes-style safety — it keeps both. Several people can propose suggestions simultaneously, all visible live to everyone currently viewing, each one clearly marked as tentative rather than final, resolving the exact tension Chapter 2's own closing warning box raised.

FORGETTING WHICH MODE IS ACTIVE RECREATES CHAPTER 2'S OWN RISK

Intending to propose a tentative change but forgetting to switch out of Editing mode first directly edits the live text every current viewer sees, instantly — exactly the risk Chapter 2 warned about, simply reintroduced by not checking the mode switcher first. Separately, suggestions left unresolved for a long time can accumulate and visually clutter a document for everyone still viewing it, unlike Word's own tracked changes, which are typically all reviewed together in one batch before a document moves forward — it's worth resolving or addressing suggestions promptly rather than letting them pile up.

Hands-On Exercises

EXERCISE 1

Switch a shared Google Doc to Suggesting mode and rewrite one sentence. Have a second viewer confirm what they see appear in the document while you type, and describe how it visually differs from what they'd see if you had done the same edit in Editing mode instead.

View solution

EXERCISE 2

Add a Comment (not a Suggestion) to a paragraph asking a genuine open question about its content, and @mention a specific colleague's email address inside it. Explain what happens as a result of the @mention, and why a Comment (not a Suggestion) was the correct tool for this specific situation.

View solution

EXERCISE 3

A colleague meant to propose a tentative rewrite of an entire paragraph but forgot to check which mode was active first, and the paragraph changed instantly for every current viewer with no suggestion marker at all. Diagnose exactly what happened and what single check would have prevented it.

View solution

CHAPTER 3 QUICK REFERENCE

- **Editing / Suggesting / Viewing** — the three modes, switched from the top-right of the interface
- **Suggesting mode** — proposed changes shown live to current viewers, never applied until accepted; same visual convention as Word's Track Changes
- **Comments** — discussion attached to a selection, never alters the text; can @mention someone by email for a direct notification
- Comment = a question about content; Suggestion = an actual proposed edit — the same distinction as Word & PowerPoint Chapter 6
- **Resolve** collapses a comment thread without deleting the discussion
- Always check which mode is active before a large edit — forgetting recreates Chapter 2's own "no private staging area" risk

Chapter 1 established there's no manual Save step at all. This chapter covers what actually protects your work instead — **Version History** — and why it's a genuinely more comprehensive safety net than the manual-save-plus-AutoRecover model every other course in this subject assumed by default.

Recap: What Manual Saving Protects Against

Word and Writer's own safety net is built around explicit save points — each save creates a new "point in time" you could theoretically return to, backed up by AutoRecover's own background temp-file copy in case of a crash between saves. The protection is fundamentally tied to remembering to save, and to whatever the AutoRecover interval happens to be.

Version History: Continuous, Automatic Snapshots

File → Version History → See Version History shows a list of automatically-created checkpoints, generated continuously as you work — based on time intervals and significant edit activity, not on anyone remembering to trigger anything. Each checkpoint shows exactly who made which changes, highlighted in that editor's own colour — the same colour-per-person convention Chapter 2 introduced for live cursors, now applied to historical attribution.

Naming a Version

A specific version can be given a name — "Draft sent to client," for instance — making it easy to find and return to that exact point later, rather than scrolling through a long list of auto-generated timestamps looking for the right one.

Restoring an Earlier Version

Selecting an earlier version and choosing **Restore this version** reverts the entire live document to that earlier state. Crucially, this doesn't delete anything that happened in between — the restore itself simply becomes a new version in the history, meaning a restore can always be undone by restoring the version from immediately before it. Nothing is ever permanently lost.

Why This Is a Genuinely Different Safety Net Than Manual Saving

Manual saving protects against forgetting to save — a risk that doesn't apply here per Chapter 1 — and gives you specific points you deliberately chose to save at. Version History instead automatically creates far more checkpoints than any person would ever manually save, continuously, with no action required from anyone. The protection is comprehensive and automatic rather than dependent on remembering to trigger it.

Recovering from an Accidental Deletion

A common real scenario: someone accidentally deletes an entire section or table, and — per Chapter 2's own real-time model — that deletion has already propagated live to every current viewer. A local **Ctrl+Z** undo only helps within the same browser session that made the change. Version History, being a server-side record rather than a local undo buffer, lets you recover the deleted content days later, from any session or device, since it isn't tied to any one browser tab's own memory.

	MANUAL SAVE + AUTORECOVER	VERSION HISTORY
What creates a checkpoint	An explicit user action (Ctrl+S)	Automatic, continuous, based on time and edit activity
If you do nothing	Risk of losing unsaved work since the last save	Nothing to lose — every change is already captured
Recovering from a mistake	Reopen from a local AutoRecover file, if one exists	Browse and restore any version, from any session
Scope of protection	Tied to one machine's own last save/recovery file	A continuous, server-side record spanning every session and device

THE MISSING SAVE BUTTON ISN'T A GAP — IT'S WHAT MAKES THIS POSSIBLE

Version History exists specifically because there's no manual save step to interrupt it. If Google Workspace had a traditional Save button the way Word does, the fine-grained, comprehensive protection Version History actually provides would be undermined — checkpoints would only exist at whatever moments someone happened to save, exactly the more limited model Chapter 1 already contrasted this against.

RESTORING AN OLD VERSION REVERTS THE ENTIRE DOCUMENT, NOT JUST THE PART YOU WANT BACK

If you only need to recover one deleted paragraph or table, restoring an old version wholesale also discards any other good work done since that earlier point — not just the mistake you wanted undone. The better approach for a partial recovery: open the old version, copy just the specific content you actually need, then paste it back into the current, up-to-date version manually — rather than doing a full restore that throws away everything else that's happened since.

Hands-On Exercises

EXERCISE 1

Open Version History on a document you've been editing for a while. Name one specific version something meaningful. Explain why naming a version is useful given how many auto-generated checkpoints likely exist in the same history.

View solution

EXERCISE 2

A colleague accidentally deletes an entire table from a shared document, and it's already gone from everyone's live view. Describe exactly how Version History would let you recover it, and explain why a simple Ctrl+Z from a different browser session wouldn't necessarily work.

View solution

EXERCISE 3

A document has had 3 hours of good new work added since an accidentally-deleted paragraph. A colleague suggests just restoring the version from before the paragraph was deleted. Explain what would go wrong with that approach, and describe the correct alternative.

View solution

CHAPTER 4 QUICK REFERENCE

- **File → Version History → See Version History** — automatic, continuous checkpoints, no manual action needed
- Each version shows **who changed what**, colour-coded per editor, the same convention as Chapter 2's live cursors
- **Naming a version** makes it easy to find later among many auto-generated checkpoints
- **Restore this version** reverts the ENTIRE document — the restore itself becomes a new version, so nothing is ever permanently lost
- Version History is a **server-side record**, unlike a local Ctrl+Z undo buffer tied to one browser session
- For a partial recovery, copy just the needed content from an old version rather than restoring wholesale

The LibreOffice course's own Chapter 5 already covered formula syntax gotchas between Excel and Calc in real depth. This chapter won't repeat that ground — Sheets' formula syntax needs only a brief mention before pivoting to what's actually distinctive here: Chapters 2 through 4's collaboration concepts, now applied directly to spreadsheet data.

Formula Syntax: Comma-Separated, Just Like Excel

Sheets uses a comma as its function argument separator by default, matching Excel directly — unlike Calc, where the separator depends on locale settings (LibreOffice Chapter 5's own point). `SUM`, `AVERAGE`, `IF`, `VLOOKUP`, and `XLOOKUP` all exist under the same names, transferring with minimal friction. That's genuinely the extent of what's interesting on the syntax side — the real content of this chapter lies elsewhere.

The Real Point: Real-Time Collaboration on a Spreadsheet

Everything Chapter 2 described for Docs applies directly to Sheets: multiple people can edit different cells — or even the same cell — of the same spreadsheet simultaneously, each shown as a distinct, named, coloured cursor, live, exactly the same underlying mechanism as a shared Doc.

Cell-Level Attribution

A genuinely spreadsheet-specific detail: right-clicking or checking a specific cell's own edit history can show exactly who last changed that individual cell and when — a finer-grained version of Chapter 4's own whole-document version snapshots, now available at individual-cell resolution.

Simultaneous Formula Editing: What Happens to Dependent Cells

If one person changes a raw input value while another is actively viewing a formula elsewhere in the sheet that depends on it, that dependent formula recalculates live for both people at once — the same real-time merge behaviour Chapter 2 described for shared text, just applied here to a calculation dependency chain instead of a paragraph.

Comments and Suggesting Mode Apply Here Too

Chapter 3's Comments work identically in Sheets — attached to a specific cell rather than a text selection, but otherwise the same discussion-without-editing concept. Cell values themselves don't get quite the same underline/strikethrough "Suggesting" treatment Docs applies to flowing text, for a sensible structural reason: a cell holds one discrete value, not a sequence of characters that can be partially struck through and partially underlined the way a sentence can.

	EXCEL	CALC	SHEETS
Argument separator	Comma	Comma or semicolon, by locale	Always comma
Live, simultaneous multi-person editing	Co-authoring exists, added later	Not a core design assumption	Built in from the start (Chapter 1's own architecture)
Cell-level "who edited this" attribution	Not a standard feature	Not a standard feature	Available per cell
Comments attached to a cell	Yes	Yes	Yes — same underlying concept

THIS CHAPTER DELIBERATELY ISN'T A SYNTAX TOUR

LibreOffice Chapter 5 already gave formula-syntax gotchas their proper depth. The genuine value of this chapter is confirming that Chapters 2 through 4's collaboration concepts — live cursors, comments, version history — hold up identically when applied to numeric, formula-driven data, not just flowing text.

TWO PEOPLE EDITING THE EXACT SAME CELL DOESN'T MERGE CHARACTER-BY-CHARACTER THE WAY SHARED TEXT DOES

Chapter 2 described two people typing in the same paragraph merging both sets of keystrokes automatically. A single spreadsheet cell is a single atomic value, not a flowing sequence of characters — if two people genuinely edit the exact same cell at the exact same moment, Sheets resolves this by having one edit take effect (typically whichever registers last), rather than interleaving both people's input the way text in a Doc can be. This is a sensible consequence of what a cell actually is, not a bug — but it's worth knowing this differs from Chapter 2's own text-merging behaviour before assuming cell edits always combine the same way.

Hands-On Exercises

EXERCISE 1

Share a Sheet with a second account. Have one person edit a raw number in one cell while the other is actively viewing a formula elsewhere that depends on it. Describe exactly what the second person sees happen to the formula's displayed result.

View solution

EXERCISE 2

Two people type genuinely different values into the exact same cell at the exact same moment. Explain what you'd expect to happen to that cell's final value, and why this differs from how two people typing in the same paragraph of a shared Doc would behave, per Chapter 2.

View solution

EXERCISE 3

A colleague coming from LibreOffice Calc asks whether they need to worry about the argument-separator issue from LibreOffice Chapter 5 when writing formulas in Sheets. Explain why this concern doesn't apply here, and identify the one separator character Sheets always uses.

View solution

CHAPTER 5 QUICK REFERENCE

- Sheets' formulas use a **comma** separator always — no locale-dependent semicolon question, unlike Calc
- Common function names (SUM, IF, VLOOKUP, XLOOKUP) transfer with minimal friction
- Live, simultaneous multi-person editing (Chapter 2) applies directly — different cells, or even the same cell
- **Cell-level attribution** shows who last edited a specific cell — finer-grained than Chapter 4's own version snapshots
- A formula recalculates live for everyone the instant its own inputs change, anywhere in the sheet
- Two people editing the exact same cell at once resolves to one winning edit — cells don't merge character-by-character the way shared text does

Every difference the LibreOffice course covered was, at its core, "the same underlying power, organized or reached differently" — Page Styles, the Pivot Table dialog, the Master/Layout split. This chapter is honest about something genuinely different: in several areas, Slides simply has a smaller feature set than PowerPoint or Impress, by design, not because an equivalent capability is hiding somewhere less obvious.

Master Slides in Google Slides

Slide → Edit Master opens a master-editing view, the same "define once, apply everywhere" idea Word & PowerPoint Chapter 8 and LibreOffice Chapter 7 both covered — a presentation can hold multiple master layouts, and editing one updates every slide using it. This part transfers directly, no genuine gap here.

A Deliberately Simpler Theme System

Slides' own Theme picker offers a curated gallery of pre-built themes as the primary, day-to-day workflow — genuine custom master editing exists too, but Slides leans much more heavily toward picking a ready-made theme than building an elaborate custom master from scratch the way PowerPoint or Impress both encourage. This is a real design choice favouring simplicity and speed over deep customization.

Animations: A Smaller Set of Options

Slides offers entrance, exit, and emphasis-style animations too, but with meaningfully fewer effect choices and less granular timing control than PowerPoint's own Animation Pane or Impress's own Custom Animation panel. This is a genuine feature-depth gap — not a differently-organized version of the same underlying capability.

Transitions: Simpler Too

The same pattern holds for slide-to-slide transitions: Slides offers a smaller set of transition styles than either PowerPoint or Impress, with fewer options to fine-tune.

Why This Is Different From LibreOffice's Own Differences

LibreOffice's own genuine differences from PowerPoint were mostly organizational — the underlying capability was still there, just reached or structured differently (Page Styles instead of section breaks, a static Layout dialog instead of a live panel). What this chapter describes is a different kind of difference entirely: Google made a deliberate choice to keep Slides' feature depth smaller, almost certainly prioritizing ease of use and fast, browser-based performance over exhaustive customization options. Neither kind of difference is "wrong" — but conflating them, assuming every Slides limitation must have a hidden equivalent somewhere, would be a genuine mistake.

What Slides Emphasizes Instead

The simplicity is a deliberate trade-off, not just a missing checklist of features. Slides is fast to build a deck in, easy to learn, and — like Docs and Sheets — supports genuine real-time, simultaneous multi-person editing of the same presentation, live, exactly as Chapter 2 described for Docs. Multiple people can build the same deck together, at the same time, watching each other's slides take shape — a capability neither PowerPoint nor Impress offers as a core, built-in feature the same way.

	POWERPOINT / IMPRESS	SLIDES
Animation options and timing control	Many effects, granular timing (Animation Pane / Custom Animation)	Fewer effects, simpler timing controls
Theme customization depth	Deep, hands-on custom Master editing as the norm	A curated theme gallery as the primary workflow
Real-time, simultaneous multi-person slide editing	Limited or newer in PowerPoint; not a core Impress feature	Built in from the start, live, the same as Docs

RECOGNISING A GENUINE GAP IS ITS OWN USEFUL SKILL

LibreOffice Chapter 8 was honest about a real gap in video-export maturity rather than assuming feature parity by default. This chapter applies the same honesty here: Slides' smaller animation and theming depth is a genuine simplification, not a puzzle to solve by hunting for a hidden equivalent feature that isn't actually there.

A COMPLEX POWERPOINT OR IMPRESS DECK MAY SIMPLIFY ON IMPORT

An elaborate custom Master layout or a complex, precisely-timed animation sequence built in PowerPoint or Impress may lose some of that detail when the file is imported into Slides, since Slides genuinely doesn't have an equivalent slot for every possible setting the other two applications offer. Chapter 8 covers file format interoperability in more depth — for now, it's worth expecting some simplification on import rather than assuming a complex deck will translate with everything perfectly intact.

Hands-On Exercises

EXERCISE 1

Open Slide > Edit Master in a new Slides presentation and change the title placeholder's colour. Return to Normal view and confirm which slides updated. Explain why this specific part of Slides works identically to what Word & PowerPoint Chapter 8 and LibreOffice Chapter 7 both described.

View solution

EXERCISE 2

A colleague asks you to recreate a PowerPoint deck's elaborate custom animation sequence (multiple precisely-timed effects per slide) in Google Slides. Explain honestly what to expect, referencing this chapter's own central distinction between an organizational difference and a genuine feature-depth gap.

View solution

EXERCISE 3

Share a new Slides presentation with a colleague and have both of you build different slides in the same deck at the same time. Explain what capability this demonstrates that neither PowerPoint nor Impress offers as a core built-in feature, and connect it back to Chapter 2's own explanation of why this is possible at all.

View solution

CHAPTER 6 QUICK REFERENCE

- **Slide → Edit Master** — the same "define once, apply everywhere" concept as PowerPoint/Impress, transfers directly
- Slides' **Theme gallery** is the primary workflow, favouring speed over deep custom master editing
- **Animations and transitions** have genuinely fewer options and less granular timing than PowerPoint/Impress — a real gap, not just a different panel
- This is a deliberate **simplicity trade-off**, not a hidden equivalent feature waiting to be found
- Slides supports genuine **real-time, simultaneous multi-person deck editing** — not a core feature of either PowerPoint or Impress
- A complex imported deck may lose some elaborate detail — Chapter 8 covers this in depth

LibreOffice Chapter 9 established that VBA and LibreOffice Basic are genuinely different languages — a macro written for one doesn't run in the other. This chapter extends that finding a third way: Google Apps Script is a third, equally distinct automation language, and none of the three are interchangeable with one another.

Recap: Two Macro Languages Already Covered

Excel Advanced Chapters 6 through 8 covered **VBA** (Visual Basic for Applications) — Microsoft Office's own macro language. LibreOffice Chapter 9 covered **LibreOffice Basic**, LibreOffice's own equivalent, explicitly flagged as incompatible with VBA despite both belonging to the same broad language family.

Google Apps Script: A Third, Genuinely Different Language

Apps Script is **JavaScript-based** — not derived from Visual Basic at all, unlike the previous two. It also runs somewhere genuinely different: in the cloud, on Google's own servers, rather than locally on your own machine the way VBA and LibreOffice Basic macros run. Accessed via **Extensions → Apps Script** from within a Doc, Sheet, or Slide, it opens its own separate script editor.

Why It's Genuinely a Different Language, Not Just a Different Dialect

VBA and LibreOffice Basic are both BASIC-family languages — similar syntax shapes (`Sub`/`End Sub`, `Dim`, similar control-flow keywords) even though neither runs the other's code directly. Apps Script is genuinely different at the syntax level: real JavaScript, using `function`/curly braces instead of `Sub`/`End Sub`, no `Dim` declarations, and a completely different underlying object model — a bigger syntactic gap than the one between VBA and LibreOffice Basic.

A Simple Apps Script Example

```
// Sets cell A1 on the active sheet to "Hello"
function setGreeting() {
  var sheet = SpreadsheetApp.getActiveSpreadsheet().getActiveSheet();
  sheet.getRange("A1").setValue("Hello");
}
```

Notice the shape: a `function` declaration, curly braces marking its body, and `SpreadsheetApp` — Apps Script's own object model for working with a Sheet — chained through method calls. Nothing here resembles VBA's or LibreOffice Basic's own `Sub` / `End Sub` structure.

What Apps Script Can Do That the Others Can't

Because Apps Script runs on Google's own servers rather than requiring a desktop application to be open, it can run on a genuine **time-based schedule** — once a day, once an hour — even when nobody has the file open at all. It can also respond directly to events like a Form being submitted or a specific Sheet being edited, triggering automatically without anyone manually running anything. Neither VBA nor LibreOffice Basic can do this, since both require the actual desktop application to be running the file for a macro to execute at all.

Custom Functions and Menu Items

Apps Script can define custom functions callable directly inside Sheets formulas, just like a built-in function, and can add custom menu items to the Docs/Sheets/Slides interface itself — a similar spirit to VBA's own ability to add custom Ribbon buttons, just implemented in JavaScript instead.

	VBA	LIBREOFFICE BASIC	APPS SCRIPT
Language family	Visual Basic	Visual Basic (compatible in spirit, not in practice)	JavaScript
Where it runs	Locally, in the desktop app	Locally, in the desktop app	In the cloud, on Google's servers
Can run with the file closed	No	No	Yes — time-based or event triggers
Runs the other two languages' code	No	No	No

NONE OF THE THREE AUTOMATION LANGUAGES IN THIS SUBJECT ARE INTERCHANGEABLE

LibreOffice Chapter 9 established that VBA and LibreOffice Basic don't transfer to each other. This chapter extends that same finding a third way: automating a workflow in Excel/Word/PowerPoint, LibreOffice, or Google Workspace means committing to that specific ecosystem's own language — none of the three is a portable, write-once-run-anywhere automation layer across all of them.

A VBA-DEPENDENT WORKBOOK MOVED TO SHEETS DOES NOT BRING ITS MACROS ALONG IN ANY FORM

Exactly like LibreOffice Chapter 9's own central warning, moving an Excel workbook containing VBA macros over to Google Sheets does not carry those macros along, even in a broken or partial form — they need to be genuinely rewritten in Apps Script's own JavaScript, understanding what the original VBA logic did and reimplementing it from scratch. Separately, since Apps Script runs on shared Google infrastructure rather than your own machine, it has its own execution time limits and usage quotas — a genuinely different practical constraint that simply doesn't apply the same way to a macro running locally in VBA or LibreOffice Basic, worth checking Google's own current documentation for if you're building something with real automation demands.

Hands-On Exercises

EXERCISE 1

Open Extensions > Apps Script from a Google Sheet, paste in a function that sets a specific cell's value, and run it. Confirm the cell updates. Explain, using this chapter's own vocabulary, why this code's structure looks nothing like a VBA Sub procedure even though both are automating the same basic kind of task.

View solution

EXERCISE 2

A colleague wants a script that automatically emails them a summary every morning at 8am, whether or not the spreadsheet is currently open in anyone's browser. Explain why this is possible with Apps Script but would not be possible with an equivalent VBA macro in Excel, referencing what each one requires to actually run.

View solution

EXERCISE 3

A team wants to migrate an Excel workbook full of VBA automation entirely to Google Sheets, assuming the macros will "just come along" the way the workbook's actual data and formulas mostly will. Explain exactly what will and won't transfer, and what real work is actually required for the automation specifically.

View solution

CHAPTER 7 QUICK REFERENCE

- **Google Apps Script** — JavaScript-based, runs in the cloud, accessed via Extensions → Apps Script
- Genuinely different from both VBA and LibreOffice Basic — a bigger syntax gap than those two share with each other
- Can run on a **time-based schedule** or in response to an event, even with the file closed — VBA and LibreOffice Basic cannot
- Can define custom Sheets functions and add custom menu items, similar in spirit to VBA's own Ribbon customization
- None of VBA, LibreOffice Basic, or Apps Script runs the other two's code — each requires its own genuine rewrite
- Apps Script has its own execution time limits and quotas, a constraint that doesn't apply to a locally-run VBA/Basic macro

LibreOffice Chapter 9 covered interoperability between two genuinely different FILE formats — ODF and OOXML. This chapter is a structurally different question, foreshadowed since Chapter 1: what happens when a real file is converted into a cloud-native object that, per Chapter 1, isn't a file at all anymore — and what happens going the other direction.

A Genuinely Different Kind of Conversion

LibreOffice's own interoperability question was file-to-file: `.docx` in, `.odt` out, both still genuine files the whole time. This chapter's question has an extra wrinkle Chapter 1 already set up: converting *into* Google Workspace means a file stops being a file at all, becoming a live cloud object instead — and converting back out means generating a brand-new file from something that, moments before, had no file behind it.

Uploading a Microsoft or LibreOffice File

Uploading a `.docx`, `.xlsx`, `.pptx`, `.odt`, `.ods`, or `.odp` file to Google Drive offers two genuinely different paths: opening and editing it directly while it stays closer to its original format (sometimes called **Office compatibility mode**), or explicitly converting it into a true, native Google Doc/Sheet/Slide via **File → Save as Google Docs/Sheets/Slides**. These are meaningfully different outcomes, not two names for the same thing.

What Happens During the Conversion

Once genuinely converted to native Google format, the overall pattern is similar to LibreOffice Chapter 9's own framing: most everyday content — text, standard formatting, common formulas, ordinary slide content — survives reasonably well. The specific risk areas worth knowing here: VBA macros do not transfer in any form (Chapter 7's own point, restated in this specific context); elaborate custom PowerPoint animations or deep theme customization may simplify (Chapter 6's own forward-reference, now addressed directly); and some advanced Word field codes or Excel-specific features may have no direct equivalent in Google's own feature set.

Exporting Back Out

File → **Download** works from any Google Doc, Sheet, or Slide, offering Microsoft formats, ODF formats, and PDF, directly. This is the reverse conversion — generating a genuine file from a cloud-native object that, until that moment, had no file behind it at all, exactly the same "Download creates a separate, disconnected copy" behaviour Chapter 1 first established, just now producing a file in whichever format is actually needed.

The "Office Compatibility Mode" Middle Ground

Worth being clear about: opening an uploaded Microsoft file without explicitly converting it lets you view and edit it while it stays closer to its original format, rather than becoming a genuine native Google Doc immediately — a real third option sitting between "still just a file, untouched" and "now a fully native cloud object." It's a reasonable choice when you specifically want to avoid a full conversion, but it's still worth understanding it's a distinct middle path, not simply "the same as opening any other file."

	LIBREOFFICE CHAPTER 9'S QUESTION	THIS CHAPTER'S QUESTION
What's on each side	Two genuinely different FILE formats	A file on one side, a cloud-native object on the other
The underlying risk	Two formats representing overlapping but not identical features	The same, plus one side isn't a file at all
Macros	VBA and LibreOffice Basic don't transfer	VBA and Apps Script don't transfer (Chapter 7)
Going "back"	Saving in the other file format	File → Download, generating a fresh file from a cloud object

RECOGNISING WHICH KIND OF INTEROPERABILITY QUESTION YOU'RE ACTUALLY FACING

This entire subject has now covered three distinct interoperability situations: Excel and Word & PowerPoint's own native formats (no conversion needed at all), LibreOffice's file-to-file conversion (Chapter 9 there), and this chapter's file-to-cloud-object conversion. Recognising which one actually applies to a given situation — and in particular, whether one side of the conversion is even a file in the first place — determines which specific risks are worth checking for.

OFFICE COMPATIBILITY MODE DOES NOT PRESERVE MACROS EITHER

It's tempting to assume that opening a file in the "closer to original" compatibility mode, rather than fully converting it, might preserve more of the original file's behaviour — including its VBA macros. It does not. Regardless of which upload path is used, VBA macros do not run in Google Workspace in any form, exactly as Chapter 7 established — compatibility mode affects how closely the visible formatting and structure resemble the original file, not whether embedded automation code executes.

Hands-On Exercises

EXERCISE 1

Upload a .docx file to Google Drive containing standard text and formatting, but no macros or unusual features. Convert it to a native Google Doc. Based on this chapter, predict whether this specific conversion is likely to cause visible changes, and explain your reasoning.

View solution

EXERCISE 2

A colleague uploads a .xlsm workbook containing VBA macros and opens it in "Office compatibility mode" specifically because they assumed this would keep the macros working. Explain why this assumption is incorrect, referencing this chapter's own warning box.

View solution

EXERCISE 3

Explain, in your own words, why converting a .pptx file into a native Google Slides presentation is a genuinely different KIND of interoperability question than LibreOffice Chapter 9's own .pptx-to-.odp conversion, even though both situations involve moving content between two different presentation tools.

View solution

CHAPTER 8 QUICK REFERENCE

- This chapter's interoperability question is **file ↔ cloud object**, structurally different from LibreOffice's own file ↔ file question
- Uploading offers two paths: **Office compatibility mode** (stays closer to the original) or a full conversion (**Save as Google Docs/Sheets/Slides**)
- Most everyday content survives conversion well; VBA macros, elaborate custom animations, and some advanced field codes are the real risk areas
- **File → Download** is the reverse conversion — generating a fresh file from a cloud object that had none
- Compatibility mode does **not** preserve VBA macros — nothing preserves them; they require a genuine Apps Script rewrite (Chapter 7)

The Excel/Word & PowerPoint and LibreOffice capstones each rebuilt one report-plus-deck project end to end. This capstone takes a genuinely different shape, because Google Workspace's own genuine differences aren't about individual features inside one document — they're about what happens when several people work on that document at the same time. This capstone is a scenario, not a rebuild: a three-person team producing one shared quarterly report, exercising every chapter's own collaboration-first angle along the way.

The Scenario

Three teammates — a writer, a data analyst, and a manager — need to produce a quarterly report together, plus a short companion deck for a stakeholder meeting, all inside a single working day, without ever emailing a file back and forth.

Step 1 — Setting Up the Doc for Real Collaboration

The writer creates a new Google Doc and shares it directly with the analyst and manager, choosing **Editor** access for both — the cloud-native "share the object, not a copy of a file" model Chapter 1 established as the foundation everything else in this course builds on.

Step 2 — Simultaneous Editing in Action

All three open the Doc at once. The writer drafts the opening summary while the analyst adds a data section below it, each visible to the other in real time with their own labelled cursor — the live, simultaneous editing model Chapter 2 contrasted directly against Word & PowerPoint's own asynchronous Track Changes.

Step 3 — Suggesting Mode and Comments for Review

The manager reviews the draft in **Suggesting** mode rather than editing directly, proposing wording changes the writer can accept or reject, and leaves a **Comment** flagging a section that needs the analyst's input — Chapter 3's own pairing of Suggesting mode (Google's live equivalent of Track Changes) with Comments (discussion without editing), both happening in real time alongside the live editing from Step 2.

Step 4 — An Accidental Deletion, and Recovering It

The analyst accidentally deletes an entire paragraph while reformatting a table and doesn't notice right away. Rather than panic, the writer opens **File → Version History**, finds the version from just before the deletion, and restores the missing paragraph — Chapter 4's own continuous, automatic safety net standing in for the manual-save-and-recover habit every other course in this subject assumed by default.

Step 5 — Pulling in Data from Sheets

The analyst builds the supporting numbers in a separate Google Sheet, then inserts a live chart into the Doc linked back to that Sheet, so it updates automatically if the underlying figures change — Chapter 5's own point about Sheets' formulas transferring familiar syntax while adding live, multi-editor collaboration on the same data.

Step 6 — A Companion Slide Deck

The manager spins up a short Google Slides deck summarizing the report's key points for the stakeholder meeting, deliberately keeping it simple — a small Theme, no elaborate custom animation — consistent with Chapter 6's own "simplicity over depth" framing rather than trying to force PowerPoint-level polish out of Slides.

Step 7 — Automating the Weekly Status Email

Rather than manually remembering to notify the wider team each Monday, the analyst writes a short Apps Script function, attached to a time-based trigger, that emails a summary of the report's current state every Monday at 8am — Chapter 7's own example of Apps Script's cloud-native execution doing something no locally-running macro language could replicate without external help.

Step 8 — Sharing the Final Report Externally

The external stakeholder uses Microsoft Word, not Google Docs, so the writer uses **File → Download** to export the finished report as a `.docx` file before sending it along — Chapter 8's own reverse conversion, generating a genuine file from a cloud-native object that had none, going the opposite direction from Step 1's own upload-free starting point.

CAPSTONE STEP	CHAPTER IT DRAWS FROM
Step 1 — Sharing the Doc directly	Chapter 1 — Cloud-native, not file-based
Step 2 — Simultaneous editing	Chapter 2 — Real-time collaboration
Step 3 — Suggesting mode and Comments	Chapter 3 — Suggesting and Comments
Step 4 — Recovering the deletion	Chapter 4 — Version History
Step 5 — Linked chart from Sheets	Chapter 5 — Sheets vs. Excel/Calc
Step 6 — The companion deck	Chapter 6 — Slides vs. PowerPoint/Impress
Step 7 — The Monday email automation	Chapter 7 — Google Apps Script
Step 8 — Exporting for an external stakeholder	Chapter 8 — File Format Interoperability

WHY THIS CAPSTONE LOOKS DIFFERENT FROM THE OTHER TWO

Excel/Word & PowerPoint's and LibreOffice's capstones each rebuilt a single, self-contained document-plus-deck project. This one is deliberately a multi-person scenario instead, because Google Workspace's own genuine differences — live simultaneous editing, Suggesting mode and Comments happening in real time, continuous Version History, and Apps Script's cloud-native execution — only actually show up when more than one person, and more than one moment in time, are involved at once.

A REMINDER THIS COURSE KEPT RETURNING TO

Nothing about this scenario involves VBA or LibreOffice Basic macros carrying over into Apps Script, or a locally-running application being required for the Monday email to send — both points this course made explicitly, in Chapter 7 and Chapter 8, specifically because they're the two places a habit carried over from Excel, Word & PowerPoint, or LibreOffice would otherwise lead to a wrong assumption.

Hands-On Exercises

EXERCISE 1

Recreate Steps 1–4 of this capstone yourself with a real Google Doc (solo is fine — simulate the "second editor" by making changes from an incognito window signed into a second account, or simply narrate what each step would look like). Note one moment where Version History's continuous snapshot model gave you an option a manual "Save As" habit wouldn't have.

View solution

EXERCISE 2

Write the Apps Script function (and describe the trigger you'd set up) that the analyst in Step 7 would actually need to send the Monday 8am status email. It should read the Doc's current word count and email it to a fixed address.

View solution

EXERCISE 3

The external stakeholder in Step 8 later replies with tracked edits made in Microsoft Word, and re-sends the .docx file. Explain, using this course's own chapters, what happens when the writer uploads that edited file back into Google Drive, and what the writer should check for before assuming everything transferred cleanly.

View solution

COURSE COMPLETE

Google Workspace — 9 of 9 chapters complete. This closes the Office & Productivity Software subject's fourth and final planned course.

CHAPTER 9 QUICK REFERENCE

- This capstone is a **collaboration scenario**, not a document rebuild — the shape that actually exercises this course's own angle
- **Every chapter** maps to one concrete step: cloud-native sharing, live editing, Suggesting/Comments, Version History, Sheets data, Slides, Apps Script, and file export
- The two reminders repeated throughout: **macros never transfer** to Apps Script, and Apps Script runs **server-side**, independent of anything being open