# POO²L Language Reference

# POO²L Language Reference

*A Reader's Companion to the v1.0 Specification*

InZeMobile

# TABLE DES MATIÈRES

**Partie 1 — Concepts**

Chapter I — Why POO²L Exists ..... 9
Chapter II — The Six Maturity Levels ..... 17
Chapter III — The Processing Model in Eleven Phases ..... 29
Chapter IV — Terms and Definitions Cheatsheet ..... 41
Chapter V — How to Read This Book ..... 51

**Partie 2 — Building blocks (single-template scope)**

Chapter VI — Identity ..... 59
Chapter VII — Sections ..... 68
Chapter VIII — Comments ..... 88
Chapter IX — Verbatim Blocks ..... 97
Chapter X — System Variables ..... 105

**Partie 3 — Variables and binding**

Chapter XI — Parameters ..... 115
Chapter XII — Constraints ..... 125
Chapter XIII — Assertions and Scopes ..... 133
Chapter XIV — Constants ..... 141
Chapter XV — Parameter Sets ..... 148
Chapter XVI — Attachments ..... 157

**Partie 4 — Composition (multi-template scope)**

Chapter XVII — Includes ..... 169
Chapter XVIII — Includes with Parameter Overrides ..... 178
Chapter XIX — Inheritance ..... 185
Chapter XX — `parent` and `self` ..... 194
Chapter XXI — Folders ..... 204
Chapter XXII — Collections ..... 213

Chapter XXIII — Collection Inheritance, Virtual and Final Templates ..... 223
Chapter XXIV — Section Composition Semantics ..... 235

**Partie 5 — Lifecycle and governance**

Chapter XXV — Aliases ..... 245
Chapter XXVI — Deprecation ..... 254
Chapter XXVII — Versioning Policy ..... 262
Chapter XXVIII — Dependency Management ..... 270
Chapter XXIX — Access Control Across Groups ..... 277

**Partie 6 — Reference**

Chapter XXX — Resolution Phases in Depth ..... 287
Chapter XXXI — Error Catalog ..... 302
Chapter XXXII — Conformance Levels ..... 321
Chapter XXXIII — The Closed-Extension Policy ..... 330
Chapter XXXIV — Implementer Notes ..... 336

**Appendices**

Appendix C — Appendix C — Anti-Patterns and Refactorings ..... 347
Appendix D — Appendix D — Cookbook ..... 357
Appendix E — Appendix E — Crosswalk to `standard.md` ..... 367

Lexique ..... 376

Index thématique ..... 387

Index alphabétique ..... 393

# Partie 1

*Concepts*

# Chapter I — Why POO²L Exists

> *« This specification is COMPLETE and CLOSED. There are no planned extensions, variants, or alternative syntaxes. » — POO²L Specification v1.0, §1.*
>
> ***`standard.md §1` — SCOPE[1] (excerpt) :***
>
> *POO²L is a structured language for engineering Large Language Model templates with progressive architectural complexity across six levels. It applies established software engineering principles (modularity, INHERITANCE, COMPOSITION) to TEMPLATE management in production LLM systems. The specification is COMPLETE and CLOSED : no control flow (if/else, loops), no dynamic code generation, no macro systems beyond defined mechanisms, no alternative syntaxes with different semantics. This ensures POO²L remains deterministic, testable, and maintainable.*

## 1.1 — The trouble with plain text

A prompt is a piece of writing. For the first few months of a project, that is enough. You open a text editor, you write what you want the model to do, you paste it into a chat. The whole thing fits on a postcard. The team shares a Google Doc, and life is good.

Then the prompt grows. A colleague adds a paragraph. Another colleague clones the file into a FOLDER called `templates/` because the Doc is getting hard to scroll. A third colleague needs *almost* the same prompt for *almost* the same task, so they copy it, modify three lines, and save it under a new NAME. Six weeks later, twelve files

1. ***Les termes en petites capitales renvoient au lexique.***

contain the same opening sentence, drifted into seven variants of itself. Nobody remembers which file was the one that worked best in production.

This is the moment a prompt stops being a piece of writing and starts being a *system*. And systems need a language.

Four pathologies appear, predictably, in every team that scales past a handful of templates :

- **Duplication.** The same paragraph appears in twelve files. When the business decides the phrase « in a professional tone » should become « in a clear, plain-spoken tone, » you do not know if you found every copy.
- **Drift.** A prompt that worked in March no longer works in October — not because the model changed, but because each successive editor made a small, well-intentioned tweak. Nobody can point at the change that broke it.
- **Hidden coupling.** A prompt mentions the format of a date, and another prompt elsewhere parses that date. When you edit one, the other silently breaks. The dependency was real ; it was never written down.
- **Untraceable changes.** Versions live in the names of files : `review_v2.txt`, `review_v2_final.txt`, `review_v2_final_FINAL.txt`. The intent of each change is lost. The decision log is the chat thread above each file, scattered across half a dozen apps.

These are not new problems. Software engineering has spent fifty years building tools for them. POO²L brings those tools, intact, to prompt management.

## 1.2 — Four linguistic answers

POO²L is not a framework, not a library, and not a runtime. It is a *language* — a small set of grammar rules that turn a piece of writing into a structured artefact. Four answers, in order of increasing power :

**1. IDENTITY.** Every prompt has a name, an `id`, and a DESCRIPTION. Those three fields are not decoration ; they are how other templates refer to this one. A prompt without an identity is anonymous, and anonymous things cannot be reused. Identity is the first promise the language enforces.

**2. Parameterization.** A prompt declares the values it expects to vary at use time. The author writes `@@@param tone : select("formal", "casual") = "formal", "Tone"`, and the dozen near-duplicate « tone variants » collapse into a single, parameterized prompt. The values that should *not* vary — the brand name, the company signature, the safety preamble — are declared as constants, and the PARAMETER dialog no longer asks the user about them. The variable and the invariant become separate, named things.

**3. Composition.** A prompt includes another prompt by reference. The shared safety preamble lives in one file, and twelve other templates INCLUDE it. The « glossary block » used by every customer-support prompt lives in one file. When the wording changes, it changes in one place. Composition is what makes a *library* of templates possible, as opposed to a *folder* of templates.

**4. Inheritance.** A prompt declares itself a *specialization* of another. A `code-reviewer` base prompt describes how to review code ; a `python-reviewer` child inherits all of that and adds Python-specific instructions. The child does not duplicate the parent ; it extends or overrides specific sections. When the parent improves, every child improves at the same time.

These four answers — identity, parameterization, composition, inheritance — are the same four ideas that turned text-file programming into modern software. POO²L applies them, unchanged, to the artefact that is now central to LLM systems : the prompt.

## 1.3 — What « Object-Oriented Squared » means

POO²L's name is *Prompt Object-Oriented²* — *squared*. The exponent is not a flourish. It signals that two object-oriented axes operate at once.

The first axis is **template-level**. A prompt is treated as a class : it has identity, fields (parameters and constants), methods (sections that can be overridden), and an `inherits` relation that produces specialization. This is the axis you reach as soon as you start writing templates at scale.

The second axis is **Collection-level**. A *collection* — POO²L's name for a directory of related templates — is itself treated as a class. The members of the collection are the *methods*. A collection can declare *virtual templates* (abstract members that derived collections must implement) and can inherit from another collection. A `reviewer` collection might declare three virtuals — `summarize`, `score`, `suggest` — and derive into `code_reviewer` and `prose_reviewer`, each providing its own concrete implementations.

The two axes are orthogonal. You can use either one without the other, or both together. The first axis specializes a single prompt ; the second axis specializes a *family* of related templates. Together, they cover the cases real teams hit when their prompt library grows past a few dozen artefacts.

POO²L's six architectural levels — L0 through L5, detailed in the next chapter — map directly to *which* of these answers you have adopted :

- **L0** uses none of them (plain text).
- **L1** uses identity and structure (sections).

- **L2** adds composition (includes).
- **L3** adds structured data and validation (typed parameters, constraints, assertions).
- **L4** adds template-level OO (inheritance) — the first axis.
- **L5** adds collection-level OO — the second axis, the *squared*.

You do not have to start at L5. Most teams begin at L0, move to L1 the first time someone wants to OVERRIDE a single SECTION, reach L2 the day they extract a glossary, reach L3 the first time a prompt needs typed and validated inputs, and reach L4 the first time they realize a base/specialization relationship would erase a class of duplications. L5 is the level where prompt libraries start to look like the codebases that the field of software engineering has been refining for decades.

## 1.4 — Why the specification is closed

POO²L version 1.0 is, by design, complete and closed. The specification states it without ambiguity (§1) :

- No control flow (if/else, loops, while).
- No dynamic code generation.
- No macro systems beyond the defined mechanisms.
- No alternative syntaxes with different semantics.

A reader new to language design might find this surprising. Languages usually *grow*. Why does POO²L refuse to ?

Three reasons.

**Determinism.** A prompt that compiles today must compile the same way next year, on every conformant implementation, regardless of when the implementation was built. The moment a language allows control flow or dynamic generation, two implementations can disagree about what a prompt means. The specification's closure is what allows the team that wrote a prompt at L4 in 2026 to trust that the prompt will still render the same way in 2030.

**Testability.** Every POO²L construct can be statically analyzed. Inheritance chains can be checked for cycles. Parameter constraints can be validated at definition time. Virtual templates can be checked for completeness. A language with arbitrary control flow loses these guarantees, and with them, the ability to give engineering-grade quality assurances about the prompt library.

**Fragmentation resistance.** Open-ended languages fragment. JavaScript, SQL, regular expressions — every widely-used language with an « extensible » syntax ends up with a thicket of dialects that nominally share a name but in practice cannot run each other's code. POO²L's closure is what allows the name « POO²L » to mean the same thing in every implementation that claims it.

The specification is closed ; the *implementations* are not. Any team is free to build a POO²L authoring tool, a renderer, a linter, an editor plug-in, a continuous-integration check. IziBuddy — the reference implementation cited throughout this book — is one such tool. Others will follow. They will compete on user experience, performance, and integration ; they will not compete on what the language means.

A closed specification is not a refusal to improve. It is a refusal to break the contract. If a v2 of the language ever happens, it will be a *new specification* with its own version number ; v1.0 templates will continue to compile under v1.0 implementations, indefinitely.

## 1.5 — What this book is for

`POO²L/standard.md` — the specification document — is the authoritative reference. Every clause in this book that describes the language is derived from a clause in the specification, and every chapter cites the specification's section numbers (`standard.md §X.Y`) so the reader who wants the normative version can find it in one step.

This book reorganizes that material for a different reader : not the implementer building a parser, but the prompt engineer using the language. Every construct is introduced in the same five-section

template — **Syntax**, **Purpose**, **Possibilities**, **Related features and why**, **Practical examples** — so that learning a new construct never requires inventing a new mental scaffold. The order of chapters follows the natural progression of a team : identity and sections first, then parameters and constants, then composition, then the two OO axes, then governance.

The reader who already knows a POO²L authoring tool by its surface will find this book a natural deepening. The reader who knows software-engineering OO but not POO²L will recognize most of the ideas under different syntactic clothes. The reader new to both will find the chapters of Part I (Concepts) designed to ground the mental model before any construct is introduced.

A short reading map appears in Chapter V ; the construct chapters of Parts II–IV can be read in any order once the concepts of Part I are in place. The reference material in Part VI is intended for lookup, not narration.

We begin, in the next chapter, with the six architectural levels.

## Skills use the same composition

These are two separate registry sources :

```
@@@template REVIEW_BASE
@@@section evidence
Support each finding with an observed example.
```

```
@@@template REVIEW_HELPER inherits REVIEW_BASE
@@@type skill
@@@description Use when reviewing a change and explaining actionable
@@@section process
Read the change, verify its effects, and report relevant findings.
```

Rendering REVIEW_HELPER yields a skill. Under host policy `allow` it includes the base instructions and warns `MIXED_DOCUMENT_INHERITANCE`. Under `deny`, that edge is rejected. Includes, parameters and collection membership work as for templates. Drafts and publication are host features; no `@@@draft` exists.

## Templates and document nature

A **template** is the common reusable entity. Its **Nature** is `prompt`, `skill` or a custom identifier such as `mod` or `acme.review`. Create **New prompt** or **New skill**, then edit Nature in Properties when needed. Changing nature preserves the UUID, references, parameters and history. The leaf template owns its type; includes never change it. Mixed inheritance can be allowed with warnings or denied.

```
@@@template review_assistant
@@@type skill
@@@description Reusable review instructions
@@@section rules
Separate observations from assumptions.
```

Use `type:acme.review` to search a custom nature. The **Example** flag replaces the old Template flag; `flag:example` finds these starting examples. `flag:draft` continues to find private drafts. A prompt copies the rendered text, a skill exports SKILL.md and resources, and other types export composed text without claiming a specialized package format.

# Chapter II — The Six Maturity Levels

POO²L describes a prompt library not as a single artefact but as a position along a maturity curve. The specification calls these positions *levels*, numbered zero through five, and the choice of level is not academic : it tells you which constructs you may use, which guarantees the language gives you in return, and which kind of conformance test your tooling must pass.

This chapter walks the six levels, end to end, and gives the criterion for choosing one. The order in which a team moves up the curve is rarely a deliberate decision — it is the order in which their library forces them to.

## 2.1 — The six levels at a glance

| Level | Name | What you gain | What you give up |
|---|---|---|---|
| L0 | Unstructured | Nothing to learn ; immediate productivity. | Every later improvement is harder. |
| L1 | Structured | Named sections ; the prompt is now navigable. | A small syntactic discipline. |
| L2 | Procedural | Includes ; one place for one fact. | A second file to track. |
| L3 | Structured Data and Validation | Typed parameters, constants, attachments, constraints, assertions, lifecycle metadata. | Declaring and validating your data surface. |
| L4 | Object-Oriented | Template inheritance ; specialization without duplication. | A mental model of « parent / child ». |

| Level | Name | What you gain | What you give up |
|---|---|---|---|
| L5 | OO² | Collections as classes ; the second axis. | A `@@@collection` declaration layer ; a richer toolchain. |

Each level is a *strict superset* of the one below it. An L5 implementation accepts L0 templates unchanged, and an L0 prompt is a valid L1 prompt that happens to define zero sections. There is no migration step from level to level ; there is only an editing step that adds a new construct.

## 2.2 — L0 — Unstructured

Spec `standard.md` §4.1. The starting point : a prompt is a piece of plain text. No delimiters, no sections, no metadata. The specification is explicit that L0 is a valid POO²L level : every conformant implementation must accept and process unstructured templates.

> ***`standard.md §4.1` — Level 0 : Unstructured (excerpt) :***
>
> *Templates exist as unstructured natural language text without formal sections, includes, or compositional mechanisms. Level 0 is a VALID POO²L level — implementations MUST accept and process unstructured templates as valid POO²L input. Intended for exploratory prototyping, one-off queries, and simple use cases.*
>
> *You are an expert Python developer. Analyze this code and suggest improvements focusing on performance and readability.*

This is a complete L0 prompt. It is also a complete L1, L2, L3, L4, and L5 prompt — the levels are not gates that an artefact passes through ; they are simply the *constructs* that a given prompt happens to use.

**Use L0 for** exploratory prototyping, one-off queries, and the first prompt of any new project. The moment the same prompt starts appearing in multiple places with small variations, you have outgrown L0.

## 2.3 — L1 — Structured

Spec `standard.md` §4.2. The prompt is organized into *named sections*. The syntactic cost is one delimiter per section:

> ***`standard.md §4.2` — Level 1: Structured (excerpt):***
>
> *Templates are organized into named sections serving distinct purposes. Required elements: section delimiters, section naming convention, content encapsulation. Sections are OPTIONAL — a prompt may define zero, one, or multiple sections; there are NO mandatory sections. Common section names (`role`, `context`, `task`, `constraints`, `examples`, `format`) are examples, not normative.*

```
@@@section role
You are an expert in domain X.

@@@section task
Analyze the following input...
```

Sections are *optional*. There are no mandatory section names; common ones — `role`, `context`, `task`, `constraints`, `examples`, `format` — appear in the spec as examples, not requirements. A prompt may define zero, one, or fifty sections.

What L1 gives you is the ability to *target* a fragment. The moment a section has a name, another prompt can reference that section by name. That single move — naming the parts — is the precondition for everything in L2, L3, L4, and L5. → §VII covers section syntax in detail.

**Use L1 for** templates that are starting to grow past one screen and where a reader might benefit from skimming to a specific part. You reach L1 the day someone asks « where is the task in this prompt ? ».

## 2.4 — L2 — Procedural

Spec `standard.md` §4.3. The prompt may *include* another prompt. The shared safety preamble lives in one file ; twelve other templates include it. The « glossary block » used by every customer-support prompt lives in one file.

> ***`standard.md §4.3` — Level 2 : Procedural (excerpt) :***
>
> *Templates support modular composition through external references, enabling reusability and DRY. Required elements : include mechanism, reference resolution, namespace or path specification. Characteristics : content reusability across templates, single source of truth for shared components, hierarchical organization of prompt libraries.*

```
@@@include common/expert_role
@@@include domain/constraints

@@@section task
Specific task content...
```

L2 is the first level at which the *library* matters more than any individual prompt. The trade-off is that you now have to track where a fragment lives, but the gain is that an edit propagates everywhere it should.

L2 also introduces *cycle detection* as an implementation requirement (a prompt cannot include a prompt that includes it back). → §XVII covers includes in depth.

**Use L2 for** any library above ten templates. The day you copy-paste a paragraph for the third time, you are paying the L2 tax in duplication and not collecting the benefit.

## 2.5 — L3 — Structured Data and Validation

Spec `standard.md` §4.4. The prompt carries *typed, validated data* and lifecycle metadata. L3 adds the data layer — parameters, constants, attachments, and parameter sets — declarative parameter constraints, lifecycle markers (`@@@deprecated`, `@@@alias`), and the validation layer (`@@@assert` within `@@@scope`) that checks the rendered output against declared expectations.

> ***`standard.md §4.4` — Level 3: Structured Data and Validation (excerpt):***
>
> *Templates carry typed, validated data and lifecycle metadata. L3 adds the data layer (parameters, constants, attachments, parameter sets), declarative parameter constraints (`min` / `max` / `minlen` / `maxlen` / `pattern`), the `select` type, lifecycle markers (`@@@deprecated`, `@@@alias`), and the validation layer (`@@@assert` + `@@@scope`). This level makes a prompt parameterizable, bindable, and SELF-validating WITHOUT yet introducing either Object-Oriented axis.*

```
@@@template client_analysis_v1
@@@name Client Analysis
@@@param tier : select("basic", "pro", "enterprise") = "pro"
@@@const VERSION : text = "2.0"

You are analyzing the client (@@@=tier)...
```

L3 is the level at which a prompt stops being a fixed string and becomes a *generic template*: inputs are typed and validated before substitution, and the rendered output can be asserted against declarative expectations. No template-inheritance or collection semantics yet — those are L4 and L5.

**Use L3 for** any prompt that takes variable inputs which must be typed, bounded, or validated, or whose rendered output must satisfy a checkable contract. → §XI (Parameters), §XII (Constraints), §XIV (Constants), §XV (Parameter sets), §XVI (Attachments), §XXV (Aliases) and §XXVI (Deprecation) cover the data and lifecycle layer.

## 2.6 — L4 — Object-Oriented

Spec `standard.md` §4.5. A prompt may declare itself a *specialization* of another prompt. This is the first Object-Oriented axis (OO #1): the vertical, template-level axis where sections behave as methods that can be inherited and overridden.

> ***`standard.md §4.5` — Level 4: Object-Oriented (excerpt):***
>
> *Templates support inheritance, enabling specialization and derivation of prompt families. This is the first Object-Oriented axis (OO #1). Required elements: all L3 capabilities, parent prompt reference, inheritance resolution mechanism, override capability. Semantics: child inherits all sections (and the parent's L3 data layer), may override parent sections, may add new sections; multiple levels of inheritance supported; circular inheritance prohibited.*

```
@@@template specialized_analyst inherits expert_analyst
@@@name Specialized Analyst

@@@section task
Specialized analysis task...
```

The child inherits all sections from the parent, may override any of them, and may add new ones. Multiple levels of inheritance are supported ; circular inheritance is forbidden.

L4 is the first level at which the difference between *form* and *value* is enforced by the language. An include reuses a fragment ; an inheritance reuses a *contract*. The two are not interchangeable — picking one over the other is one of the most common choices a POO²L author makes, and one of the most consequential. → §XIX (Inheritance) and §XVII (Includes) are the chapters to read side by side when in doubt.

**Use L4 for** any library where the same base pattern has more than two variants. A `code_reviewer` base prompt with `python_reviewer`, `go_reviewer`, `prose_reviewer` children is the canonical case.

## 2.7 — L5 — OO² (Object-Oriented Squared)

Spec `standard.md` §4.6. The most powerful level. L5 adds the *second* OO axis : a *collection* of templates is itself treated as a class.

> ***`standard.md §4.6` — Level 5 : OO² (excerpt) :***
>
> *Both axes of Object-Orientation are realized : the vertical (template-level, sections-as-methods, established at L4) and the horizontal (collection-level, templates-as-methods of a collection). Required elements include all L4 capabilities plus collection inheritance (the `inherits` clause on a `@@@collection` block header), VIRTUAL PROMPT declaration (the `virtual` modifier on a `@@@template` header), FINAL PROMPT declaration (the `final` modifier on a `@@@template` header), member override by ID in a derived collection, and the collection-axis references `^parent.<template_id>` / `^parent.<template_id>.<section>` / `^self.<template_id>` (the unary `^` operator names the collection axis, disjoint from the template-axis `parent` / `self`).*

A collection is declared by a `@@@collection <name>` block header that lists its public members, and a prompt joins it by qualifying its header id (`@@@template <collection_name>.<template_id>`). A collection may:

- Inherit from another collection (the inline `inherits <parent>` clause on the `@@@collection` header).
- Contain *virtual templates* — members whose `@@@template` header carries the `virtual` modifier, whose parameter list is the abstract signature that derived collections must implement.
- Be overridden member-by-member in a derived collection.
- Be navigated from inside one of its members using the collection-axis references `^parent.<template_id>` and `^parent.<template_id>.<section>` (the `^` prefix marks the collection axis; the plain `parent` without `^` stays on the prompt axis).

The first axis (spec `standard.md` §4.5) specializes a *single* prompt. The second axis (`standard.md §4.6.1`) specializes a *family* of related templates. The two axes compose orthogonally: a derived prompt may live inside a derived collection and participate in both chains at instantiation time.

> ***`standard.md §4.6.1` — Template Collections (excerpt):***
>
> *A prompt collection is the OO² (horizontal-axis) entity of POO²L. Its methods are the templates that belong to it. A collection may inherit from another collection, declare abstract (virtual) prompt members that derived collections MUST implement, and a prompt rendered inside a derived collection may reference the parent collection's templates via the collection-axis reference `^parent.<template_id>`. Collections NEST by containment (a collection may contain sub-collections, to unbounded depth); a member's qualified id has as many segments as the containment run requires. Virtual templates are*

> *declared at the **prompt level** (the `virtual` modifier on the `@@@template` header), not at the collection level — the prompt's parameters constitute its abstract signature.*

The data and lifecycle machinery — parameters with constraints, parameter sets bundled per prompt or per collection, attachments in the two modes (`fixed` / `to-include`), declarative `select` options, deprecation, and aliases — belongs to L3 and is available to every L5 collection by virtue of cumulativity.

**Use L5 for** the prompt libraries that look like codebases — multiple teams, multiple roles, multiple flavours of the same role, and a deployment process that needs to be auditable. L5 is also where the second OO axis pays off: when a `reviewer` collection has three abstract methods (`summarize`, `score`, `suggest`) that *every* concrete reviewer must implement, the type system catches the missing implementations before any prompt is rendered.

## 2.8 — Folders versus collections

Spec `standard.md §4.7`. POO²L makes one distinction that is easy to miss: a *folder* is a pure location addressed with `/`, whereas a *collection* is a POO²L entity addressed with `.`. A prompt's folder placement is physical and implicit (where its file is stored); its collection membership is declared explicitly by qualifying its header id (`@@@template <collection_name>.<template_id>`), with the collection itself opened by a `@@@collection` block. Whether the target is a folder or a collection decides what membership *means*.

> ***`standard.md §4.7` — Folders (excerpt):***
>
> *A folder is a pure organizational grouping of templates, analogous to a file-system directory. Folders carry NO compositional semantics: no inheritance, no virtual members, no `parent` resolution. Available at ALL levels (L0–L5). A folder is addressed with `/`; a collection is a POO²L entity addressed with `.`.*

> *There is no `kind` flag: an entity is a collection precisely because a `@@@collection` block declares it (an OO² entity, L5 required), and a folder because it is a storage location with no such declaration. An implementation conforming below LEVEL 5 MUST NOT silently give a `@@@collection` block a DEGRADED meaning: faced with a construct it does not implement, it MUST either reject the input or preserve it unrendered and refuse to render that collection — never quietly treat the members as a flat folder namespace, which would make the same source yield different results across engines.*

| **Folder** | **Collection** |
|---|---|
| Pure organization. | OO² class. |
| Available at all levels (L0–L5). | Requires L5 conformance. |
| No inheritance, no virtuals, no `parent`. | All of those. |
| Acts like a file-system directory; addressed with `/`. | Acts like a base class; addressed with `.`. |
| A storage location with no in-language declaration. | Declared in-language by a `@@@collection` block. |

The same prompt can be moved between a folder and a collection without changing a single character in its body. The promotion is in-language: a `@@@collection` block is added (and the members' `@@@template` ids are qualified with the collection name), and the OO² machinery activates. → §XXI (Folders) and §XXII (Collections) cover the two halves of this distinction.

## 2.9 — How teams actually move up the curve

A library rarely starts at the level it ends up at. The typical progression follows the pathologies of Chapter I :

1. **L0 → L1** the first time someone wants to override a single piece of a long prompt without rewriting the whole thing.
2. **L1 → L2** the first time someone extracts a shared paragraph into its own file.
3. **L2 → L3** the first time a prompt needs typed, validated inputs — parameters with bounds, a `select` list, or an ASSERTION on the rendered output.
4. **L3 → L4** the first time someone realizes that a *family* of templates share a base contract and the include relationship is the wrong fit.
5. **L4 → L5** the first time the team's organizational structure (departments, products, languages) starts to mirror the inheritance structure of the templates themselves.

The progression is incremental. There is no « L0 to L5 migration » — each move adds a single construct. A team that has reached L5 is still using L0 prose inside its sections.

## 2.10 — Choosing a target level for a team

The level a team writes *to* — the highest construct it permits in its library — is a governance decision, not a syntactic one. Three considerations :

- **Audit needs.** Higher levels give you more leverage but more places to inspect. L5 lets a single virtual change ripple through every derived collection ; that is a feature when the change is correct and a footgun when it is not. Teams with rigorous audit needs sometimes deliberately freeze themselves at L4.

- **Toolchain availability.** Editor support and linting for L5 are heavier than for L2. An implementation that does not advertise L5 conformance must not silently degrade a `@@@collection`

block to a folder; it must either reject the input or refuse to render that collection (`standard.md §4.7`, *conformance note*). A library targeting multiple implementations should pick the highest level supported by the *least* capable target.

- **Team familiarity with OO.** L4 and L5 expose the same set of concepts that confuse newcomers to object-oriented software design: inheritance, polymorphism, virtual methods, the substitution principle. A team that has never written a class before will pay tuition the first time it writes a deep `inherits` chain.

The next chapter walks the *Processing model* — the eleven phases that a conformant implementation executes when rendering a prompt at any level.

# Chapter III — The Processing Model in Eleven Phases

A POO²L prompt is a piece of declarative input. Turning it into the text that the language model actually sees is the job of a *processor*. The specification tells us exactly how that job is done : eleven phases, executed in a fixed order, with well-defined inputs and outputs. The order is normative — `standard.md §6.1` requires that conformant processors execute the phases in this sequence — and the determinism it guarantees is one of the language's central promises.

> ***`standard.md §6.1` — Resolution Phases (excerpt) :***
>
> *POO²L processors SHALL execute the following phases in order : PHASE 1 Parsing — tokenize, build AST, extract inline declarations, register aliases, check deprecation, validate static syntax. PHASE 2 Collection context application — resolve collection inheritance chain, compute effective prompt set, validate virtuals. PHASE 3 Template inheritance resolution — merge parent sections and L3 data layer, apply overrides (PHASE 3.b validates the call contract). PHASE 4 Section selection — apply the `?/!` toggle map, mark OFF sections inert. PHASE 5 Parameter binding (constants are resolved at the start of this phase). PHASE 6 CONSTANT resolution. PHASE 7 ATTACHMENT binding. PHASE 8 Include resolution — resolve external references, then validate interfaces, detect cycles. PHASE 9 Parameter substitution — substitute values, compute the required-PARAMETER SET, validate constraints. PHASE 10 Assertion evaluation — validate scope nesting, evaluate `@@@assert` predicates against rendered content, block on failure. PHASE 11 Output generation — assemble final text, strip every directive.*

This chapter walks the eleven phases at the level a *reader* needs to understand. Part VI (§29) revisits the same material at the level an *implementer* needs, with detailed inputs and outputs per phase.

## 3.1 — Why the order matters

A POO²L prompt may declare constants, inherit from a parent, include other templates, and use parameters whose values come from a parameter set or are provided at instantiation time. Each of these mechanisms wants something from the others : an included prompt may use parameters, an inherited prompt may declare constants, a parameter set may bind a parameter whose declaration is itself in the parent.

If the resolution order were left to the implementation, two implementations could disagree about what a prompt means. POO²L removes that ambiguity by fixing the order once and for all. A prompt that compiles today will compile the same way next year, on every conformant implementation, regardless of how the implementation is built internally.

## 3.2 — The eleven phases

The phases below correspond to `standard.md §6.1` PHASE 1 through PHASE 11. The IziBuddy reference implementation delegates rendering to the Rust POO²L engine (`src/engine/crates/pool-core`, consumed through its bindings) ; other implementations may organize the code differently as long as the *observable* behaviour is unchanged.

### Phase 1 — Parsing

The input text is tokenized. Delimiters and keywords are recognized ; the abstract syntax tree is built. Inline declarations — parameters, constants, attachments — are extracted from the body. Aliases are resolved : references that use an ALIAS are rewritten in terms of the canonical id. Deprecation markers are noted and their warnings are scheduled for emission.

After Phase 1, the rest of the pipeline works on a clean tree, not on raw text. Syntax errors are reported here (INVALID_SYNTAX).

### Phase 2 — Collection context application

If the prompt is a member of a collection (its `@@@template` header id is qualified with a collection name), the collection's *inheritance chain* is resolved root-first. The chain is walked starting from the leaf collection toward its `inherits` ancestors ; cycles are detected (CIRCULAR_COLLECTION_INHERITANCE), and any collection whose `inherits` clause names an entity that has no `@@@collection` block (a plain folder) is rejected (UNKNOWN_COLLECTION).

The chain produces the *effective prompt set* of the leaf collection : the union of all member templates along the chain, keyed by prompt id, where the closest declaration wins (member override). Virtual templates are validated : every id whose `@@@template` header carries the `virtual` modifier somewhere in the chain must have a non-virtual concrete implementation in the leaf collection or its descendants relevant to this render — otherwise UNIMPLEMENTED_VIRTUAL_TEMPLATE.

Collection-level parameter sets are merged root-first along the chain.

A prompt whose `@@@template` header is unqualified (it belongs to no collection) skips the inheritance sub-steps ; folders carry no compositional semantics ( `§4.7` ).

> ***`standard.md §4.7` — Folders (excerpt) :***
>
> *A folder is a pure organizational grouping of templates, analogous to a file-system directory. Folders carry NO compositional semantics : no inheritance, no virtual members, no `parent` re-*

> *solution. A prompt with an unqualified `@@@template` header skips the OO² sub-steps of Phase 2; only collection-level configurations are applied.*

### Phase 3 — Template inheritance resolution

If the prompt's `@@@template` header carries an inline `inherits <parent>` clause, the parent's sections — and its L3 data layer (parameters, constants, embedded parameter sets, attachments) — are merged into the child. Overrides are applied by name: a child-declared section with the same name as a parent section replaces the parent's content. New child sections are appended (or placed by an explicit placement clause). Multiple levels of inheritance are walked. Cycles are detected (CIRCULAR_INHERITANCE).

After Phase 3, the prompt has a single, flat effective definition — the OO #1 chain has been collapsed. Phase 3.b then validates the **call contract** on every public override: an overriding section or prompt may not break a call that was valid against the inherited one.

### Phase 4 — Section selection

The `?` / `!` presence-toggle map (the caller's instantiation choices) is applied to the effective definition. Each `?`/`!` section is ON or OFF (default from its suffix, overridden by the user's choice); a plain `@@@section` is always ON. An OFF section and all its subsections become *inert*: their body text, includes, and assertions take no further part. Selection happens here — before any binding — so inert material never triggers Phase 8 (includes) or Phase 10 (assertions), and substitutions confined to inert material never enter the Phase 9 required-parameter set.

### Phase 5 — Parameter binding

Constants are resolved and type-validated at the *start* of this phase (hoisted from Phase 6) because an include binding value may reference a caller constant. Each parameter's bound value is then com-

puted in its own prompt frame: the one selected parameter set (if any) is applied, then explicit instantiation values, then the declared default for any parameter still unbound. The values for an `@@@include B(name=value)` call site are also computed here, in the caller's frame — but not yet checked against B's interface (B is not resolved until Phase 8).

### Phase 6 — Constant resolution

Constant values are already resolved at the start of Phase 5 (above). This phase confirms that resolution and validates immutability constraints. A constant cannot be re-bound by a later declaration of the same name.

### Phase 7 — Attachment binding

`fixed` attachments are validated according to their source: the path must exist (`file`), the inline data URI must be well-formed (`blob`, a static check already made at Phase 1), or the ref must resolve through the store (`record`). Required `to-include` attachments must be provided at instantiation; their absence is ATTACHMENT_REQUIRED. Optional `to-include` attachments (`attachment?`) are bound if provided and silently omitted if not.

After Phase 7, the prompt's attachment slots are either filled or known to be empty (for the optional ones).

### Phase 8 — Include resolution

Every `@@@include path` (line form) and `@@@{include path}` (inflowing form) is resolved to a single target, then its interface is validated, then the target is rendered recursively and its text spliced in. Includes may reference the canonical id, an alias (already registered in Phase 1 and resolved here during segmentation), or — when the prompt is in a derived collection — the collection-axis form `^parent.<template_id>` resolved against the effective prompt set computed in Phase 2.

Cycles are detected (CIRCULAR_INCLUDE); the maximum include depth is implementation-defined but always finite. The binding values computed in Phase 5 are applied to the resolved target; a binding naming a parameter the target does not declare raises UNDECLARED_INCLUDE_PARAMETER.

### Phase 9 — Parameter substitution

Every `@@@=name` (and `@@@{= name }`) in the retained body is replaced by the bound value. The required-parameter set is computed and checked: a referenced non-optional parameter with no default that is unbound raises UNDEFINED_PARAMETER; a substitution whose symbol is neither parameter nor constant nor SYSTEM VARIABLE raises UNDEFINED_REFERENCE. Optional parameters with no binding resolve to their (mandatory) default.

Declarative constraints are validated here: `min`, `max`, `minlen`, `maxlen`, `pattern`, and the `select(...)` option list. A violation raises CONSTRAINT_VIOLATION or SELECT_INVALID_OPTION.

### Phase 10 — Assertion evaluation

Now that substitution, includes, and inheritance are fully resolved, the content of each scope (`@@@scope`, section, include, prompt) is final, and the assertions can be checked. First the scope nesting is validated: a `@@@scope` that straddles a section boundary — a new `@@@section` opening before the matching `@@@end-scope` — raises MALFORMED_SCOPE. Then each `@@@assert` predicate (`contains`, `not contains`, `matches`, `minlen`, `maxlen`) is evaluated against the *entire* rendered content of its nearest enclosing scope, independent of where the assert sits; multiple asserts in the same scope combine as an implicit AND. Evaluation is deterministic and does not alter the output. Any unsatisfied predicate raises ASSERTION_FAILED and blocks instantiation — no output is emitted. Assertions validate the rendered prompt only, never the model's response (Chapter XIII).

### Phase 11 — Output generation

Every directive is stripped — identity and all others alike: `@@@template`, `@@@name`, `@@@description`, `@@@template-body`, `@@@section`, `@@@collection` / `@@@end-collection`, `@@@param`, `@@@const`, `@@@attachment`, `@@@alias`, `@@@deprecated`, the `@@@parameter-set` block, and the `@@@scope` / `@@@end-scope` boundaries (which are transparent to output). The `virtual` / `final` modifiers vanish with the headers that carried them. A full-line `@@@include` is replaced in place by its target's rendered text. Comments disappear entirely; verbatim *content* remains but its delimiters are stripped.

The output is a `RenderedTemplate` whose `text` member is a plain string — exactly the text that will be passed to the language model — accompanied by the ordered list of bound attachments.

## 3.3 — Where errors and warnings appear

Each phase has its own catalogue of error and warning conditions. The following table summarizes the *first* phase at which each condition can be raised; later phases inherit the right to raise it too (an include introduced in Phase 8 may reference a DEPRECATED prompt, in which case the Phase 1 deprecation warning fires for the included prompt).

| Phase | Errors raised here | Warnings raised here |
|---|---|---|
| 1 — Parsing | `INVALID_SYNTAX`, `DUPLICATE_DECLARATION`, `ALIAS_CONFLICT`, `SYSTEM_VARIABLE_OVERRIDE`, `VIRTUAL_SECTION_HAS_BODY`, `VIRTUAL_TEMPLATE_HAS_BODY` | `DEPRECATED_TEMPLATE`, `MISSING_DESCRIPTION` |

| Phase | Errors raised here | Warnings raised here |
|---|---|---|
| 2 — Collection context | `CIRCULAR_COLLECTION_INHERITANCE`, `UNKNOWN_COLLECTION`, `UNIMPLEMENTED_VIRTUAL_TEMPLATE`, `OVERRIDE_OF_FINAL_TEMPLATE`, `UNDEFINED_REFERENCE` | — |
| 3 — Template inheritance | `CIRCULAR_INHERITANCE`, `OVERRIDE_OF_FINAL_SECTION`, `UNIMPLEMENTED_VIRTUAL_SECTION`, `UNDEFINED_REFERENCE` | — |
| 5 — Parameter binding | `TYPE_MISMATCH` (constants), `UNDEFINED_REFERENCE` | — |
| 6 — Constant resolution | `IMMUTABILITY_VIOLATION`, `UNDEFINED_REFERENCE` | — |
| 7 — Attachment binding | `ATTACHMENT_REQUIRED`, `ATTACHMENT_NOT_FOUND` | — |
| 8 — Include resolution | `CIRCULAR_INCLUDE`, `AMBIGUOUS_INCLUDE`, `UNDECLARED_INCLUDE_PARAMETER`, `INCLUDE_OF_VIRTUAL_SECTION`, `UNDEFINED_REFERENCE` | — |
| 9 — Parameter substitution | `UNDEFINED_PARAMETER`, `UNDEFINED_REFERENCE`, `CONSTRAINT_VIOLATION`, `SELECT_INVALID_OPTION` | `PARAMETER_NAME_IS_PREFIX` |

| Phase | Errors raised here | Warnings raised here |
|---|---|---|
| 10 — Assertion evaluation | `MALFORMED_SCOPE`, `ASSERTION_FAILED` | — |
| 11 — Output | — | — |

The complete catalogue, with cause and remediation for each condition, is §XXXI.

## 3.4 — Determinism in practice

The combination of fixed phase order, no control flow, and statically validated references means that POO²L processing is deterministic in a strong sense. Given the same prompt source, the same parent chain, the same collection chain, and the same parameter bindings, every conformant processor produces the same output, byte for byte (up to whitespace normalization, which is implementation-defined within narrow limits).

This is not the same kind of determinism that traditional programming languages give. A POO²L processor *cannot* loop or branch on the value of a parameter; it cannot generate code at runtime; it cannot evaluate arbitrary expressions. The phases listed above are the *only* things a processor does. The language model that receives the output may behave non-deterministically, but the *input* to the model is fully determined by the source.

That distinction is the engineering payoff. A change to a parent prompt ripples through the children deterministically; a constraint added to a parameter is enforced before any text reaches the model; a deprecated ancestor surfaces in the warning channel on every instantiation. None of this requires running the prompt to discover.

## 3.5 — Mapping phases to authoring decisions

The phase order also explains a number of POO²L's authoring rules that might otherwise look arbitrary :

- *Why must a prompt declare its id before its sections ?* Only the `@@@template` header — which carries the id — must come first ; it is the entity-opening header. The remaining declarations may appear in any order ( `§4.5` ). The recommended order exists for readability, not for the parser.

> ***`standard.md §4.5` — Standard Declaration order (excerpt) :***
>
> *POO²L generators SHOULD organize declarations in this order : (1) Identity (the `@@@template` header first, then `name` and the optional `description` ) ; (2) Lifecycle (alias, deprecated) ; (3) Parameters ; (4) Constants ; (5) Content ; (6) Attachments. Apart from the `@@@template` header, which MUST open the entity, POO²L processors MUST accept declarations in ANY order — the order of appearance has NO semantic significance for processing.*

- *Why are parameter sets forbidden inside a prompt that belongs to a collection ?* Because parameter binding (Phase 5) draws on the collection-level sets resolved in Phase 2 (collection context), which must be merged before any template-level set is considered. Allowing both opens a conflict the language has chosen to outlaw rather than resolve.
- *Why are aliases registered in Phase 1, before everything else ?* Because every later phase that references templates by id must work on canonical ids only. The effective rewrite of an alias to its canonical id happens during include resolution (Phase 8), once the registry is fully known, but the registration and local-conflict check happen up front in Phase 1.

- *Why do the two OO axes never collide on `parent`?* Because they are DISJOINT and named SYNTACTICALLY, not by precedence. A plain `parent` / `parent.<section>` (no `^`) is ALWAYS the prompt axis (OO #1), consumed during prompt inheritance; a `^parent.<template_id>` (with the unary `^`) is ALWAYS the collection axis (OO #2), resolved during include resolution against the effective prompt set from Phase 2. There is NO precedence and NO silent fallback: a `parent.<name>` that matches no section of the parent prompt is a clean UNDEFINED_REFERENCE — it never re-interprets itself as a collection member, and vice versa. `§4.6.1` states this as « TWO DISJOINT AXES — NO PRECEDENCE ».

> ***`standard.md §4.6.1` — Template Collections (excerpt):***
>
> *A prompt collection is the OO² horizontal-axis entity. A collection may inherit from another collection and a prompt rendered inside a derived collection may reference the parent collection's templates via `^parent.<template_id>` or `^parent.<template_id>.<section>`. The axis is fixed SYNTACTICALLY by the unary prefix `^`: `self` / `parent` (no `^`) are the TEMPLATE axis only; `^self` / `^parent` (with `^`) are the COLLECTION axis only. TWO DISJOINT AXES — NO PRECEDENCE, no silent fallback. A `parent.<name>` that matches no section of the parent prompt is an ERROR (UNDEFINED_REFERENCE); it never falls through to a collection member.*

The full mapping between authoring rules and phases lives in Reference §29; the present chapter is a reading map.

## 3.6 — When phases are observable to the author

Two phases produce visible side effects beyond the rendered output:

- **Phase 1** emits warnings for deprecated templates (and the advisory `MISSING_DESCRIPTION` for a structured prompt that omits `@@@description`). A renderer should surface those warnings somewhere the author can see them.
- **Phase 7** binds attachments. An author renders a prompt with the intent of including a `to-include` attachment, and the renderer's response indicates whether the attachment was provided and bound.

Every other phase is silent on success. An author who wants to *see* the intermediate state — the resolved inheritance chain, the effective collection set, the parameter values before substitution — needs tooling support; the language does not mandate one.

The next chapter turns to the language's vocabulary itself: a short cheatsheet of the terms that appear in every subsequent chapter.

# Chapter IV — Terms and Definitions Cheatsheet

The specification's `§3` defines every term that appears in the rest of the document. This chapter restates those definitions in the order in which the reader will encounter them, with cross-references to the chapter of this book where each one is treated at length. Refer here whenever a later chapter uses a term whose meaning has slipped from memory; consult the specification when a normative phrasing is required.

> ***`standard.md §3` — Terms And Definitions (intro):***
>
> *Section 3 of the specification lays down the controlled vocabulary of POO²L. Each term has a single normative definition, and every later clause uses these terms with the meaning fixed here.*

## 4.1 — Identity-related terms

**Template** — A structured or unstructured text input provided to a Large Language Model to elicit a desired response or behaviour. The atomic unit of POO²L. (Spec §3.1.)

> ***`standard.md §3.1` — Template:***
>
> *A structured or unstructured text input provided to a Large Language Model to elicit a desired response or behavior.*

**Identity** — The `id` (carried on the `@@@template` header) and the `@@@name` declaration that every structured POO²L prompt carries, plus an OPTIONAL `@@@description`. The `id` is the

prompt's stable handle ; the `name` is human-readable ; the `description` is short prose whose omission raises only the advisory `MISSING_DESCRIPTION` warning. → §VI.

**Alias** — An alternative identifier for a prompt, enabling backward compatibility when the canonical `id` changes. Aliases are resolved transparently in Phase 1 of the processing model. (Spec §3.13.) → §XXV.

> ***`standard.md §3.13` — Alias :***
>
> *An alternative identifier for a prompt, enabling backward compatibility when a prompt's primary identifier changes.*

**Deprecated** — A lifecycle marker on a prompt signalling that it is scheduled for removal and should be replaced. Purely informational ; does not prevent instantiation. (Spec §3.12.) → §XXVI.

> ***`standard.md §3.12` — Deprecated :***
>
> *A lifecycle marker on a prompt signaling that it is scheduled for removal and should be replaced by an indicated alternative.*

## 4.2 — Structural terms (single-template scope)

**Section** — A named, delimited component of a structured prompt serving a specific purpose (e.g., role, task, context, constraints). Sections are *optional* — a prompt may define zero, one, or many. (Spec §3.2.) → §VII.

> ***`standard.md §3.2` — Section :***

> *A named, delimited component of a structured prompt serving a specific purpose (e.g., role, task, context, constraints). Sections are OPTIONAL.*

**Comment** — Text removed at instantiation time. Three forms: line (`@@@-- ...`), bounded (`@@@(-- ... )`), and block (`@@@comment ... @@@end-comment`). All introduced by the `--` marker; the block form nests by depth. → §VIII.

**Verbatim block** — A region whose POO²L syntax is *not* processed; the content remains in the rendered output but the inner directives are not interpreted. Used for meta-templates and escaping. → §IX.

**System variable** — A name resolved by the processor at render time without an author-side declaration. The names cover date and time (`DATE`, `TIME`, `DATETIME`, `YEAR`, `MONTH`, `DAY`, `WEEKDAY`), identity (`USER` — the user's display name — and, host-provided, `USERNAME` — the login), locale (`LANG`), and group membership (`GROUPS`, `GROUPS_ADMIN`). The engine reserves eleven names natively; `USERNAME` (the login) is supplied by the host shell, not computed by the engine. → §X.

## 4.3 — Variable and binding terms

**Parameter** — A named variable within a prompt whose value is bound at instantiation time. Parameters may be required or optional (the `?` suffix on the keyword denotes optional). Types: `text`, `textarea`, `number`, `date`, `boolean`, `select`. (Spec §3.5.) → §XI.

> ***`standard.md §3.5` — Parameter:***
>
> *A named variable within a prompt that can be bound to a value at instantiation time. Parameters may be required or optional (denoted by `?` suffix).*

**Constraint** — A declarative validation rule attached to a parameter, defining bounds or patterns that the bound value must satisfy. Available constraints: `min`, `max`, `minlen`, `maxlen`, `pattern`, and the implicit option list of a `select` type. (Spec §3.11.) → §XII.

> ***`standard.md §3.11` — Constraint:***
>
> *A declarative validation rule attached to a parameter, defining bounds or patterns that values MUST satisfy (e.g., `min`, `max`, `minlen`, `maxlen`, `pattern`).*

**Constant** — A named immutable value bound at definition time. Shares the type and label syntax of parameters; the difference is that a constant cannot be re-bound at instantiation. (Spec §3.6.) → §XIV.

> ***`standard.md §3.6` — Constant:***
>
> *A named immutable value defined within a prompt. Once bound, constants cannot be reassigned.*

**Parameter set** — A named bundle of parameter bindings that can be applied to instantiate a prompt. Multiple parameter sets per prompt are allowed. A parameter set declared *inside* a prompt is forbidden if the prompt belongs to a collection — collection-scoped sets are managed at the collection level instead. (Spec §3.8.) → §XV.

> ***`standard.md §3.8` — Parameter Set:***
>
> *A collection of named parameter bindings that can be applied to instantiate a generic prompt.*

**Attachment** — File or binary content associated with a prompt, described by two orthogonal axes. WHEN it is bound: `fixed` (bound at definition) or `to-include` (provided at instantiation, required by default, optional with `?`). SOURCE (only for `fixed`; where the content lives): `file` (filesystem path), `blob` (inline data URI), or `record` (external store ref). A `to-include` attachment carries no source. The `attachment?` prefix denotes an optional `to-include` attachment. (Spec §3.7.) → §XVI.

> ***`standard.md §3.7` — Attachment:***
>
> *A file or binary content associated with a prompt, described by two orthogonal axes. WHEN it is bound: `fixed` (bound at definition time) or `to-include` (provided at instantiation time — required by default, optional with `?`). SOURCE (only for `fixed`; where the content lives): `file` (a filesystem path), `blob` (an inline data URI), or `record` (a reference to an external store). A `to-include` attachment carries NO source — the caller supplies the content at instantiation.*

## 4.4 — Composition terms (multi-template scope)

**Include** — A reference mechanism that incorporates external prompt content by reference. Supports modularity and reuse. Includes can pass parameter overrides on the include site. (Spec §3.3.) → §XVII, §XVIII.

> ***`standard.md §3.3` — Include:***
>
> *A reference mechanism enabling the incorporation of external prompt content by reference, supporting modularity and reusability.*

**Inheritance** — A mechanism whereby a prompt derives structure and content from a parent prompt, enabling specialization. POO²L supports single inheritance only ; multi-level chains are allowed ; circular inheritance is forbidden. (Spec §3.4.) → §XIX.

> ***`standard.md §3.4` — Inheritance :***
>
> *A mechanism whereby a prompt derives structure and content from a parent prompt, enabling specialization and variant creation.*

**`self` reference** — Inside a prompt that participates in an inheritance chain, `self.<section>` resolves the *most specialized* version of `<section>` in the chain — current prompt first, then walking up. `self.<prompt>.<section>` explicitly picks a named ancestor.

**`parent` reference** — On the TEMPLATE axis (OO #1), `parent` is the parent prompt's fully resolved content ; `parent.<section>` is the parent's named section, *ignoring* any same-name override the current prompt declares. Useful for *extending* rather than *replacing*. The COLLECTION axis (OO #2) uses the distinct `^`-prefixed form : at L5, from a prompt in a derived collection, `^parent.<template_id>` is the parent collection's prompt of that id. The two axes are disjoint — the `^` names the axis, never a precedence. → §XX.

**Template collection** — A named grouping of related templates, parameter sets, and sub-groupings. Comes in two kinds : a *folder* (pure organization, all levels) and a *collection* (the OO² entity opened by a `@@@collection` block, L5 only). (Spec §3.9.) → §XXI (folders), §XXII (collections), §XXIII (collection inheritance and virtual templates).

> ***`standard.md §3.9` — Template Collection :***

> *A named grouping of related templates, parameter sets, and sub-groupings. A collection is the OO² entity opened by a `@@@collection <name>` block, supporting inheritance (the inline `inherits` clause on its header), virtual templates, member override, and collection-axis `^parent.<template_id>` resolution — required for L5 conformance. Collections NEST by containment to unbounded depth; qualified member ids carry as many segments as the containment run requires. A folder is a pure organizational grouping available at all levels and carries none of the OO² machinery.*

**Folder** — A pure file-system-like organizational grouping, addressed with `/`. Cross-level. No inheritance, no virtuals, no `parent` resolution. → §XXI.

**Virtual prompt** — A collection member whose `@@@template` header carries the `virtual` modifier. The prompt's parameter list constitutes its abstract signature. A descendant collection MUST provide a non-virtual prompt with the same id (member override) for the chain to be instantiable on that id. Unimplemented virtuals raise the UNIMPLEMENTED_VIRTUAL_TEMPLATE error. → §XXIII.

**OO²** — Object-Oriented *squared*. The combination of the two parallel OO axes: template-level (sections as methods, `inherits` between templates — OO #1) and collection-level (templates as methods, `inherits` between collections — OO #2). The two axes compose orthogonally. (Spec §3.10.) → §XXIII, §XXIV.

> ***`standard.md §3.10` — OO²:***
>
> *Object-Oriented Squared — two parallel Object-Oriented mechanisms operating on different membership relations. OO #1 (templates): a prompt's methods are its sections; a prompt may carry an inline `inherits` clause on its `@@@template` header and reference `parent.<section>`. OO #2 (collections): a collection's methods are its templates; members whose*

> *`@@@template` header carries the `virtual` modifier are abstract and MUST be overridden by a concrete prompt with the same id in some descendant. The two axes compose orthogonally.*

## 4.5 — Process and conformance terms

**Resolution phase** — One of the eleven ordered phases by which a processor turns prompt source into rendered output. (Spec §6.1.) → §III (overview), §XXX (in depth).

> ***`standard.md §6.1` — Resolution Phases (excerpt):***
>
> *POO²L processors SHALL execute eleven phases in order: (1) Parsing, (2) Collection context application, (3) Template inheritance resolution, (4) Section selection, (5) Parameter binding, (6) Constant resolution, (7) Attachment binding, (8) Include resolution, (9) Parameter substitution, (10) Assertion evaluation, (11) Output generation. The order is normative and ensures deterministic rendering across conformant implementations.*

**Conformance level** — The set of language features an implementation claims to support. L0 support is mandatory but not a claimable badge; L1–L5 are the progressive claimable levels. L5 implies all the L4 features plus the second OO axis on collections. (Spec §8.1.) → §XXXII.

> ***`standard.md §8.1` — Conformance Levels (excerpt):***
>
> *A conforming implementation declares a single conformance level N (1 ≤ N ≤ 5). L0 (unstructured templates) is REQUIRED of all implementations but is NOT a claimable level. L1 (structured sections); L2 (sections + includes); L3 (structured data and validation — parameter sets, parameters/constants/attachments, select options, declarative constraints, deprecated markers, aliases, `@@@assert` / `@@@scope`); L4 (Object-Oriented*

> *— prompt inheritance, OO #1, with the full section-modifier toolbox); L5 (full OO² — the second OO axis on collections). Folders are cross-level and MUST be supported by all conforming implementations.*

**Sidecar metadata** — Application-level data stored *outside* prompt text (permissions, tags, and similar policy fields). A prompt's collection *membership* is not sidecar data: it is declared in the source by qualifying the `@@@template` header id (`@@@template <collection>.<id>`), and the collection itself is opened by a `@@@collection` block. → §XXII.

**Effective prompt set** — For a leaf collection in a chain, the deduplicated union of all member templates along the chain, keyed by prompt id, with member overrides applied. Computed in Phase 2. → §XXIII, §XXX.

## 4.6 — The two axes of `parent`

`parent` and `self` name the two OO axes, but they never share a surface and there is NO precedence between them. The axis is fixed SYNTACTICALLY by the unary prefix `^`:

1. **Template axis (OO #1) — no `^`.** `parent` / `parent.<section>` / `self.<section>` reach the parent (or most-specialized) *section* along the `inherits` chain. A `parent.<name>` matching no ancestor section is a clean UNDEFINED_REFERENCE — it never re-reads as a collection member.
2. **Collection axis (OO #2) — with `^`.** `^parent.<template_id>` / `^parent.<template_id>.<section>` / `^self.<template_id>` reach a *member* of the parent (or current) collection, available only when the current prompt is a member of a derived collection.

Because the axes are disjoint, adding or removing a same-named section (one axis) or member (the other) can never change which one a reference denotes. The parent collection's exact full prompt is directly `^parent.<template_id>` (no trailing section needed). → §XX.

## 4.7 — A short glossary of cross-cutting jargon

- **Render** — Execute the eleven phases on a prompt to produce its final text.
- **Instantiate** — Provide concrete values for a prompt's parameters, attachments, and (if applicable) parameter set, then render it.
- **Bind** — Associate a value with a parameter slot or a constant.
- **Substitute** — Replace a `@@@=parameter_name` marker in the rendered text by the bound value.
- **Override** — In OO #1, a child section with the same name as a parent section replaces the parent's content. In OO #2, a member prompt in a derived collection with the same id as a parent collection's member replaces the parent's content.
- **Virtual** — Declared but not implemented. Applies to templates at OO #2 (members whose `@@@template` header carries the `virtual` modifier) and to sections at OO #1 (`@@@section virtual`).
- **Concrete** — Not virtual. A concrete prompt has a body usable as-is.
- **Chain** — A linear sequence of `inherits` references walked root-first by the processor.

The next chapter is a short reading map: how the parts and chapters of this book are organized, in what order to read them, and what to skip.

# Chapter V — How to Read This Book

The four chapters so far have laid the groundwork: why the language exists, the six maturity levels, the eleven processing phases, and the vocabulary that runs through the rest of the volume. This chapter is the shortest in the book. Its only job is to tell you how the remaining chapters are organized so you can read them in the order that fits the question you have.

## 5.1 — The shape of the book

The book has six parts and a small appendix tail. Their roles are not interchangeable.

| Part | What it teaches | Read when |
|---|---|---|
| I — Concepts (Ch. 1–5) | The mental model. | You are new to POO²L or returning after time away. |
| II — Building blocks (Ch. 6–10) | Constructs that operate inside one prompt. | You have written your first prompt and need vocabulary. |
| III — Variables and binding (Ch. 11–15) | Parameters, constraints, constants, parameter sets, attachments. | You start parameterizing templates. |
| IV — Composition (Ch. 16–23) | Includes, inheritance, `parent` / `self`, folders, collections, the second OO axis. | Your library has more than one prompt. |

| Part | What it teaches | Read when |
|---|---|---|
| V — Lifecycle and governance (Ch. 24–28) | Aliases, deprecation, versioning, access control. | The library is shared across a team. |
| VI — Reference (Ch. 29–33) | Phase-by-phase details, error catalog, conformance, the closed-extension policy, implementer notes. | You are looking something up. |
| Appendices A–E | Grammar (EBNF), worked examples, anti-patterns, cookbook, crosswalk to the spec. | You want a recipe, a normative reference, or a refactoring move. |

Part I is the only part that benefits from being read end to end. Parts II, III, and IV may be read out of order *within* themselves once Part I is in place; many readers skip to the construct they are about to use and read its chapter as a self-contained unit. Part V is read once and returned to when the library outgrows the soloist stage. Part VI is for lookup, not for narration.

## 5.2 — The per-construct chapter template

Parts II, III, and IV are the *construct* parts. Every chapter in those three parts uses the same five-section template, in this order:

1. **Syntax** — the formal grammar fragment for the construct, citing the relevant `standard.md §X.Y`. This is the section to copy from when typing the construct for the first time.

2. **Purpose** — the problem the construct solves; what was painful or impossible without it. This is the section that connects the syntax to a reason for using it.

3. **Possibilities** — variants, options, edge cases; what *is* and what *is not* allowed. This is the section that catches the corner cases the syntax fragment doesn't show.

4. **Related features and why** — the other POO²L constructs that combine with this one and the design intent behind each combination. This is the section that prevents the common mistake of reaching for one construct when another would have done the job.

5. **Practical examples** — three or four examples, graduated : trivial, idiomatic, real-world, and (when illustrative) an anti-pattern. This is the section to read when the previous four did not quite click.

When you find a construct in the middle of someone else's prompt and want to understand what it does, skip to its chapter and read the Syntax and Purpose sections. When you are *writing* the construct yourself for the first time, read all five.

## 5.3 — When the chapter is not a construct

Parts I, V, and VI are *not* construct chapters. They use whatever structure serves their content. Their chapters are signposted in their opening paragraphs.

## 5.4 — Cross-references

References between chapters of this book use the form `→ §N`, generated from chapter slugs by bkgen's reference placeholder mechanism. References to the specification itself appear as `standard.md §X.Y` and always point to a numbered section, never to a paraphrase. The reader who wants the normative phrasing of a clause should follow the `standard.md §X.Y` reference ; the reader who wants the pedagogical treatment should stay in this book.

If a chapter says « the spec requires X » without a `standard.md` reference, that is an editorial bug — please report it through the errata channel listed in the front matter.

## 5.5 — A reading map for three audiences

The same book serves three quite different readers. Each one has a shortest useful path through it.

**The new author** — someone writing their first parameterized prompt :

1. Chapter I — Why POO²L Exists.
2. Chapter II — The Six Maturity Levels.
3. Chapter VI — Identity.
4. Chapter VII — Sections.
5. Chapter XI — Parameters.
6. Chapter XIV — Constants.
7. Chapter XVII — Includes.

The new author can defer Parts IV, V, and VI until the day a specific need arises.

**The author who already knows software-engineering OO** — someone familiar with classes, inheritance, virtual methods, and the substitution principle :

1. Chapter I — Why POO²L Exists (skim).
2. Chapter II — The Six Maturity Levels (read carefully — the table at the top is the key).
3. Chapter XIX — Inheritance.
4. Chapter XX — `parent` and `self`.
5. Chapter XXII — Collections.
6. Chapter XXIII — Collection inheritance and virtual templates.

The OO-fluent reader will recognize most concepts under different syntactic clothes ; the unique POO²L material is concentrated in the L4 chapters and in the absence of features (Chapter XXXIII — the closed- extension policy).

**The implementer** — someone building a POO²L processor, linter, or editor plug-in :

1. Chapter III — The Processing Model in Eleven Phases.
2. Chapter XXX — Resolution phases in depth.
3. Chapter XXXI — Error catalog.
4. Chapter XXXII — Conformance levels.
5. Chapter XXXIV — Implementer notes.
6. Appendix E (crosswalk to the spec) — the EBNF is the specification's own Appendix B (`standard-syntax.md`), not reproduced in this book (single source of truth).

For the implementer, the book is a *companion* to the specification, not a substitute. Every normative statement still lives in `standard.md`.

## 5.6 — What this book deliberately leaves out

A few topics are intentionally not covered here :

- *The POO²L specification document itself.* `POO²L/standard.md` is authoritative and freely available. This book reorganizes its content for a reader ; it does not re-spec it.
- *The IziBuddy user interface.* The dashboard's panels, dialogs, and shortcuts are an authoring tool's concern, not the language's. When this book mentions IziBuddy, it does so in its role as *reference implementation*, not as an authoring tool.
- *Hands-on tutorials.* The book has examples but no exercises. The step-by-step « do this on your own library » treatment is the job of a separate learning path, not of this reference.
- *Operational topics* (deployment, Git sync, backups). Those belong to a tool's operations documentation, not to the language reference.

- *Future versions of POO²L.* POO²L v1.0 is closed by design (Chapter 32). A v2 would be a new specification with its own version number. When that happens, this book will be revised ; until then, this volume is stable.

## 5.7 — A note on the syntax of the language inside this book

Throughout this book, POO²L source appears inside fenced code blocks with no language tag, exactly as the specification renders it. The choice is deliberate : POO²L is not a programming language in the sense that triggers syntax-highlighting libraries, and a generic plain formatting keeps the focus on what the prompt *says* rather than what its keywords *look like*. Editor tooling for POO²L, when it exists, typically renders the delimiters in a muted colour and the keywords in a slightly more prominent one ; the book makes no such distinction.

Identifiers inside POO²L examples are italicized when introduced for the first time in prose ; subsequent references use the bare identifier. Spec keywords ( `section` , `param` , `inherits` , …) appear inside backticks. Section titles in tables match the title of the chapter they refer to.

With the reading map in place, we move on to the language itself. The next chapter — and the first of the construct chapters — is *Identity*.

# Partie 2

*Building blocks (single-template scope)*

# Chapter VI — Identity

> **`standard.md §5.2` — Identity Declarations (excerpt):**
>
> *`@@@template <id> [inherits <parent>] (id: unique, no spaces; optional parent prompt) @@@name <template_name> (human-readable name, spaces allowed) @@@description <short_desc> (OPTIONAL; brief description, spaces allowed)` `@@@template` and `@@@name` are MANDATORY for a structured (L1+) prompt; `@@@description` is OPTIONAL. A structured prompt that OMITS `@@@description` renders normally; the omission is the `MISSING_DESCRIPTION` WARNING condition (§6.2) — advisory only, never blocking. The `id` carries no spaces, is used for includes and references, MAY be qualified (`<collection_name>.<template_id>`) to declare collection membership, and MAY carry an inline `inherits <reference>` clause naming the parent prompt.*
>
> **`standard.md §4.4.1` — Template Identity (rationale):**
>
> *Identity metadata ensures templates are not anonymous when exported, migration preserves prompt identity, collections can reference templates by `id`, and includes use the `id` for resolution.*

Every POO²L prompt carries an identity: a stable `id`, a human-readable name, and — optionally — a short description. The `id` and the `name` are mandatory; the description is recommended but not required. The identity is also the first thing the language asks the author to commit to — every later construct, from includes to inheritance to collection membership, refers back to it.

The `id` is not a separate `@@@id` directive: it rides on the `@@@template` **header** that opens the entity. The header does double duty — it both declares the prompt exists and names it.

## 6.1 — Syntax

The identity declarations follow the standard delimiter form. The `@@@template` header opens the entity and MUST come first; `@@@name` and the optional `@@@description` follow.

```
@@@template <id> [inherits <parent>]
@@@name <template_name>
@@@description <short_desc>
```

The specification's `§5.2` is explicit on the rules:

- `@@@template <id>` — the prompt header. The `id` is unique within scope; **no spaces**; used for includes, inheritance, and collection references; **case- and diacritic-insensitive** (canonical form is UPPERCASE with diacritics stripped). The header MAY be **qualified** (`<collection_name>.<template_id>`) to declare collection membership, and MAY carry an inline `inherits <reference>` clause naming the parent prompt (OO #1).
- `@@@name` — human-readable; spaces and special characters allowed; MANDATORY.
- `@@@description` — brief prose; spaces allowed; **OPTIONAL**. Its absence raises the `MISSING_DESCRIPTION` warning (`§6.2`).

The `@@@template` header is a STRUCTURAL EXCEPTION to free declaration order (`§4.5`): unlike `@@@name`, `@@@param`, `@@@const`, etc. — which a processor accepts in any order — `@@@template` is an entity-opening header and MUST appear first in its entity. A `@@@template` line anywhere but first is INVALID_SYNTAX. The remaining declarations may be reordered, but the recommended writing order places identity at the very top, before any lifecycle markers (`alias`, `deprecated`), parameters, constants, content, and attachments.

> ***`standard.md §4.5` — Recommended writing order (excerpt) :***
>
> *POO²L generators SHOULD organize declarations in the following order : (1) Identity ( `@@@template` header first, then `name` and the optional `description` ) ; (2) Lifecycle ( `alias` , `deprecated` ) ; (3) Parameters ; (4) Constants ; (5) [Content] ; (6) Attachments (at the end). Apart from the `@@@template` header, which MUST open the entity, processors MUST accept declarations in ANY order — the order of appearance has NO semantic significance for processing.*

## 6.2 — Purpose

Identity is the first answer POO²L gives to the four pathologies of Chapter I. The `id` solves *naming things* once and for all : every later reference uses it. The `name` makes the prompt findable by a human. The `description` saves the author from re-reading the body every time they encounter the prompt in a list.

Without identity, a prompt is anonymous. Anonymous things cannot be referenced ; cannot be inherited from ; cannot be included ; cannot be deprecated ; cannot be aliased. The whole compositional half of the language depends on every prompt being addressable by a stable token, and that token is the `id` on the `@@@template` header.

The split between `id` and `name` is deliberate. The `id` is for machines : it is the key in lookup tables, in include resolution, in collection membership. The `name` is for people : it is what shows up in the dashboard, in a generated table of contents, in an audit log. The two roles do not compete because they are not the same field.

## 6.3 — Possibilities

**`@@@template` and `@@@name` are mandatory; `@@@description` is optional.** A structured prompt without a `@@@template` header or a `@@@name` is not a conformant POO²L prompt (`§5.2`). The `@@@description`, by contrast, may be omitted: the prompt renders normally, and the omission raises the `MISSING_DESCRIPTION` warning (`§6.2`). Warnings are advisory — a processor MAY ignore them and still conform, surface them, or treat them as blocking in a stricter mode. Implementations are free to provide authoring defaults (for instance, deriving an `id` from a slugified `name`), but the rendered source must contain the mandatory declarations.

**No spaces in `id`.** The `id` starts with a Unicode letter and continues with Unicode letters, digits, and underscore only — NO hyphen (accented names such as `Problème` are valid; the grammar in Appendix B is the normative reference). The hyphen is reserved for the language's own multi-word keywords (`end-section`, `parameter-set`, ...), never for user names. Identifiers are **case- AND diacritic-insensitive**: they fold by Unicode case-folding with combining marks removed, canonicalising to UPPERCASE with diacritics stripped. `code_review` and `Code_Review` name the SAME prompt (as do `café` and `cafe`); two ids differing only by case or by diacritics are one id, and a redeclaration is a `DUPLICATE_DECLARATION`. String literals and rendered content are NOT folded — only identifiers.

**Reserved keywords cannot be ids.** A RESERVED KEYWORD — whether a line-directive keyword or a contextual keyword (`§5.1`) — MUST NOT be used as ANY user-defined identifier: prompt ids, collection and library names, folder segments, parameter / constant / attachment / section names, parameter-set names, and aliases. The list is the union of the language's keywords: `prompt`, `end-template`, `name`, `description`, `section`, `end-section`, `template-body`, `include`, `inherits`, `requires`, `uses`, `as`, `param`, `const`, `attachment`, `parameter-set`, `end-parameter-set`, `collection`, `end-collection`, `library`, `end-library`, `depreca-`

`ted`, `alias`, `verbatim`, `end-verbatim`, `comment`, `end-comment`, `assert`, `scope`, `end-scope`, the section-placement clauses `first` / `last` / `before` / `after`, the attachment keywords `fixed` / `to-include` / `file` / `blob` / `record`, and the modifiers `virtual` / `final`. Using one as an identifier is INVALID_SYNTAX.

> ***`standard.md §5.1` — Reserved keywords (excerpt):***
>
> *A reserved keyword MUST NOT be used as ANY user-defined identifier. This covers, without exception: parameter, constant, attachment and section names; prompt ids; collection names; library names; folder segments; parameter-set names; and aliases. Using a reserved keyword as an identifier is an ERROR (INVALID_SYNTAX).*

**Identity uniqueness is scoped, not global.** The `id` must be unique within the scope in which the prompt is referenced. A team that organizes templates by groups, projects, or directories may have two templates called `summary` in different scopes; the resolution rule — which scope wins when the same `id` is referenced — belongs to the referencing site (Chapter XVII, *Includes*), not to the identity declaration. Implementations enforce scope-based uniqueness at save time.

**Description has no length cap in the language.** The spec does not mandate a maximum length. Implementations may impose UI-level caps for ergonomic reasons; the rendered output is unaffected by description length because the description is not part of the rendered body.

**Renaming.** The `id` is the primary identifier; renaming it breaks every reference unless aliases (Chapter XXV) carry the old name forward. The `name` and `description` can be edited freely; they have no referential role.

## 6.4 — Related features and why

**Alias (Chapter XXV).** When an `id` needs to change — typically because a typo was committed years ago, or because a v1 prompt is being re-versioned as v2 — the old `id` is preserved as an alias. References to the alias resolve transparently to the canonical `id` during static analysis. Without aliases, every rename would be a breaking change.

**Deprecated (Chapter XXVI).** Identity is the *what*; deprecation is the *when to stop using it*. A `@@@deprecated "..."` marker says that the prompt of this identity is on its way out; the migration message points to the new identity that takes over. Deprecation does not change the identity; it annotates it. (Referencing a deprecated prompt raises the advisory `DEPRECATED_TEMPLATE` warning.)

**Inheritance (Chapter XIX).** A prompt inherits via the inline `inherits <parent>` clause ON its `@@@template` header — `@@@template child inherits parent` — not a separate directive. The `parent` reference resolves against the same identity space the prompt declares its own `id` in. An inheritance reference to an undefined `id` raises UNDEFINED_REFERENCE; inheritance is resolved at PHASE 3 of the processing model.

**Includes (Chapters XVII–XVIII).** Identical to inheritance: includes reference templates by `id`. The same scope rules apply; external references are resolved at PHASE 8.

**Collections (Chapters XXI–XXIII).** A prompt joins a collection by QUALIFYING its header id — `@@@template <collection_name>.<template_id>` — not via a separate membership directive. Membership is therefore part of how the prompt names itself, while the bare terminal segment (`<template_id>`) remains the key the collection's effective prompt set addresses. Moving a prompt between collections is re-qualifying its id.

**Virtual / final templates (Chapter XXIII).** `virtual` and `final` are MODIFIERS written on the `@@@template` header (`@@@template virtual <collection_name>.<template_id>`), immediately after the keyword — never directives of their own (a bare `@@@virtual` is INVALID_SYNTAX). A `virtual` member is abstract: a descendant collection MUST provide a non-virtual override under the same id. A `final` member is sealed: descendants MUST NOT override it. The modifiers are independent of the identity itself; the `id` is what makes the override match the abstract slot.

**Versioning (Chapter XXVII).** POO²L's versioning policy lives at the identity level: `id` is stable across versions, and a v2 prompt that relates to a v1 prompt typically does so via *alias* (v1 becomes an alias of v2) plus *deprecation* (v1's id is deprecated with a pointer to v2). The identity itself does not carry a version number; the versioning lives outside the prompt body.

## 6.5 — Practical examples

**Trivial — a one-liner prompt with full identity.**

```
@@@template code_review_assistant
@@@name Code Review Assistant
@@@description Reviews code for best practices and potential issues

You are an expert code reviewer. Review the attached code.
```

The header opens the entity, the name and description follow, and the body is L0 prose. The identity declarations are stripped during final assembly (PHASE 11) — they never appear in the rendered output.

**Optional description — valid, but warned.**

```
@@@template quick_note
@@@name Quick Note
```

```
Jot a one-line note about the current task.
```

This prompt is conformant: `@@@template` and `@@@name` are present. Because `@@@description` is omitted, a warning-surfacing processor reports `MISSING_DESCRIPTION` (`§6.2`); the render itself is unaffected and never blocks.

**Idiomatic — identity on a parameterized prompt.**

```
@@@template email_draft
@@@name Email Draft
@@@description Drafts an email in the requested tone

@@@param recipient : text, "Recipient name"
@@@param tone : select("formal", "casual") = "formal", "Tone"

Dear @@@=recipient,

[Draft the body in a @@@=tone tone.]
```

The `@@@template` header comes first, then the rest of the identity, then parameters, then content. Apart from the header, the processor would accept these declarations in any order (`§4.5`).

**Real-world — a collection member that is also a child prompt.**

```
@@@template reviewers.python_code_reviewer inherits code_reviewer
@@@name Python Code Reviewer
@@@description Specialization of code_reviewer for Python source

@@@section task
Review the Python code with emphasis on type hints,
PEP-8 conformance, and `dataclass` usage.
```

A single `@@@template` header carries all three relationships at once: the qualified id `reviewers.python_code_reviewer` declares **membership** in the `reviewers` collection, the terminal segment `python_code_reviewer` is the prompt's **id**, and the `inherits code_reviewer` clause declares **inheritance**. The name and description describe the prompt regardless of those relationships.

**Anti-pattern — duplicating intent across name and description.**

```
@@@template code_reviewer
@@@name Code Reviewer (review code for best practices)
@@@description Code Reviewer — review code for best practices
```

The `name` should be a *handle*; the `description` should be a *sentence*. Repeating the same content across both wastes the description's role of saying *what is this for* when the name already told the reader *what is this called*. A better split:

```
@@@template code_reviewer
@@@name Code Reviewer
@@@description Reviews code for correctness, performance, and
   adherence to language-specific best practices.
```

The next chapter walks the next single-template construct: *sections*.

# Chapter VII — Sections

> ***`standard.md §5.3` — Section Syntax (excerpt):***
>
> *A section declaration combines up to three independent axes: the **section keyword** with an optional **presence toggle** (`?` or `!`, glued to the keyword), an optional **inheritance modifier** (`virtual` or `final`) following the keyword, the section **name** — which MAY be a dotted `<parent>.<child>` name declaring a **sub-section** — and an optional **description** introduced by a colon `: <description>`. A plain `@@@section <name>` is always emitted in the rendered output — no dialog. The presence toggle turns the section into a user choice at instantiation time: `?` defaults the toggle to OFF (opt-in), `!` defaults it to ON (opt-out); the renderer shows the description in the include/exclude dialog. `?` and `!` are mutually exclusive and both REQUIRE a description, because the description is the dialog text. The inheritance modifier `virtual` marks an abstract section (no body; concrete descendant must implement); `final` marks a sealed section (descendants MUST NOT redeclare). `virtual` and `final` are mutually exclusive — combining them is INVALID_SYNTAX. The toggle is independent of the modifier and may combine freely. There is NO section numbering `<N>` in POO²L; ordered enumerations are expressed as dotted sub-sections (`steps.one`, `steps.two`). Rendering CARRIES the declared structure into the output as markdown ATX headings — `##` for a section, `### <n>.` for a sub-section. The prompt itself contributes no heading: its `@@@name` is a label and its id an address, neither being content. Levels 1 to 3 are RESERVED to the processor inside a declared section: a body-text line opening one is INVALID_SYNTAX.*
>
> ***`standard.md §4.2` — Level 1: Structured (excerpt):***

> *Templates are organized into named sections serving distinct purposes. Sections are OPTIONAL — a prompt may define zero, one, or multiple sections; there are NO mandatory sections. Common section names (`role`, `context`, `task`, `constraints`, `examples`, `format`) are examples, not normative.*

A section is a named, delimited component of a structured prompt. Sections are the *first* compositional unit POO²L offers — the move from L0 to L1 is, syntactically, the move from « no sections » to « any sections ». From that single move, includes, inheritance, `parent` / `self` references, and the entire OO half of the language become possible.

## 7.1 — Syntax

A section declaration is built from up to four independent axes. Each axis is optional and they may be combined freely, with one exception : `virtual`, `final` and `delete` are mutually exclusive.

1. **Inheritance modifier** (optional) : `virtual`, `final` or `delete`, placed after the `section` keyword (and after the presence toggle, if any), before the name. The slot holds a SET, not a single value — several modifiers may sit there, and repeating one is `INVALID_SYNTAX`.
2. **Presence toggle** (optional) : `?` or `!`, glued directly to the `section` keyword (never to the name).
3. **Transparency** (optional) : parentheses around the NAME — `(<name>)`. The SAME mark may be asked for at the other end, by the consumer : `@@@include (<reference>)` renders the target with no heading of its own, without touching the target's document. The transparency of a render is the LOGICAL OR of the two, so asking twice is asking once (see Includes).
4. **Sub-section** (optional) : the name MAY be a dotted `<parent>.<child>`, declaring `<child>` as a sub-section of an already-declared `<parent>`.

The general shape is:

```
@@@section[?|!] [virtual|final|delete] [(]<name>[)] [<placement>] [: <de
```

where `<name>` is a bare identifier or a dotted `<parent>.<child>` name.

**The name is an IDENTIFIER, and it is checked**: a letter, then letters, digits and `_` — never a hyphen, never a space, never empty. A stray token after the name that is not a placement clause raises `INVALID_SYNTAX` rather than being silently dropped (`@@@section a b` used to lose `b` without a word). A sub-section is one level only, and a segment made entirely of digits is tolerated on the CHILD alone, for the legacy index idiom (`steps.1`) — to be renamed to a meaningful name.

The always-rendered base forms (no inheritance modifier, no toggle):

```
@@@section <name>                                    (always rendered)
@@@section <name> : <description>                    (with description)
@@@section <parent>.<child>                          (sub-section)
```

The user-toggled forms (the `?` / `!` is glued to `section`):

```
@@@section? <name> : <description>                   (user-toggled, defa
@@@section! <name> : <description>                   (user-toggled, defa
@@@section? <parent>.<child> : <description>         (toggled sub-sectio
```

The transparent forms (the section emits NO heading of its own):

```
@@@section (<name>)
@@@section (<name>) : <description>
@@@section? (<name>) : <description>
@@@section virtual (<name>)
```

The name stays MANDATORY : a transparent section is a section in every other respect — presence toggle, placement clause, inheritance, override — only its heading is withheld. **Transparency propagates to what the section GOVERNS** : under a transparent container a sub-section emits neither ATX marker nor its title, only its RANK ( `1.` , `2.` ), which marks the sequence. The title is not lost — it lives in the declaration and reappears the moment a descendant NAMES the container, redeclaring it without the parentheses, at which point the container emits `## <Title>` and its sub-sections their `### <n>. <Title>` . That pairing is the sharing idiom : a parent ships a titleless bundle, each child titles it. The IMPLICIT section governs the same way — it emits no heading either, so what it receives emits none.

The deletion form (an INHERITED section renders nothing at all) :

```
@@@section delete <name>
```

It is valid ONLY on a section the prompt INHERITS — removing one’s own is simply not writing it ( `DELETE_OF_UNINHERITED_SECTION` ). It carries no body ( `DELETED_SECTION_HAS_BODY` ), is exclusive with `virtual` , is forbidden on a `final` section, and on a container it carries the whole subtree. It renders nothing — heading included — yet the section stays ADDRESSABLE ( `parent.<name>` still reaches the covered body) and ANCHORABLE ( `after <name>` still designates its place). That is how content removed from its inherited position gets re-placed elsewhere. A descendant may resurrect it by redeclaring it normally.

**An empty section is an error** ( `EMPTY_SECTION` ). Emptying an override was the de-facto way to suppress inherited content : the parent’s body vanished in silence and an orphan heading stayed. `delete` says it out loud, so the mute form has no reason to exist. Four exemptions, each already a rule of the language rather than a dispensation : a container (its content is its sub-sections), a

`virtual` section (abstract), a `delete` section (renders nothing), and every section of a `virtual` prompt. The rule is judged on the EFFECTIVE shape, because containment can be inherited — a child that NAMES an inherited transparent container writes no body of its own, and its sub-sections come from the parent.

The virtual forms (abstract section — descendant MUST implement) :

```
@@@section virtual <name>
@@@section? virtual <name> : <description>
@@@section! virtual <name> : <description>
@@@section virtual <parent>.<child>
```

The final forms (sealed section — descendant MUST NOT override) :

```
@@@section final <name>
@@@section? final <name> : <description>
@@@section! final <name> : <description>
@@@section final <parent>.<child> : <description>
```

A section's *content* is everything that follows the opening delimiter until the next section opening, or until the end of the prompt body. There is no explicit closing delimiter for sections.

Rules from `§5.3` :

- `name` — starts with a Unicode letter, then letters, digits, underscore only (NO hyphen) ; **case- and diacritic-insensitive** ( `Intro` , `intro` , `INTRO` are one section ; `résumé` and `resume` are one section).
- `<parent>.<child>` — a dotted name declares a **sub-section** ; `<parent>` MUST already be declared. Depth is limited to ONE level ( `<a>.<b>.<c>` is INVALID_SYNTAX). See *Sub-sections* below.

- A plain `@@@section <name>` is always emitted in the rendered output. The section is in the source ; the renderer has no say in keeping it. No dialog is shown.
- Both `?` and `!` turn the section into a **user choice at instantiation time** : the renderer presents the user with the section's description and lets them decide whether to include the section in the rendered output. They differ only by the **default value of the toggle** : `?` is pre-set to OFF (the user must opt *in*), `!` is pre-set to ON (the user must opt *out*). `?` and `!` are mutually exclusive.
- For `?` and `!` sections, the description is therefore **mandatory** — the description is the text the user reads in the include/exclude dialog. `@@@section? examples` without a description is TOG-
  GLABLE_SECTION_WITHOUT_DESCRIPTION — an empty `:` counts as absent, and an override that redeclares `?` without one erases the inherited description rather than inheriting it.
- Descriptions are preceded by a colon ( `: <description>` ) and are *not* enclosed in quotes — contrast with parameter and constant labels which use the comma + quoted form.
- `virtual` declares the section *abstract* : it carries no body (the source between the marker and the next section MUST be empty modulo whitespace and inline comments ; non-empty content raises VIRTUAL_SECTION_HAS_BODY). A concrete descendant MUST implement the section ; otherwise UNIMPLEMENTED_VIRTUAL_SECTION fires at render time.
- `final` declares the section *sealed* : no descendant may redeclare it under any form (no override of content, no change of toggle, no change of description). Any redeclaration is OVERRIDE_OF_FINAL_SECTION.
- `virtual` and `final` are mutually exclusive — combining them is INVALID_SYNTAX.
- Scoring and other application-level concerns are OUT of the language ; `section!` is **not** a scoring directive — it is purely a presence toggle with a default-ON value.

## 7.2 — Purpose

A piece of prose without sections is opaque. A reader must scan the whole body to find the part they want ; a child prompt that wants to override « just the constraints » has nothing to address. Sections give the prompt body a *vocabulary of named parts*, and that vocabulary is the foundation of every later compositional construct.

The specification names six common sections — `role` , `context` , `task` , `constraints` , `examples` , `format` — explicitly as *examples, not normative*. POO²L mandates no section names. The author chooses the vocabulary that fits the prompt ; the language only requires that once a name is chosen, it is used consistently across the templates that need to reference it.

Ordered enumerations use dotted **sub-sections** ( `steps.one` , `steps.two` , …) rather than any numbering syntax. Each sub-section has its own dotted name for resolution purposes ; `parent.steps.one` is not the same reference as `parent.steps.two` . A legacy trailing index is preserved as a name segment ( `steps 1` → `steps.1` ) and is best renamed to a meaningful segment.

The two inheritance modifiers — `virtual` and `final` — extend sections with the *abstract* and *sealed* contracts familiar from class-based OO. A `virtual section` says « I declare this slot exists and is mandatory ; some descendant must fill it. » A `final section` says « I declare this slot, I fill it, and no descendant may change it. » Together with plain (overridable) sections they cover the three contract shapes a section may take.

## 7.3 — Possibilities

**Zero, one, or many sections.** A prompt may declare any number of sections — including zero, in which case it is L0 unstructured prose. The specification phrases this as « Section Optionality : Sections are OPTIONAL » ( `§4.2` ).

**No mandatory section names.** The six conventional names (`role`, `context`, etc.) are not required, not reserved, and not enforced. An author may use any name allowed by the identifier rules.

**Independent axes.** Inheritance modifier and presence toggle act independently, and either may apply to a bare name or a dotted sub-section name. A section may carry any combination: `@@@section foo` (neither), `@@@section! foo : ...` (toggle only), `@@@section? virtual steps.one : ...` (toggle + modifier on a sub-section), and so on. The only forbidden combination is `virtual` + `final` on the same declaration.

**Presence toggle changes the user dialog default.** A section declared without `?` is part of the prompt's content unconditionally. A section declared with `?` opens a user dialog (default OFF); a section declared with `!` opens the same dialog (default ON). Both require a description because the description is what a user sees in the dialog.

**Order matters for output.** Sections render in the order they are declared. PHASE 3 (inheritance resolution) merges parent sections in the parent's declaration order, then child overrides replace by name without reordering, and new child sections are appended.

**Section content is plain text or other POO²L constructs.** A section may contain parameter references (`@@@=name`), includes, comments, verbatim blocks, and so on. A section MAY have **sub-sections** one level deep (`@@@section steps.one`); a section that has sub-sections carries **no body of its own** (its content is exactly its sub-sections in order) — a body directly under a container raises SECTION_WITH_SUBSECTIONS_HAS_BODY. A `virtual` section may carry no body at all.

**Section names are case- and diacritic-insensitive.** `@@@section Task` and `@@@section task` are the SAME section (a redeclaration is `DUPLICATE_DECLARATION`), and so are `@@@section résumé`

and `@@@section resume`. lowercase_with_underscores is the dominant convention in the IziBuddy reference library, but casing is a spelling choice, not an identity distinction.

**Sub-sections.** Ordered or structured enumerations use a dotted name (`@@@section steps.one`, `@@@section steps.two`) whose `<parent>` (`steps`) is declared first. A container section holds no own body; its rendering is its sub-sections' content, in declaration order. Depth is limited to one level — `<a>.<b>.<c>` is INVALID_SYNTAX. Turning a leaf into a container (or vice versa) across an inheritance chain raises SECTION_SHAPE_MISMATCH. `self.steps.one` and `parent.steps.two` address individual sub-sections.

**Description on a required section is allowed.** Required sections (no `?` and no `!`) *may* carry a description; only the toggled flavours require one. The description is informational and surfaces in tooling but is stripped from the rendered output.

## 7.4 — Override rules

Override of a section by a descendant is governed by the axes, the integrality rule, the toggle-monotonicity rule, and the most-derived-declaration rule.

**Each declaration is read integrally.** When a descendant prompt redeclares a section that already exists in the inheritance chain, the descendant's declaration replaces the ancestor's *as a whole*: inheritance modifier (`virtual` / `final` / none), presence toggle (`?` / `!` / none), number, and description all come from the descendant. There is no partial inheritance of any one field. To keep an ancestor's description while changing the body, the descendant must repeat the description.

**Toggle changes are monotonic, not free.** A descendant may ADD a toggle to a section that had none, or RELAX an opt-out `!` into an opt-in `?`, but it must not tighten the contract the other way (turning `?` into `!`, nor making a defaulted-OFF section mandato-

ry). A disallowed change raises OVERRIDE_CONTRACT_VIOLATION. This keeps a caller's valid instantiation of the parent valid against the child.

**Most-derived declaration wins.** At render time the processor walks the chain root-first, collects every declaration of each section, and uses the *most derived* declaration — virtual or not — to determine the effective modifier, toggle, description, and body. If the most-derived declaration is still `virtual` (no concrete override anywhere below it), rendering raises UNIMPLEMENTED_VIRTUAL_SECTION.

**`final` is sealed.** A `final` section forbids any further redeclaration in any descendant. The override attempt raises OVERRIDE_OF_FINAL_SECTION at PHASE 3, even if the descendant only changes the toggle or the description and leaves the body identical. A `virtual` section may be re-declared `virtual` (re-escalation) or turned into a concrete `final` (the override fixes the content and forbids further changes).

**The BOUNDED include pulls only the body.** A `@@@{include parent.<section>}` or `@@@{include <prompt>.<section>}` retrieves the *content* of the referenced section as resolved at the declaration site ; the inheritance modifier, presence toggle, and description stay at the including site's declaration. That is a property of the BOUNDED form, which splices a PHRASE at the marker : only the full-LINE form carries structure (6.1 11.a), and a block-level heading in mid-sentence would break the very line the bounded form exists to keep.

**The LINE form carries the heading.** `@@@include <prompt>.<section>` on a line of its own emits that section's heading at the point of splice — the title is information the source states, and dropping it loses it. One exception, and it is the « extend what I override » idiom : `@@@include parent.<section>` written INSIDE the body of `<section>` splices the body it SHADOWS into the section already being rendered, with no heading of its own, because

the heading is already there. The discriminator is the ENCLOSING section, not who wrote the reference — an inherited body carries its meaning with it.

An include that resolves to a `virtual` section (no body to substitute) raises INCLUDE_OF_VIRTUAL_SECTION at PHASE 8.

**Polymorphism without static/dynamic ambiguity.** POO²L has polymorphism — a section reference resolves to the most specialized implementation along the chain — but the chain walk is deterministic. There is no C++-style static-vs-dynamic distinction to choose between ; one resolution rule covers every site.

## 7.5 — Placement and visibility

**Placement clauses (on a child's ADDED sections).** By default a child's new sections are appended after the inherited skeleton. A child MAY instead position an added section relative to the inherited sections with a placement clause after the name :

```
@@@section <name> first            (before all inherited sections)
@@@section <name> last             (after all inherited sections)
@@@section <name> before <anchor>  (just before inherited section <ancho
@@@section <name> after <anchor>   (just after inherited section <anchor
```

The clause is valid ONLY on a section an inheriting prompt ADDS ; on an override or in a prompt with no `inherits` parent it raises INVALID_SECTION_PLACEMENT. `<anchor>` MUST be an INHERITED section — an anchor that is not raises INVALID_SECTION_ANCHOR. The clause composes with `virtual` / `final` and `?` / `!` and precedes the optional `: description`.

**Public sections (`@@@template-body`).** A structured prompt MAY split into a HEADER and a BODY with a single `@@@template-body` marker. A `@@@section` listed (as a bare name) BEFORE the marker is a PUBLIC listing ; a `@@@section` (with content) AFTER it is the DEFINITION. A section is public iff the header lists it ; an unlisted section is PRIVATE (reachable only along the prompt's own

inheritance chain, never from another prompt). A listing that names no section in the prompt's effective definition raises EXPORT_OF_UNKNOWN_SECTION. A prompt with no `@@@template-body` exposes no section. Cross-template access to a private section raises NOT_EXPORTED.

## 7.6 — What a section renders to

A section that emitted its text and dropped its title would lose information the source states : `@@@section objectif` says *this part is the objective*, and a reader of the rendered prompt has no way to recover that from the prose alone. So rendering carries the structure out with the text, as markdown ATX headings ( `§6.1` PHASE 11) :

| Source | Output |
|---|---|
| `@@@section objectif` | `## Objectif` — the name, first letter upper-cased |
| `@@@section objectif : But de l'analyse` | `## Objectif` — the description is not rendered |
| `@@@section steps.lecture : Lecture` | `### 1. Lecture` — the child segment |
| `@@@section steps.1` | `### 1.` — the rank already says the name |

```
@@@template analyse
@@@name Analyse à sections
@@@section objectif : But de l'analyse
Évaluer la lisibilité du texte fourni.
@@@section etapes : Déroulé
@@@section etapes.lecture : Lecture
Lire le texte en entier, en notant :
1. la première phrase
2. le dernier paragraphe
```

```
@@@section etapes.synthese : Synthèse
Résumer en trois lignes.
```

renders to :

```
## Objectif

Évaluer la lisibilité du texte fourni.

## Etapes

### 1. Lecture

Lire le texte en entier, en notant :
1. la première phrase
2. le dernier paragraphe

### 2. Synthese

Résumer en trois lignes.
```

Four things are worth naming precisely.

**The title is the NAME.** First character upper-cased, nothing else touched (`objectif` → `Objectif`, `output_format` → `Output_format`, `étapes` → `Étapes`); for a sub-section, the child segment alone. The `: description` plays no part — it is the prose the include/exclude dialog shows, and never reaches the output — which is why a prompt whose prose is in one language should name its sections in that language : `@@@section contraintes` reads as `## Contraintes`, and `@@@section constraints : Contraintes STRICTES` as `## Constraints`.

**A description is inert in the output.** `@@@section! rules : ## Règle finale` emits `## Rules` : whatever the description carries, a reserved marker included, is neither emitted nor refused. The reserved levels below govern BODY TEXT, which no phase turns into structure — a description is not body text.

**Sub-section items are NUMBERED, and the marker is what makes them unambiguous.** `### 1.` is the item's rank in its list; a plain `1.` inside the item's own text is the author's list. The two never collide. The rank counts what is RENDERED: a `?` item left OFF leaves no hole, so a parent whose second of three items is excluded emits `### 1.` and `### 2.`. And the rank belongs to the item, not to the render call — an `@@@include p.steps.two` emits `### 2.`, not `### 1.`. An item named by its own index — `steps.1`, the preserved legacy idiom — contributes no title beyond the rank: `### 1.`, not `### 1. 1`, which would say it twice.

**An included prompt gets no title of its own.** An include splices rendered text; it does not open a document. Only the instantiated prompt titles the output. Its *sections*, however, keep their headings — they appear where the include splices them. The one exception is the INLINE form `@@@{include …}`, which splices a phrase mid-sentence and emits no heading at all: a block heading there would break the very line the inflowing form exists to keep.

**Levels 1 to 3 belong to the processor — inside a declared section.** Declaring sections is what makes a prompt L1, and it is there that a competing `##` / `###` is emitted: in the body of a declared section, a line whose first non-whitespace characters open an ATX heading of level 1, 2 or 3 — one, two or three `#` followed by a space, a tab or the end of the line — is INVALID_SYNTAX, and the remedy is usually to make that line a section of its own, declared with `@@@section <name>`: the structure step emits the heading from the name. A fenced code block (`` ``` `` or `~~~`) is code, not headings — a `#` line inside it is emitted verbatim. The check runs in PHASE 1, on the SOURCE, before any heading has been introduced: run afterwards it could only ever reject, the author's markers and the processor's being indistinguishable by then. **Unstructured body content is outside the rule** — a Level 0 source, a prompt that declares no section, the implicit section itself — because no section heading is emitted for it. An identity header declares no structure, so an ordinary Markdown document that merely car-

`ries` @@@template `/` @@@name `keeps its own` # `title` and `##` `parts, verbatim. Level 4 and deeper (` `####` `,` `#####` `,` `######` `) are free: they are ordinary body text and nest under the emitted` `###` `exactly as an author expects. A` # `that is not a marker is untouched (` C# rules `,` issue #12 `,` #tag `),` and @@@verbatim `content is not processed at all — that is the canonical way to ship a literal` # heading` (Chapter IX).

Two structural cases fall out of the rules rather than needing their own : a prompt that declares NO `@@@section` emits its `#` and no `##` (there is no name to show — the body is an implicit single section), and a Level 0 prompt, having no directives at all, emits no heading and renders its raw body wholesale.

## 7.7 — Related features and why

**Inheritance (Chapter XIX).** Sections are the unit of *override*. A child prompt that declares a section with the same `name` as a parent's section replaces the parent's content for that section. New child sections are appended after the inherited ones. The `virtual` / `final` modifiers refine *which* overrides are allowed in the chain.

**`parent` and `self` references (Chapter XX).** `@@@{include parent.<section>}` retrieves the parent's named section *as the parent renders it* — skipping the child's override even if one exists. `@@@{include self.<section>}` retrieves the most specialized version available in the chain. Both rely on sections being named.

**Includes (Chapter XVII).** An include may reference a *specific section* of another prompt using the path form `<prompt>.<section>`. Without sections, the include can only pull the whole prompt's body.

**Optional parameters and parameter sets (Chapters XI, XV).** A parameter set may, by convention, toggle optional sections on or off (the renderer maps a parameter-set switch to an optional-section inclusion). The language does not formalize this mapping ; implementations do.

**Comments and verbatim (Chapters VIII, IX).** Inline and block comments inside a section are stripped at PHASE 11 ; verbatim blocks inside a section pass through to the output. The section delimiters themselves are stripped at PHASE 11 unless the renderer chooses to emit them as headings.

**Virtual templates and final templates (Chapter XXIII).** The section-level `virtual` / `final` modifiers have collection-level counterparts : a *prompt* may carry the `virtual` modifier ( `@@@template virtual <collection_name>.<template_id>` — abstract collection member) or the `final` modifier ( `@@@template final <collection_name>.<template_id>` — sealed collection member) on its header. The two pairs of constructs are symmetric across the OO #1 / OO #2 axes.

## 7.8 — Practical examples

**Trivial — a prompt with three sections.**

```
@@@template plain_reviewer
@@@name Plain Reviewer
@@@description Three-section reviewer template

@@@section role
You are an expert reviewer.

@@@section task
Review the attached document.

@@@section format
Reply in three bullet points.
```

Sections render in declaration order. No overrides, no inheritance — just a structured L1 prompt.

**Idiomatic — overriding one section in a child.**

```
@@@template strict_reviewer inherits plain_reviewer
@@@name Strict Reviewer
@@@description Reviewer with a strict format rule

@@@section format
Reply in three bullet points, each at most fifteen words.
```

The child inherits `role` and `task` from the parent and overrides `format`. The rendered output uses the parent's `role` and `task` followed by the child's `format`.

**Idiomatic — extending a parent section using `parent`.**

```
@@@template annotated_reviewer inherits plain_reviewer
@@@name Annotated Reviewer
@@@description Reviewer that adds a justification after each bullet

@@@section format
@@@include parent.format
After each bullet, justify in one sentence.
```

The child's `format` section pulls the parent's `format` content verbatim with `parent.format`, then appends additional instructions. The effect is *extension,* not replacement.

**Real-world — toggled sections per audience.**

```
@@@template customer_reply
@@@name Customer Reply
@@@description Drafts a customer-support reply

@@@section role
You are a customer-support agent for Acme.
```

```
@@@section task
Reply to the customer's message below.

@@@section? policy_reference : Quote from policy when the question is
Refer to the refunds policy document explicitly.

@@@section! signature : Sign the reply with the agent's name
Sign as "Acme Support, <agent name>".

@@@section format
Reply in plain prose, two paragraphs maximum.
```

The optional `policy_reference` section is excluded by default (`?` — opt-in). The `signature` section is included by default (`!` — opt-out). Both surface their descriptions in the user's include/exclude dialog at instantiation.

**Idiomatic — a VIRTUAL SECTION in a base prompt.**

```
@@@template reviewer_base
@@@name Reviewer Base
@@@description Abstract reviewer — descendants must implement the tas

@@@section role
You are an expert reviewer.

@@@section virtual task

@@@section format
Reply in three bullet points.
```

The `task` section is abstract: the base declares the slot but provides no body. A concrete descendant must implement `task`:

```
@@@template code_reviewer inherits reviewer_base
@@@name Code Reviewer
@@@description Reviews source code
```

```
@@@section task
Review the source code for correctness, performance, and style.
```

Rendering `reviewer_base` alone raises UNIMPLEMENTED_VIRTUAL_SECTION ; rendering `code_reviewer` succeeds with the concrete `task` .

**Idiomatic — a FINAL SECTION sealing boilerplate.**

```
@@@template licensed_reviewer_base
@@@name Licensed Reviewer Base
@@@description Reviewer whose license notice cannot be changed

@@@section role
You are an expert reviewer.

@@@section final license : Boilerplate license text
This reviewer's output is licensed under CC BY-SA 4.0.
```

Any descendant attempting to redeclare `license` — even just to change its description or its toggle — raises OVERRIDE_OF_FINAL_SECTION at PHASE 3. The license slot is sealed.

**Anti-pattern — using sections as ad-hoc comments.**

```
@@@section _comment
TODO: rewrite this section in a friendlier tone.

@@@section task
Review the document.
```

A section is a *content* unit, not a comment carrier. The TODO above renders into the output as if it were a real section, which is almost certainly not what the author wanted. The fix is to use the comment syntax of Chapter VIII :

```
@@@-- TODO: rewrite the task section in a friendlier tone.

@@@section task
Review the document.
```

**Anti-pattern — body on a virtual section.**

```
@@@section virtual task
Review the document.
```

A `virtual` section MUST be empty between its marker and the next section opening. The body above raises VIRTUAL_SECTION_HAS_BODY at PHASE 1. The fix is either to remove the body (keeping the section abstract) or to remove `virtual` (making the section concrete with a default body).

**Anti-pattern — combining `virtual` and `final`.**

```
@@@section virtual final task
```

The two modifiers are mutually exclusive — an abstract slot cannot also be sealed. The declaration raises INVALID_SYNTAX at PHASE 1. The author must pick one: abstract (descendants must implement) or sealed (descendants must not override).

The next chapter covers comments — both the inline and block flavours.

# Chapter VIII — Comments

> ***`standard.md §5.11` — Comment Syntax (excerpt) :***
>
> *POO²L supports a line comment and a bounded comment (the INLINE COMMENT forms), a **tail** comment, and a BLOCK COMMENT form, all introduced by the hyphen marker `--`. All comment forms are REMOVED during instantiation.*
>
> ***`standard.md §5.11.1` — Line, Bounded and Tail Comment Syntax (excerpt) :***
>
> *Line : `@@@-- <comment_text>` (to end of line). Bounded : `@@@(-- <comment_text> )` (any bracket family, any run count). **Tail :** an unquoted `--` (NO SIGIL) on a directive line opens a comment to end of line ; it is removed at PHASE 1 BEFORE the argument grammar runs, so it is NOT a nesting violation and MAY sit at the end of any directive line ( `@@@param x = 1 -- note` ). All three forms are OPAQUE — their content is literal text and the sigil `@@@` is INERT inside them, which is what lets you comment OUT POO²L code. The bounded form is multi-line and NON-NESTING : the FIRST matching close run closes the comment ; raise the run count ( `@@@((-- ... ))` ) when the body contains the closer.*
>
> ***`standard.md §5.11.2` — Block Comment Syntax (excerpt) :***
>
> *Format : `@@@comment ... @@@end-comment` . Everything between the delimiters is removed at instantiation. Multi-line content supported (the primary use case). Block comments MAY be nested (depth tracked : each `@@@comment` increments, each `@@@end-comment` decrements). The block body is OPAQUE : a `@@@--` or `@@@(-- ... )` comment inside it is inert literal text, removed with the block. Use case distinction : `@@@--` for short single-line notes, `@@@(-- ... )` for short in-*

> *line or few-line comments (or to comment out a POO²L fragment), block for longer multi-line notes or for commenting out sizable chunks of prompt source.*

Comments are the simplest construct in POO²L : text that the author wants to write but that the language model should never see. The language provides three forms — line, bounded, and block — all introduced by the hyphen marker `--`, and all removed during instantiation.

## 8.1 — Syntax

The specification's `§5.11` defines a line, a bounded, and a block comment form.

**Line comment** ( `§5.11.1` ):

```
@@@-- <comment_text>
```

The marker is `@@@--` ; the comment runs to the end of the physical line. It is the preferred form for short, single-line notes.

**Bounded comment** ( `§5.11.1` ):

```
@@@(-- <comment_text> )
```

The bounded form gives the comment an explicit right boundary, so it can sit mid-text or span several lines. It is one instance of the bounded form ( `§5.1.3` ): it accepts any of the four bracket families ( `()` , `[]` , `{}` , `<>` ) and may raise the run count ( `@@@((-- ... ))` ) when the body itself contains the closer.

**Block comment** ( `§5.11.2` ):

```
@@@comment
<comment_text, may span many lines>
@@@end-comment
```

The block open keyword is bare (`@@@comment`); the block is closed by the compound reserved keyword `@@@end-comment`. The body between opening and closing is removed at instantiation.

The line and bounded forms are OPAQUE and do not nest — the sigil `@@@` is inert inside them, and the first matching close run closes a bounded comment. Block comments nest by depth.

## 8.2 — Purpose

A POO²L prompt is a source artefact maintained over time. The reasons behind decisions — *why* this `inherits` rather than `include`, *why* that parameter has the constraint it has, *why* the format section is verbatim and not a list — are best written next to the decision, not in a separate document that will drift.

Comments give the author a way to write that material directly into the prompt source without changing the rendered output by a single character. The language model sees nothing of what the comments say; the next author who edits the prompt sees everything.

The three forms exist because different lengths of comment have different ergonomics:

- The *line* form (`@@@--`) is the most compact and is comfortable for short, single-line annotations next to the thing it annotates.
- The *bounded* form (`@@@(-- ... )`) has an explicit right boundary, so it can sit mid-text or run across a few lines, and — because it is opaque — it can *comment out* a fragment of POO²L source.
- The *block* form has open and close keywords and is comfortable for long, multi-paragraph notes or for *commenting out* a sizable chunk of prompt source while a change is being trialled.

Comments are not section content. A section that contains only a comment renders empty.

## 8.3 — Possibilities

**Removal happens at Phase 11.** Comments survive Phase 1 (Parsing) intact so that nested comment depth can be tracked. The actual removal — replacement of the delimited region by nothing — happens at Phase 11 (Output Generation). An author should expect that the *rendered* output contains zero traces of any comment, including no extra whitespace where the comment was.

**Block nesting is tracked by depth; line and bounded forms do not nest.** The line form (`@@@--`) runs to the end of the line and the bounded form (`@@@(-- ... )`) is closed by the first matching close run (`§5.11.1`) — neither nests. Block comments do nest, with their own depth counter (`§5.11.2`): each `@@@comment` increments depth, each `@@@end-comment` decrements it, and the block closes when depth returns to zero.

**The opaque forms inside a block.** A block comment body is opaque: a `@@@--` or `@@@(-- ... )` comment appearing inside it is inert literal text and is removed along with the block. The sigil itself is inert inside any comment form, which is exactly what lets you comment out POO²L source.

```
@@@comment
This block contains a line comment marker that is inert here:
@@@-- this text is removed with the block
The whole block is removed.
@@@end-comment
```

**Use cases that the specification calls out (`§5.11.2`, *Use case distinction*):**

- `@@@--`: preferred for short, single-line notes.
- `@@@(-- ... )`: preferred for short inflowing or few-line comments, or to comment out a fragment of POO²L code (sigil

inert; non-nesting); any bracket family, raise the run count when the body contains the closer.

- `@@@comment ... @@@end-comment`: preferred for longer notes or for commenting out a sizable chunk of prompt source (nests by depth).

**A subtle interaction with verbatim (Chapter IX).** Comments inside a verbatim block are *not* removed — verbatim suspends interpretation of POO²L syntax, and that includes the comment markers. The result is that the comment markers render in the output as literal text. This is rarely what the author wants and is one of the few footguns at the intersection of two POO²L constructs.

**Comments and directive lines.** A parsed directive *argument body* must not contain the sigil `@@@` (no nesting, `§5.1.3`), so a `@@@`-form comment cannot be nested *inside* a `@@@param` / `@@@include` / similar argument list. But a **tail comment** — an unquoted `--` with no sigil — MAY end any directive line (`@@@param x : number = 5 -- headroom`): it is stripped at PHASE 1 before the argument grammar runs, so it is never a nesting violation. The line and bounded `@@@`-comment markers, by contrast, live *between* directives or inside section bodies, not within a directive's arguments.

> ***`standard.md §5.1.3` — Nesting prohibition (excerpt):***
>
> *The body of a parsed bounded construct MUST NOT contain the sigil `@@@` — a parsed body is plain directive payload, and bounded constructs MUST NOT be nested. The opaque bodies (`verbatim`, `comment`, and the `--` bounded comment) are the sole exception: there `@@@` is inert. Block comments and verbatim blocks self-nest by tracking `@@@comment`/`@@@end-comment` and `@@@verbatim`/`@@@end-verbatim` depth.*

**Authorial conventions.** A team often adopts a convention for *what kind* of information goes in comments — the spec does not mandate one. A useful split :

- *Why* notes — explanations of decisions — go in block comments near the construct they explain.
- *To-do* notes go in line comments tagged with a TODO prefix.
- *Examples* of output that the prompt should produce go in verbatim blocks (Chapter IX), not in comments — they are not meant to be stripped.

## 8.4 — Related features and why

**Verbatim (Chapter IX).** Verbatim blocks preserve their content including POO²L syntax ; comments inside a verbatim block render literally. When teaching POO²L itself (a meta-template that explains the language), the author uses verbatim, not comments.

**Deprecated (Chapter XXVI).** A `@@@deprecated "..."` marker is *not* a comment — it has its own semantics (it emits a warning at every instantiation) and its message renders in tooling. A note that says « this prompt is going away » should be a `deprecated` marker, not a comment, so the tooling can act on it.

**Sections (Chapter VII).** A section that contains only comments renders empty. The processor does not warn about empty sections — that is an authoring concern. If a section ends up consistently empty after comments are stripped, the section itself is probably redundant.

**Inheritance (Chapter XIX).** Comments survive the inheritance phase (Phase 3) as nodes in the parsed tree — inheritance operates on that tree, not on rendered text — but they never reach the model, because they are removed at output generation (Phase 11). An author who wants documentation to travel along inheritance *and* still be stripped from the model's view may write it as comments ; an author who wants it to appear in the rendered output should write it as prose inside the section body instead.

**Comments survive Phase 3 and are removed in Phase 11.** The inheritance phase (Phase 3) operates on the parsed tree, which still contains comment nodes. The child's override semantics operate at the section level: a child section with the same name replaces the parent's, *and the replacement happens before Phase 11*. So if the parent had comments in section `task` and the child inherits `task` (no override), those comments travel along with the section into the child's effective body and are removed at Phase 11. If the child overrides `task`, the parent's comments are simply not present in the child's version. Either way, the comments never reach the model.

## 8.5 — Practical examples

**Trivial — a one-line note next to a declaration.**

```
@@@param max_tokens : number = 4096
@@@-- adjust upward only if the model truly needs more headroom
```

The note explains why the default exists, for the next author who might be tempted to raise it casually. The model never sees the note.

**Idiomatic — a block comment explaining a non-obvious decision.**

```
@@@comment
We declared `tone` as `select("formal", "casual")` rather than free
text because the downstream brand-voice review fails when authors
type variants like "professional" or "informal". Stick to the two
options; if a third is genuinely needed, coordinate with the
brand-voice owner before extending.
@@@end-comment

@@@param tone : select("formal", "casual") = "formal", "Tone"
```

The comment captures institutional memory at the point of impact. Six months later, an author tempted to add a third option will see the rationale before making the change.

**Idiomatic — temporarily disabling a fragment during editing.**

```
@@@section role
You are a code reviewer.

@@@comment
@@@section experimental
This experimental section is parked while we A/B test the prompt.
We may bring it back if the test favours it.
@@@end-comment

@@@section task
Review the code.
```

The block comment wraps an entire section declaration. Phase 1 sees the block and it is removed at Phase 11 (Output Generation) ; the rendered output looks exactly as if the experimental section had never been written.

**Anti-pattern — using a comment for a deprecation notice.**

```
@@@template code_review_v2
@@@name Code Review v2
@@@description Older reviewer, kept for backward compatibility

@@@-- DEPRECATED: please migrate to code_review_v3.
...
```

The comment is invisible to tooling. Implementations cannot collect « all deprecated templates » because a comment is just text. The correct construct is the `@@@deprecated "..."` marker (Chapter XXVI) :

```
@@@template code_review_v2
@@@name Code Review v2
@@@description Older reviewer, kept for backward compatibility
@@@deprecated "Use code_review_v3 instead — migration deadline 2026-06-0
...
```

The next chapter covers the other suppression construct: *verbatim blocks*, which keep content in the output but suspend POO²L interpretation inside them.

# Chapter IX — Verbatim Blocks

> ***`standard.md §5.12` — Verbatim Syntax (excerpt):***
>
> *Format: `@@@verbatim ... @@@end-verbatim`. Verbatim blocks preserve their content WITHOUT interpretation or transformation. Unlike comments, verbatim content REMAINS in the output but POO²L syntax inside is NOT processed — including the sigil itself: any `@@@` inside a verbatim block is literal output. A bounded `@@@(verbatim ... )` form (any of the four bracket families, `§5.1.3`) is also available; its body is likewise OPAQUE. Verbatim blocks MAY be nested: each `@@@verbatim` increments depth, each `@@@end-verbatim` decrements depth, and the block closes when depth returns to zero.*

A verbatim block is content that survives to the rendered output *and* suspends POO²L interpretation inside itself. It is the language's escape hatch — the construct that lets a prompt quote POO²L source literally without triggering substitution, inclusion, or any other transformation from the binding phases onward.

## 9.1 — Syntax

```
@@@verbatim
<content>
@@@end-verbatim
```

The open keyword is bare (`@@@verbatim`); the block is closed by the compound reserved keyword `@@@end-verbatim`. Anything between the two delimiters is preserved as-is: parameter references, include directives, even other directive-looking fragments are not interpreted. The delimiters themselves are stripped from the rendered output; only the *content* remains.

Verbatim blocks nest. The processor tracks nesting depth : each `@@@verbatim` increments the depth, each `@@@end-verbatim` decrements it, and the block closes when depth returns to zero ( `§5.12` ).

## 9.2 — Purpose

Most POO²L constructs change the output : a parameter is substituted, an include is expanded, a section delimiter is replaced by a heading or stripped. Verbatim is the opposite : a verbatim block is *content*, not a directive, and its inside is shielded from every later phase.

The most common reason to reach for verbatim is a *meta-template* — a prompt whose job is to teach POO²L itself. A meta-template needs to quote POO²L syntax in its body, and without verbatim, every quoted construct would be processed.

```
@@@verbatim
A POO²L parameter looks like this: @@@param tone : text
@@@end-verbatim
```

Without the surrounding verbatim, the inner `@@@param tone : text` would be interpreted as a declaration in the *outer* prompt — exactly what the author was trying to *show*, not *do*.

The second common reason is *escaping* : an author needs the literal sequence `@@@=` to appear in the output because the prompt is teaching about parsers, regexes, or markup, and POO²L's delimiter happens to collide with the subject matter.

The same route covers the **reserved heading levels**. Rendering emits the prompt's structure as `##` / `###` (Chapter VII), so a body-text line opening one of those three levels is INVALID_SYNTAX. A prompt whose *subject* is markdown — a style guide, a template the consumer must receive with its own `##` intact — writes them inside a verbatim block, where nothing is processed and nothing is reserved :

```
@@@section format : Format attendu
Répondre en suivant ce squelette :
@@@verbatim
# Titre
## Résumé
@@@end-verbatim
```

## 9.3 — Possibilities

**Content survives ; delimiters do not.** The text between the opening and closing keywords reaches the rendered output. The `@@@verbatim` and `@@@end-verbatim` lines themselves are stripped.

**Nothing inside is processed.** The binding and resolution phases — Phase 3 (Template inheritance), Phase 6 (Constants), Phase 8 (Includes), and Phase 9 (Parameter substitution) — all skip content that is inside a verbatim region. The parser tracks the region boundaries in Phase 1 ; subsequent phases consult those boundaries before transforming anything.

**Nesting is by depth.** Verbatim blocks may contain other verbatim blocks. Each `@@@verbatim` increments the depth, each `@@@end-verbatim` decrements it, and the outer block closes when the depth counter returns to zero. This is what lets a verbatim block teach about verbatim blocks :

```
@@@verbatim
Here is how to write a verbatim block:
@@@verbatim
inner content here
@@@end-verbatim
The above shows the syntax.
@@@end-verbatim
```

The inner block’s delimiters appear in the rendered output as literal text, exactly as the author wrote them.

**Verbatim interacts with comments in a non-obvious way.** Comments inside a verbatim block are *not* removed; the verbatim region suspends the comment-removal pass along with every other transformation. The comment markers render literally:

```
@@@verbatim
@@@-- this looks like a line comment but renders verbatim
@@@end-verbatim
```

This is sometimes useful (when teaching comment syntax) and sometimes surprising (when the author meant the inner content to be a comment). The fix in the surprising case is to either move the comment outside the verbatim block or remove the verbatim wrapping.

**Verbatim does *not* preserve whitespace at the delimiter boundaries.** The opening and closing keyword lines are stripped, including the newline characters immediately attached to them. The body's internal whitespace is preserved exactly. An author who needs precise indentation around a verbatim region should test the rendered output; the spec leaves boundary-whitespace handling as an implementation detail within the language's flexible-whitespace rules (`§5.1`).

> ***`standard.md §5.1` — General syntax (excerpt):***
>
> *Every POO²L construct begins with the single sigil `@@@`. Whitespace around a directive's keyword and arguments is flexible. A bare `@@@` is INVALID_SYNTAX unless its first following character is a valid starter (a letter, `=`, `-`, `(`, `[`, `{`, `<`, or `@`). To output a literal `@@@`, double the sigil: `@@@@` renders as a literal `@@@`.*

**Verbatim is one of two escape routes for the sigil.** For a whole literal block, a verbatim block is the tool. For a *single* inline literal `@@@`, POO²L also provides the doubling escape: `@@@@` ren-

ders as a literal `@@@` (`§5.1.2`). There is no backslash-escape mechanism ; the two mechanisms above are the only ones, kept deliberately narrow by the closed-extension policy (Chapter XXXIII).

## 9.4 — Related features and why

**Comments (Chapter VIII).** Comments *remove* content ; verbatim *keeps* content but suspends interpretation. The two are complementary, not overlapping. Reaching for one when the other was needed is the most common confusion at this layer of the language.

**Includes (Chapter XVII).** An `@@@{include path}` placed *inside* a verbatim block is not resolved. The literal include directive renders as text. This is exactly what an author teaching includes wants ; nothing else interacts well with verbatim's « suspend interpretation » contract.

**Parameter substitution (Chapter XI).** A `@@@=parameter_name` inside a verbatim block does not substitute. The literal `@@@=parameter_name` renders in the output. Authors of documentation templates rely on this ; authors of operational templates must avoid wrapping a parameter they intended to substitute.

**System variables (Chapter X).** Same as parameters : `@@@=DATE` inside a verbatim block renders literally as `@@@=DATE`, not as today's date.

**Sections (Chapter VII).** A verbatim block may live inside a section ; the section delimiters themselves are not « interpretation » in the substitution-phase sense — they are structural — so the section still identifies a region of the prompt body, but the verbatim content inside is shielded from substitution and inclusion.

**Attachments (Chapter XVI).** Attachment declarations live outside prompt body proper (recommended at the end, per `§4.4.4`) and are not typically wrapped in verbatim. A verbatim region inside a section may *describe* an attachment, but it does not *declare* one.

> ***`standard.md §4.4.4` — Position in prompt (excerpt) :***
>
> *Attachments SHOULD be declared at the END of the prompt, after all content. This ensures the prompt content is immediately visible and readable.*

## 9.5 — Practical examples

**Trivial — quoting a POO²L fragment in a teaching prompt.**

```
@@@template pool2l_cheatsheet
@@@name POO²L Cheat Sheet
@@@description Quick reference for POO²L authors

A required parameter with a label looks like this:

@@@verbatim
@@@param recipient : text, "Recipient name"
@@@end-verbatim

The `?` suffix on the keyword makes it optional.
```

The verbatim block lets the cheat sheet print the parameter syntax literally. Without it, the body would contain a *real* parameter declaration that the renderer would extract and try to bind.

**Idiomatic — preserving a template that should not be processed.**

```
@@@section format
Reply using this template, filling each placeholder with the
client's answer:

@@@verbatim
Name: {NAME}
Date: {DATE}
```

```
Comments: {COMMENTS}
@@@end-verbatim
```

The template uses curly braces with no POO²L meaning, but the verbatim wrapping is a safety net: if a future edit adds a POO²L placeholder that *would* be substituted, the verbatim block prevents the substitution and the template stays intact.

**Real-world — a meta-template that builds POO²L for the model.**

```
@@@template pool2l_codegen
@@@name POO²L Code Generator
@@@description Asks the model to produce a POO²L prompt

@@@param topic : text, "Topic of the generated prompt"

You will produce a POO²L prompt for the topic @@@=topic.
The output must follow this exact shape, with the placeholders
replaced by your generated content:

@@@verbatim
@@@template <generated id>
@@@name <generated name>
@@@description <generated description>

@@@section role
<generated role>

@@@section task
<generated task>
@@@end-verbatim

Return only the source between (but not including) markers.
```

The `@@@=topic` reference *outside* the verbatim block is substituted normally (Phase 9). The verbatim region inside is the template the model must emit verbatim; the renderer does not touch it.

**Anti-pattern — wrapping a real declaration in verbatim by mistake.**

```
 @@@verbatim
@@@param recipient : text, "Recipient name"
@@@end-verbatim

Dear @@@=recipient,

[Draft the body.]
```

The author meant to declare a parameter and use it. The verbatim wrapping shields the declaration from Phase 1, so `recipient` is never registered as a parameter; Phase 9 then raises UNDEFINED_REFERENCE on the usage site (the symbol `recipient` is neither a declared parameter nor a constant nor a system variable). The fix is to remove the verbatim wrapper.

The next chapter covers *system variables* — the names that POO²L resolves at render time without any author-side declaration.

# Chapter X — System Variables

> *`standard.md §5.7.1` (system variables). Implementation reference: the reserved system-variable names in the Rust engine (`src/engine/crates/pool-core/src/parse.rs`).*

A system variable is a name that the processor resolves at render time without the author ever declaring it. The author writes `@@@=DATE` and the rendered output contains today's date. The standard lists twelve names; no more may be added without extending the language, which the closed-extension policy (Chapter XXXIII) forbids. The engine reserves eleven of them natively; `USERNAME` (the login) is *host-provided* — the host shell supplies its value, since a login name is connection data the engine does not compute on its own.

## 10.1 — Syntax

System variables use the same substitution syntax as parameters: the name appears between the standard delimiters.

```
@@@=DATE
@@@=USER
@@@=LANG
```

Conventionally, system-variable names are written in *uppercase* to distinguish them from author-declared parameters. The processor resolves the uppercase names as system variables and the lowercase names as parameters; an author who declares a parameter or constant named `DATE` (or any reserved system-variable name) raises a `SYSTEM_VARIABLE_OVERRIDE` error — the names are reserved and cannot be shadowed.

The complete list :

| Variable | Value | Example |
|---|---|---|
| `@@@=DATE` | Date in `YYYY-MM-DD` form | `2026-05-17` |
| `@@@=TIME` | Time in `HH:MM:SS` form | `14:32:00` |
| `@@@=DATETIME` | Date and time, ISO 8601 | `2026-05-17T14:32:00Z` |
| `@@@=YEAR` | Four-digit year | `2026` |
| `@@@=MONTH` | Two-digit month, `01` – `12` | `05` |
| `@@@=DAY` | Two-digit day of month, `01` – `31` | `17` |
| `@@@=WEEKDAY` | Day name in the renderer’s language | `Sunday` / `Dimanche` |
| `@@@=USER` | Display name of the rendering user | `Jacques Lovi` |
| `@@@=USERNAME` | Login of the rendering user (host-provided) | `jacques.lovi` |
| `@@@=LANG` | Locale of the rendering user | `fr` or `en` |
| `@@@=GROUPS` | Comma-separated groups of the user | `dev, qa` |
| `@@@=GROUPS_ADMIN` | Groups the user administers | `dev` |

## 10.2 — Purpose

A prompt that says « today is… » should not require the author to remember the date every time. A prompt that addresses the user by name should not need a parameter the user types into a dialog. A prompt whose locale matters should not require a parameter set per language. System variables are the language's answer to « context the renderer knows that the author should not have to plumb through. »

The set is intentionally small. The specification commits to a closed catalogue : twelve names that cover the universally useful axes (time, identity, locale, group membership) and nothing else. An author who needs a custom value — the current sprint number, the customer's tier, the model's temperature — must use a parameter or a constant. The language does not extend the system-variable namespace because that would be a language extension, forbidden by Chapter XXXIII.

## 10.3 — Possibilities

**Resolution at Phase 9.** System variables substitute in the same phase as user parameters (Phase 9 of the processing model — Chapter III, Chapter XXX). Their values come from the *renderer's* environment : the wall-clock time at render, the user identity of the request, the locale flag passed to the renderer.

**Locale of `WEEKDAY` is the renderer's locale.** If the rendering user's locale is French, `WEEKDAY` is `Dimanche` ; if English, `Sunday` . A prompt that depends on a specific casing or language for the weekday should not use `WEEKDAY` directly ; it should branch on `LANG` and use the appropriate fallback (which, in POO²L's no-control-flow world, means producing the right prose in advance and letting the model handle the locale conditional).

**`USER` versus `USERNAME` .** They are DISTINCT : `USER` is the rendering user's *display name*, `USERNAME` is their *login*. The engine reserves `USER` (and the other ten time / locale / group names) nati-

vely ; `USERNAME` is *host-provided* — the host shell supplies the login, since it is connection data the engine does not compute. An author who needs the human-readable name uses `USER` ; one who needs the account handle uses `USERNAME` .

**`GROUPS` and `GROUPS_ADMIN` .** Both produce a comma-separated list, in alphabetical order, of the user's group memberships and the groups they administer respectively. The list is a *string*, not a structured value — POO²L has no array type. An author who needs to act differently per group typically does so through a parameter set bound to the group's identity rather than by parsing `GROUPS` inside the prompt.

**Reserved — no shadowing.** A prompt that declares a `param` or `const` named identically to a system variable does NOT shadow it : the substitution `@@@=name` always resolves to the system value, so the declaration could never take effect. The language treats this as an error — `SYSTEM_VARIABLE_OVERRIDE` (Chapter XXXI) — raised at validate / save time, not deferred to render. Choose a non-reserved identifier instead.

**Verbatim shields system variables.** A system-variable reference inside a `@@@verbatim` block (Chapter IX) renders literally, the same way as a parameter would. The verbatim region suspends Phase 9 substitution.

**One substitution form only.** There is a single way to reference a system variable in a prompt : `@@@=NAME` (or the bounded `@@@{=NAME }` ). POO²L has no double-curly-brace `{{NAME}}` form — no « dashboard » syntax and no legacy `@@@=_NAME_` form. Anything that looks like `8 octobre 2026` in a rendered prompt is literal text, not a substitution.

**No custom system variables.** The closed-extension policy (Chapter XXXIII) forbids implementations from adding new system-variable names. An implementation that wishes to offer additional

context to templates must do so through parameter sets, constants pre-populated by the renderer, or some other application-level mechanism that does not extend the language.

## 10.4 — Related features and why

**Parameters (Chapter XI).** Parameters are values the *author* declares ; system variables are values the *processor* provides. A parameter or constant named `USER` (or any reserved name) is rejected with `SYSTEM_VARIABLE_OVERRIDE` rather than shadowing the system variable. The convention of uppercase names for system variables exists precisely to keep the two namespaces distinct.

**Constants (Chapter XIV).** A constant is a value the author fixes at definition time ; a system variable is a value the renderer provides at render time. If a prompt needs a *date* that is fixed when the prompt is written (a project start date, a contract expiration), that is a constant, not `DATE` .

**Localization (Chapter XXXIV, *Implementer notes*).** The `LANG` variable, together with the rendered output's encoding, is what implementations use to localize the prompt's wrapping text (system prefixes, error messages). A POO²L prompt itself is locale-agnostic in its body ; the body's *content* is whatever the author typed. `LANG` lets the author branch the *content* on locale by producing the appropriate prose in advance.

**Phase 9 substitution.** All twelve system variables are substituted in Phase 9, alongside user parameters. A reader who wants to know when in the pipeline a `@@@=DATE` becomes a real date can answer « Phase 9 » without consulting the implementation.

## 10.5 — Practical examples

**Trivial — dating an instruction.**

```
@@@template audit_snapshot
@@@name Audit Snapshot
```

```
@@@description Asks the model to date-stamp its analysis

Today is @@@=DATE. Treat any data in the attached file as
current to this date.
```

The rendered output substitutes today's date ; the model uses it to anchor its responses.

**Idiomatic — addressing the user by name.**

```
@@@template personal_briefing
@@@name Personal Briefing
@@@description A briefing addressed to the requesting user

@@@section greeting
Hello @@@=USER,
```

The greeting personalizes without a parameter dialog. The user who clicks « Render » sees their own name ; another user clicking the same prompt sees their own.

**Real-world — branching on locale.**

```
@@@template bilingual_greeting
@@@name Bilingual Greeting
@@@description Greets the user in the user's locale

@@@param? subject : text = "today's tasks", "Subject of the message"

@@@section greeting
Hello @@@=USER — your locale is @@@=LANG.
The subject today is: @@@=subject.
```

POO²L cannot run an `if` on `LANG` — control flow is forbidden (Chapter XXXIII). The prompt instead surfaces `LANG` to the model and lets the model produce the locale-appropriate reply.

For templates that must produce *exactly different content* per locale, the idiomatic approach is one prompt per locale, organized in a collection, with a parameter set selected by the renderer according to `LANG`. That is an L5 pattern; the simpler approach above suffices for most cases.

**Anti-pattern — using `DATE` as a project deadline.**

```
@@@section contract
The contract expires on @@@=DATE.
```

`DATE` is *today's* date at render time, not a fixed value. If the contract expires on a specific calendar date, that date is a constant:

```
@@@const EXPIRY : date = "2027-01-01", "Contract expiry"

@@@section contract
The contract expires on @@@=EXPIRY.
```

The next chapter — and the first of Part III — covers *parameters*, which generalize the substitution mechanism the reader has just seen for system variables to values the author wants to vary at instantiation time.

# Partie 3

*Variables and binding*

# Chapter XI — Parameters

> ***`standard.md §5.6` — Parameter Syntax (excerpt):***
>
> *Declaration forms: `@@@param <name> : <type> = <default>, "<label>"` (required, with label); `@@@param? <name> : <type> = <default>` (optional — an optional parameter `?` MUST carry a default, because it always resolves); `@@@param <name> : select("a", "b") = "a"` (select); `@@@param <name> : <type> <constraint>=<value> ... = <default>` (with constraints). Rules: `parameter_name` is a Unicode letter followed by letters, digits or `_` (NO hyphen), case- and diacritic-insensitive; `type` is one of `text`, `textarea`, `number`, `date`, `boolean`, `select`; `?` after the keyword denotes optional. Required parameters without binding are an ERROR; optional parameters always resolve to their (mandatory) default.*
>
> ***`standard.md §4.4.2` — Parameters (excerpt):***
>
> *Parameters are named variables within templates that accept runtime values. The label is preceded by a comma separator and enclosed in double quotes; when omitted, the parameter name is used as the display label. The six standard types are `text` (default), `textarea`, `number`, `date`, `boolean`, `select`. `select` MUST declare its valid values inline using parenthesized, comma-separated, double-quoted strings — at least one option is required, and a default if specified must be one of the declared options.*

A parameter is a named variable a prompt declares for the value to be supplied at instantiation time. Parameters are the construct that collapses twelve near-duplicate templates into one parameteri-

zed prompt; they are also the construct that makes a prompt *callable* — including from another prompt's `@@@{include }` with overrides.

## 11.1 — Syntax

The full form, with all the optional bits in their positions:

```
@@@param <name> : <type> <constraints> = <default>, "<label>"
@@@param? <name> : <type> <constraints> = <default>, "<label>"
```

Variants (`§5.6`):

```
@@@param <name> : <type> = <default>              (no label)
@@@param <name> : <type>, "<label>"               (required, no default,
@@@param? <name> : <type> = <default>             (optional — default is
@@@param <name> : select("a", "b") = "a"          (select with options)
@@@param <name> : number min=0 max=100 = 50       (with constraints)
```

Usage — substitution in the body:

```
@@@=<parameter_name>
```

Rules from `§5.6`:

- `<name>` — a Unicode letter then letters, digits, underscore (NO hyphen); case- and diacritic-insensitive.
- `<type>` — one of the six standard types (see below).
- `?` after the keyword denotes an optional parameter — which MUST carry a default (an optional parameter always resolves).
- `<default>` — required on optional parameters; must match the declared type and any constraints; for `select`, must be one of the declared options.

- `<label>` — optional; double-quoted; comes after a comma; used for display in instantiation UIs. When omitted, the parameter name is the display label.

The six standard types (`§4.4.2`):

| Type | Meaning | Default rendering |
|---|---|---|
| `text` | Single-line text (default when omitted) | Text input |
| `textarea` | Multi-line text | Multi-line input |
| `number` | Integer or decimal | Numeric input |
| `date` | Date in `YYYY-MM-DD` | Date picker |
| `boolean` | `true` or `false` | Checkbox/toggle |
| `select` | One of an inline option list | Dropdown |

**The set is CLOSED, and it is checked.** A type name outside these six raises `INVALID_SYNTAX` at PHASE 1 — it is a mistake, not a silent `text`. Omitting the type is not the same thing: `@@@param topic`, with no `:` at all, means `text` by definition. Before the set was enforced, a slip on `number` produced a parameter that behaved as text and said nothing about it.

## 11.2 — Purpose

A prompt without parameters can be reused only by copying it. A prompt with parameters is reused by *instantiating* it — supplying values at the call site. The two flavours that look almost identical to the model (« review this code in a formal tone » versus « review this code in a casual tone ») become one parameterized prompt:

```
@@@param tone : select("formal", "casual") = "formal", "Tone"
Review the code in a @@@=tone tone.
```

The reduction in duplication is the headline benefit. Three secondary benefits matter just as much:

1. **A self-documented contract.** A prompt's parameter list is its *signature* — what the prompt expects to receive. Anyone reading the prompt sees the inputs at a glance, without inferring them from the body's substitution points.

2. **A typed binding.** The processor validates that the supplied value matches the declared type and constraints before any text reaches the model. A `number` parameter cannot receive the string `"high"`; a `select` parameter cannot receive a value outside its option list.

3. **A handle for composition.** Includes (`@@@{include x(p="v")}`), parameter sets, and virtual-template signatures all reference parameters by name. The parameter list is the public API of the prompt.

## 11.3 — Possibilities

**Required vs optional.** A parameter with no `?` after the keyword is *required*: instantiation without a binding for it (and no default on the declaration) raises UNDEFINED_PARAMETER in Phase 9. A parameter with `?` after the keyword is *optional* and MUST carry a default; an unbound optional parameter always resolves to that default (it never silently vanishes).

**Defaults are not bindings.** A default is the value used when no binding is supplied; a binding from instantiation always wins. The default must also satisfy the parameter's constraints — a default that violates a constraint raises CONSTRAINT_VIOLATION.

**Labels are display strings.** The label is *not* an identifier; it is free text inside double quotes, used by tooling for display. It may contain spaces and special characters. Labels are stripped from the rendered output.

**Select options are part of the type.** A `select` parameter without an option list is INVALID_SYNTAX. Options are inline, comma-separated, double-quoted, parenthesized:

```
@@@param lang : select("en", "fr", "de") = "en", "Language"
```

At least one option is required (`§4.4.2`). The default, if specified, must be one of the declared options.

**Constraints are declarative.** A parameter may carry validation constraints — `min`, `max`, `minlen`, `maxlen`, `pattern` — placed between the type (and options, if `select`) and the default. The full treatment is Chapter XII; the short version is that constraints are key=value pairs validated at Phase 9.

**Each key belongs to certain types, and the binding is enforced.** `min`/`max` bound `number` and `date`; `minlen`/`maxlen`/`pattern` bind `text` and `textarea`. A key outside the five, a key on a type that does not allow it, and a repeated key all raise `INVALID_CONSTRAINT` at PHASE 1 — none of them is ignored. A constraint an author believes they set, and that quietly does not apply, is worse than one refused: `minlen=2` on a `number` used to pass and constrain nothing.

**Substitution is a textual replacement.** When Phase 9 substitutes `@@@=tone` with the bound value, it produces the value's *string* form. A boolean substitutes as `"true"` or `"false"`; a number substitutes as its decimal representation; a date substitutes as the `YYYY-MM-DD` literal. There is no implicit quoting or escaping; the substituted text becomes part of the rendered prompt exactly as-is.

**System variable names are reserved — no shadowing.** A parameter (or constant) declared with the name of a reserved system variable (`DATE`, `USER`, `LANG`, …) does NOT shadow it: the substitution `@@@=name` always resolves to the system value, so the declaration could never take effect. The language rejects this as `SYSTEM_VARIABLE_OVERRIDE` at validate / save time. Choose a non-reserved identifier.

**Reserved keywords are off-limits.** A parameter cannot be named `section`, `include`, `inherits`, `param`, `const`, `attachment`, `parameter-set`, `collection`, `deprecated`, `alias`, `verbatim`, `comment`, or any other keyword listed in `§5.1`. Using a reserved keyword is INVALID_SYNTAX.

> ***`standard.md §5.1` — Reserved keywords (excerpt):***
>
> *Reserved keywords MUST NOT be used as identifiers for parameter names, constant names, attachment names, section names, or collection names. Using a reserved keyword as an identifier is an ERROR.*

**Parameter declaration order is free.** The processor accepts declarations in any position within the prompt. The recommended writing order places parameters after identity and lifecycle markers, before the body (`§4.5`, *Standard declaration order*), purely for readability.

> ***`standard.md §4.5` — Standard declaration order (excerpt):***
>
> *POO²L generators SHOULD organize declarations in the following order: Identity, Lifecycle, Parameters, Constants, content, Attachments. Processors MUST accept declarations in ANY order — the order of appearance has NO semantic significance for processing.*

## 11.4 — Related features and why

**Constraints (Chapter XII).** A parameter's constraints are an optional addition that adds Phase 9 validation. Without constraints, the parameter accepts anything of its declared type; with constraints, the parameter has bounds the renderer enforces before the model sees the prompt.

**Constants (Chapter XIV).** A constant is the *immutable* sibling of a parameter. Both share the type and label syntax ; the difference is that a constant is bound once at definition time and never again. The choice between parameter and constant is the choice between « this value should be asked of the user » and « this value is fixed by the author ».

**Parameter sets (Chapter XV).** A parameter set bundles multiple parameter bindings under a name ; the caller selects a set instead of typing the values one by one. Parameter sets reference parameters by name ; renaming a parameter requires updating every set that binds it.

**Includes with overrides (Chapter XVIII).** An `include` may pass parameter overrides : `@@@{include greeting(name="John", formal=true)}` . The override binds the *target* prompt's parameters for the duration of the included render.

**Inheritance (Chapter XIX).** A child prompt inherits its parent's parameter declarations. The child may add new parameters, but it cannot remove a parent's parameter (the parent's body presumably substitutes it). The child may override a parameter's *default* by redeclaring the parameter with a different default.

**Virtual templates (Chapter XXIII).** A virtual prompt's parameter list is its *contract*. A concrete override in a derived collection must implement at least the same parameters with the same types ; adding parameters is allowed, removing them or changing their types is a contract break.

## 11.5 — Practical examples

**Trivial — a parameterized greeting.**

```
@@@template email_greeting
@@@name Email Greeting
@@@description Greets a recipient in a chosen tone

@@@param recipient : text, "Recipient name"
```

```
@@@param tone : select("formal", "casual") = "formal", "Tone"

@@@section greeting
Dear @@@=recipient,

[Draft the rest of the email in a @@@=tone tone.]
```

Two parameters : one free-text required, one `select` with a default. Instantiation supplies a recipient ; tone defaults to « formal » unless the caller overrides it.

**Idiomatic — an optional parameter with a typical default.**

```
@@@template document_summary
@@@name Document Summary
@@@description Summarizes a document with optional length limit

@@@param document_title : text, "Document title"
@@@param? max_words : number min=50 max=2000 = 200, "Maximum words"

@@@section task
Summarize the document titled "@@@=document_title" in at most
@@@=max_words words.
```

The `max_words` parameter is optional and bounded between 50 and 2000 ; if the caller does not supply it, the default of 200 is used.

**Real-world — a textarea with a length cap and a substituted body.**

```
@@@template customer_reply
@@@name Customer Reply
@@@description Drafts a reply to a customer message

@@@param customer_name : text, "Customer name"
@@@param customer_message : textarea minlen=10 maxlen=2000, \
   "Customer's original message"
```

```
@@@section task
You are replying to @@@=customer_name. Their message:

@@@=customer_message

Draft a reply in a friendly, helpful tone.
```

The `textarea` type signals to the renderer that a multi-line input is appropriate; the `minlen` and `maxlen` constraints reject empty or pathologically long inputs before the model is invoked.

**Anti-pattern — making a value a parameter when it never varies.**

```
@@@template brand_voice_memo
@@@name Brand Voice Memo
@@@description Memo in the company voice

@@@param company_name : text = "Acme Corp", "Company name"
@@@param brand_voice : text = "professional, plain-spoken", \
   "Brand voice description"

@@@section instructions
Write a memo for @@@=company_name in a @@@=brand_voice voice.
```

The company name and brand voice never vary; they are not parameters but constants:

```
@@@const COMPANY_NAME : text = "Acme Corp", "Company name"
@@@const BRAND_VOICE : text = "professional, plain-spoken", \
   "Brand voice description"

@@@section instructions
Write a memo for @@@=COMPANY_NAME in a @@@=BRAND_VOICE voice.
```

The next chapter covers the constraints a parameter may carry — the `min`, `max`, `minlen`, `maxlen`, and `pattern` keys that were introduced here in passing.

# Chapter XII — Constraints

> ***`standard.md §4.4.2` — Declarative constraints (excerpt):***
>
> *Parameters MAY include declarative validation constraints — key=value pairs placed after the type (and options, if `select`) but before the default value. Constraints are purely declarative: they define bounds, not logic. Available keys: `min`/`max` for `number`; `minlen`/`maxlen`/`pattern` for `text` and `textarea`. `pattern` uses the ECMAScript (RegExp) regular-expression dialect of ECMA-262 — the flavor is NORMATIVE, not implementation-defined. Implementations MUST validate; a violation is a CONSTRAINT_VIOLATION error (a malformed constraint is INVALID_CONSTRAINT). Constraints are informational for LLMs (included in rendered prompt context) and apply to optional parameters when bound and to their mandatory default.*
>
> ***`standard.md §5.6` — Parameter constraint syntax (excerpt):***
>
> *Form: `@@@param <name> : <type> <constraint>=<value> ... = <default>, "<label>"`. Constraint keys: `min`, `max` (number, inclusive bounds); `minlen`, `maxlen` (text/textarea, character count); `pattern` (text and textarea, an ECMAScript regex, double-quoted). String-valued constraints use double quotes; numeric constraints use bare numbers.*

A constraint is a declarative validation rule attached to a parameter. It defines a bound or a pattern that any bound value must satisfy — not a calculation, not a transformation, just a contract the processor enforces in Phase 9.

POO²L's constraint set is small by design. Five keys, each tied to a parameter type, cover the validation needs of every prompt without introducing computational machinery.

## 12.1 — Syntax

Constraints appear as `key=value` pairs between the parameter's type (and `select` options, if any) and the default. They are space-separated.

```
@@@param <name> : <type> <constraint>=<value> ... = <default>, "<label>"
```

Forms :

```
@@@param age : number min=0 max=150 = 25, "Patient age"
@@@param title : text maxlen=100, "Document title"
@@@param code : text pattern="[A-Z]{3}-[0-9]{4}" = "ABC-0001"
@@@param? bio : textarea maxlen=500, "Short biography"
@@@param count : number min=1 max=100 = 5
```

The five constraint keys ( `§4.4.2` ):

| Key | Applies to | Meaning |
|---|---|---|
| `min` | `number` | Minimum allowed value (inclusive). |
| `max` | `number` | Maximum allowed value (inclusive). |
| `minlen` | `text` , `textarea` | Minimum character length. |
| `maxlen` | `text` , `textarea` | Maximum character length. |
| `pattern` | `text` , `textarea` | ECMAScript regex the value must match. |

String-valued constraints (just `pattern` today) use double quotes around the value. Numeric constraints use bare numbers.

The implicit *option list* of a `select` parameter is itself a constraint of sorts — it restricts the value to the declared list — but the specification treats it as part of the type, not as a separate key.

## 12.2 — Purpose

A parameter without constraints accepts anything of its declared type. A `text` parameter receives any string ; a `number` parameter receives any number. For most templates that is enough. But for the templates where it is not — where a number must be a positive integer under a hundred, where a code field must match a specific format, where a textarea must not be empty — the language gives the author a way to write the validation rule next to the declaration.

Two payoffs :

1. **Errors caught early.** A violation surfaces in Phase 9, before any text reaches the model. The instantiating caller sees a precise message (« `age` must be between 0 and 150 ; got 9999 ») instead of nonsensical model output.

2. **The constraint is documentation.** A reader of the prompt sees the bounds without consulting external documentation. The parameter list becomes the prompt's API in a stronger sense : not just the names and types but the *legal value spaces*.

Constraints are *declarative* : they define bounds, not logic. The language deliberately stops short of conditional validations (« if X then Y must… »), arithmetic (« max is twice min »), or cross-parameter rules. Those belong to control flow, which Chapter XXXIII forbids.

## 12.3 — Possibilities

**Where constraints fit syntactically.** Between the type (and `select` options) and the default. The order within the constraint list does not matter — `min=0 max=10` is the same as `max=10 min=0`.

**Combinations.** `min` and `max` may both appear on the same `number`; `minlen` and `maxlen` may both appear on the same `text` or `textarea`; `pattern` may combine with `minlen`/`maxlen` on a `text`. Combinations that make logical sense are accepted; the processor does not check that the combination is *useful* (a `pattern` plus a `maxlen` that the pattern already implies is redundant but not an error).

**Phase 9 enforcement.** Constraints are validated in Phase 9 — the same phase that substitutes parameter values. A violation raises CONSTRAINT_VIOLATION. For `select`, the more specific error SELECT_INVALID_OPTION distinguishes the option-list case from the general constraint failure.

**Optional parameters and constraints.** Constraints apply to the bound value; an optional parameter always resolves to its (mandatory) default, which must itself satisfy the constraints. A default in violation of its own constraint raises CONSTRAINT_VIOLATION.

**Pattern syntax.** The regex flavor is NORMATIVE: `pattern` uses the ECMAScript (RegExp) dialect of ECMA-262, and applies to both `text` and `textarea`. Every conformant implementation must interpret the same pattern the same way; a pattern that is not a valid ECMAScript regex is INVALID_CONSTRAINT. Escaping inside the regex follows ECMAScript rules; the surrounding double quotes need no internal escaping unless the regex itself contains a double quote.

**Constraints are visible to the model.** `§4.4.2` states that constraints are « informational for LLMs (included in rendered prompt context) ». An implementation may surface constraint des-

criptions in the rendered output so the model understands the value space. The exact rendering is implementation-defined; the language only mandates the validation.

**Constraints do not enforce semantics, only shape.** A `text` with `pattern="[a-z]+"` allows the string `"banana"` even if `"banana"` is not a meaningful value for the parameter. Semantic validation belongs to the prompt body and the model.

## 12.4 — Related features and why

**Parameters (Chapter XI).** Constraints are the optional validation layer on a parameter. A parameter without constraints is fully valid; constraints add a contract.

**The `select` type (Chapter XI).** The option list of a `select` parameter is itself a value-space restriction. Violations of the option list use the dedicated error SELECT_INVALID_OPTION rather than the general CONSTRAINT_VIOLATION.

**Constants (Chapter XIV).** A constant may declare the same constraints as a parameter. The processor validates the constant's value at definition time (Phase 6, Constant Resolution) against the constraints; a constant whose value violates its own constraint is a Phase 6 error.

**Includes with overrides (Chapter XVIII).** A parameter override on an `@@@{include }` site must satisfy the target prompt's constraints. The override is bound *after* the target's parameters are declared but *before* substitution; the constraint validation still happens in Phase 9.

**Parameter sets (Chapter XV).** A parameter set's bindings must satisfy the parameters' constraints. A set whose value violates a constraint surfaces the error at the binding phase (Phase 5) or at the substitution phase (Phase 9), depending on when the violation is detected.

**Inheritance (Chapter XIX).** A child prompt inherits a parent's parameter declarations including their constraints. Overrides may only **relax** a constraint, by monotonic rules (Phase 3.b, call-contract): `min` / `minlen` MAY decrease or be removed but MUST NOT increase; `max` / `maxlen` MAY increase or be removed but MUST NOT decrease; `pattern` MUST be identical or removed, never replaced by a different one. Tightening a constraint in a child breaks the substitutability contract and raises OVERRIDE_CONTRACT_VIOLATION.

**Assertions and scopes (Chapter XIII).** A constraint validates a parameter *value* in Phase 9; an *assertion* is the text-level counterpart, validating the *rendered content* in Phase 10. The two form one validation story — constrain the inputs, then assert the result. See the chapter on assertions for the predicate set and the `@@@scope` boundary.

## 12.5 — Practical examples

**Trivial — a numeric range.**

```
@@@param age : number min=0 max=150 = 30, "Age in years"

The patient is @@@=age years old.
```

Phase 9 rejects bindings of `-1` or `200`. The default of `30` satisfies the bounds (Phase 1 verifies this at parse time).

**Idiomatic — a regex on an identifier field.**

```
@@@param ticket_id : text pattern="[A-Z]{3}-[0-9]{4}" = "ABC-0001", \
   "Ticket identifier"

The ticket is @@@=ticket_id.
```

The format `XYZ-1234` is enforced before the model sees the prompt. A user who types `XY-1234` or `XYZ-12345` gets a clear error in the renderer.

**Idiomatic — a length cap on user-supplied text.**

```
@@@param customer_message : textarea minlen=10 maxlen=2000, \
   "Customer's original message"

Reply to the customer's message below:

@@@=customer_message
```

The lower bound rejects empty inputs; the upper bound rejects pathologically long ones (the model's context budget would otherwise be at risk). The constraints document the prompt's expectation in a way the renderer can act on.

**Real-world — a parameter set whose values are validated by the parameter's constraints.**

```
@@@template account_outreach
@@@name Account Outreach
@@@description Outreach email parameterized by tier and budget

@@@param tier : select("free", "pro", "enterprise") = "pro", \
   "Customer tier"
@@@param budget : number min=0 max=1000000 = 5000, \
   "Account budget in USD"

@@@parameter-set HighValueOutreach
tier: "enterprise"
budget: 250000
@@@end-parameter-set

@@@parameter-set EarlyStage
tier: "free"
budget: 0
```

```
@@@end-parameter-set

@@@section task
Draft an outreach email for a @@@=tier-tier account with
a budget of @@@=budget USD.
```

Both parameter sets bind values that the parameter's constraints accept. A set that violated a constraint would be rejected at Phase 5.

**Anti-pattern — cross-parameter validation in disguise.**

```
@@@param min_words : number min=0 max=10000 = 100, "Minimum"
@@@param max_words : number min=0 max=10000 = 500, "Maximum"
```

The author probably wants `max_words >= min_words`. The language cannot express that — constraints are single-parameter and non-computational (Chapter XXXIII). The constraint hides the non-expressible intent rather than capturing it. Solutions are non-linguistic: validate in the calling code, or restructure the prompt to take a `word_target : number` and let the model treat it as approximate.

The next chapter covers *assertions and scopes* — the text-level analogue of these value constraints, validating the rendered prompt itself rather than the inputs that feed it.

# Chapter XIII — Assertions and Scopes

> ***`standard.md §5.15.1` — `@@@assert` (excerpt):***
>
> *`@@@assert <predicate>` is a LINE DIRECTIVE; each assertion stands on its own line and is ALWAYS BLOCKING — an unsatisfied predicate raises the typed error `ASSERTION_FAILED`. The only predicates are `contains "<text>"`, `not contains "<text>"`, `matches "<pattern>"`, `minlen <N>`, `maxlen <N>`. There are NO boolean combinators (`and` / `or` / `if`); multiple assertions combine as an IMPLICIT AND.*
>
> ***`standard.md §5.15.3` — `@@@scope … @@@end-scope` (excerpt):***
>
> *A `@@@scope` is a PURE VALIDATION BOUNDARY: transparent to output (its content renders normally), with no structural effect (it is not a section, not addressable, not a target of `include` or inheritance). An optional label — `@@@scope "<label>"` — names the scope for readable diagnostics. Partial overlap of a scope and a section raises `MALFORMED_SCOPE`.*

An assertion is the text-level analogue of a parameter constraint (Chapter XII). Where a constraint bounds a *value* before it is substituted — `min`, `max`, `pattern` — an assertion bounds the *rendered content* after everything has resolved. It states what the finished prompt MUST satisfy, never how to produce it, and never alters the output. Together with the parameter constraints of the previous chapter, assertions form POO²L's two-layer validation story: constrain the inputs, then assert the result.

A scope is the companion construct : a pure boundary that says *which* rendered text a set of assertions governs. It introduces no flow control and no new section — it exists only to bound assertions more finely than, or across, the natural section boundaries.

## 13.1 — Syntax

An assertion is a single line directive carrying one predicate :

```
@@@assert <predicate>
```

The five predicates — and there are only five — are :

| Predicate | The rendered content MUST… |
|---|---|
| `contains "<text>"` | contain the literal text |
| `not contains "<text>"` | not contain the literal text |
| `matches "<pattern>"` | match the regular expression pattern |
| `minlen <N>` | be at least N characters long |
| `maxlen <N>` | be at most N characters long |

There is deliberately no `@@@assert?` (non-blocking) form : an assertion is always blocking, because a discreet `?` on so critical a mechanism is error-prone. There are no boolean combinators either — no `and`, `or`, or `if`. Multiple assertions in the same scope simply combine as an implicit AND : every one of them must hold. `matches` (a regular expression) already expresses starts-with, ends-with, and occurrence-count style checks, so no dedicated predicates for those are needed.

A scope is a paired boundary :

```
@@@scope
...content...
@@@end-scope
```

or, with a label for readable diagnostics :

```
@@@scope "confidentiality clause"
...content...
@@@end-scope
```

The closer is the hyphenated keyword `@@@end-scope`, consistent with the other block closers (`@@@end-verbatim`, `@@@end-comment`, `@@@end-parameter-set`).

## 13.2 — Scope of an assertion

An assertion validates the ENTIRE rendered content of its NEAREST ENCLOSING SCOPE, *independently of its own position* within that scope. Placing an assert at the top, middle, or bottom of its scope makes no difference — it sees the whole scope. The enclosing scope is the innermost of :

```
@@@scope | section | include | prompt
```

- An assert inside a **section** validates that whole section. This is the default granularity — the section is the natural unit.
- An assert in the **preamble** (no enclosing section or `@@@scope`) takes the ENTIRE TEMPLATE as its scope.
- An assert inside an **include** validates the included content.
- An assert inside a `@@@scope` validates exactly that scope — the tool to reach *finer* than a section, or to span *several* sections.

When scopes nest, an assertion binds to the innermost one.

### Nesting and `MALFORMED_SCOPE`

Sections, `@@@scope` boundaries, includes, and blocks form a PROPERLY NESTED tree — no construct may straddle another. A `@@@scope` MAY sit entirely within one section, OR enclose one or more *whole* sections (a cross-section invariant). What is forbidden is PARTIAL OVERLAP : a `@@@scope` opened in the middle of a section, where a new `@@@section` marker begins before the matching `@@@end-scope`, raises `MALFORMED_SCOPE`. An `@@@end-scope` implicitly closes any section it encloses. The error is a structural one, raised during assertion evaluation (Phase 10) as the scope nesting is validated, not a verdict on content.

## 13.3 — Purpose

A parameter constraint protects the *inputs* ; an assertion protects the *result*. Some invariants cannot be stated as a constraint on any single value — they are properties of the assembled text after substitution, includes, and inheritance have all run. Two payoffs :

1. **A guarantee about the finished prompt.** « This disclaimer must always be present », « this prompt must never leak the word *guarantee* », « the body must stay under 500 characters » — these are claims about the rendered output, enforced before a single token reaches the model.

2. **The assertion is documentation.** A reader sees, next to the content, exactly what the author considers non-negotiable about it. The check and the contract live in the same place.

Like constraints, assertions are *declarative* : they define what must hold, not how to achieve it, and never any conditional or computational logic. That restraint is what keeps rendering deterministic — the same bindings produce the same rendered content and therefore the same verdict, every time. Conditional or cross-field logic belongs to control flow, which Chapter XXXIII forbids.

## 13.4 — Possibilities

**Evaluation is late, and deterministic.** Assertions run in Phase 10 (Assertion Evaluation) — *after* substitution, includes, and inheritance have fully resolved, and *before* output generation in Phase 11. On success the rendered content is emitted unchanged ; on failure instantiation is blocked with `ASSERTION_FAILED`. (See the processing model, Chapter III, and the resolution phases, Chapter XXX.)

**Inheritance falls out of lexical scoping.** No special rule is needed. An assertion validates *its own* scope. If a child prompt overrides the section that carries an assertion, the parent's content *and* its assertion are both replaced — so the assertion governs the parent's version, not the child's specialization. A child that does not override that section inherits the parent's content together with its assertion.

**A scope changes nothing about the output.** Unlike `verbatim` or `comment`, a scope transforms no text. Its content renders normally ; the boundary exists purely to delimit assertions. Use it to scope more finely than a section (a single sensitive clause within a larger section) or to assert across several whole sections at once (a tone check spanning a greeting and a closing).

## 13.5 — Boundary — rendered prompt only

Assertions apply to the RENDERED TEMPLATE ONLY — the input POO²L builds. Validation of the LLM's *response* — schema-matching the model's output, say — is explicitly OUT OF SCOPE for the language : POO²L constructs the input and never observes the response. Response validation is a tooling or evaluation concern, not a language feature, and belongs outside the prompt. This boundary is deliberate : it is what lets an assertion's verdict stay deterministic.

## 13.6 — Related features and why

**Constraints (Chapter XII).** The sibling layer. A constraint validates a parameter *value* in Phase 9 ; an assertion validates the *rendered content* in Phase 10. Constraints answer « is this input legal ? » ; assertions answer « is the finished prompt correct ? ». Reach for a constraint when the rule is about one value's shape ; reach for an assertion when the rule is about the assembled text.

**Sections (Chapter VII).** The default scope of an assertion is its enclosing section, and `@@@scope` is the way to validate *more finely* than a section without introducing a real (addressable, includable) one.

**Includes (Chapter XVII).** An include is one of the four scope kinds : an assertion placed inside included content validates that content.

**The closed-extension principle (Chapter XXXIII).** The five-predicate, no-combinator design is a direct expression of POO²L's refusal of control flow. Anything richer — conditionals, arithmetic, response inspection — is intentionally left to external tooling.

## 13.7 — Practical examples

**Trivial — a section-scoped guarantee.**

```
@@@section disclaimer : Legal disclaimer
This output is provided for informational purposes only and does not
constitute legal advice.
@@@assert contains "informational purposes only"
@@@assert not contains "guarantee"
@@@assert maxlen 500
```

The three assertions combine as an implicit AND. All three validate the whole `disclaimer` section, regardless of where they sit in it.

**Idiomatic — a preamble assert over the whole prompt.**

```
@@@assert minlen 50
@@@section intro : Introduction
...
@@@section body : Body
...
```

With no enclosing section or scope, the assertion takes the entire prompt as its scope: the rendered prompt as a whole must be at least 50 characters.

**Idiomatic — an explicit scope finer than a section.**

```
@@@section policy : Policy
General handling rules apply.
@@@scope "confidentiality clause"
Do not reveal internal identifiers or customer records.
@@@assert contains "Do not reveal"
@@@end-scope
Continue with the standard procedure.
```

The assertion governs only the confidentiality clause, not the whole `policy` section. The label makes the diagnostic readable on failure.

**Real-world — a cross-section invariant.**

```
@@@scope "tone"
@@@section greeting : Greeting
Hello, and welcome.
@@@section closing : Closing
Thank you for your time.
@@@end-scope
@@@assert matches "(?i)welcome"
```

A single scope encloses two whole sections ; the assertion checks a tone invariant across both.

**Anti-pattern — a scope that straddles a section boundary.**

```
@@@section a : First
Some content.
@@@scope
More content.
@@@section b : Second
@@@end-scope
```

The scope opens in the middle of section `a` but does not close before section `b` begins — partial overlap. This raises `MALFORMED_SCOPE` during scope-nesting validation (Phase 10). The fix is to either close the scope before `b` opens, or open it before `a` so that it encloses both whole sections.

The next chapter covers *constants*, the immutable sibling of the parameter.

# Chapter XIV — Constants

> ***`standard.md §5.7` — Constant Syntax (excerpt):***
>
> *Format: `@@@const <NAME> : <type> = <value>, "<label>"` (with label) or `@@@const <NAME> : <type> = <value>` (without label). Rules: `NAME` is alphanumeric plus underscore, uppercase by convention; `type` is REQUIRED (one of `text`, `textarea`, `number`, `date`, `boolean`, `select`); `value` is REQUIRED and must literal-match the type; constants are immutable after definition.*
>
> ***`standard.md §4.4.3` — Constants (excerpt):***
>
> *Constants are named immutable values bound at definition time. They are essentially parameters with a fixed value — they share the same properties (name, type, label) but their value is defined at creation, not instantiation. Difference: parameters are bound at instantiation, constants at definition. Conversion: a constant can be converted to a required parameter (and vice versa).*

A constant is a named immutable value bound at *definition* time. It shares the type, label, and constraint syntax of a parameter; what it does not share is the *binding* model. A parameter is asked of the caller; a constant is fixed by the author.

## 14.1 — Syntax

```
@@@const <NAME> : <type> = <value>, "<label>"   (with label)
@@@const <NAME> : <type> = <value>              (without label)
```

Rules (`§5.7`, §4.4.3):

- `<NAME>` — alphanumeric and underscore; convention is *uppercase* to distinguish constants from parameters at a glance. The convention is not enforced by the spec.
- `<type>` — required; one of the six standard parameter types (`text`, `textarea`, `number`, `date`, `boolean`, `select`).
- `<value>` — required; literal matching the type; for `select`, must be one of the declared options.
- `<label>` — optional; double-quoted human-readable description.

A constant may carry the same constraints (`min`, `max`, `minlen`, `maxlen`, `pattern`) as a parameter. The constraint is validated against the constant's own value; a violation is a CONSTRAINT_VIOLATION (the value is known at definition time, so the check is early).

Usage — substitution in the body — is identical to parameters:

```
@@@=<CONSTANT_NAME>
```

## 14.2 — Purpose

A prompt has values that change between callers and values that do not. The brand name, the company signature, the safety preamble, the default tone — these belong to the *author's* choices, not the caller's. POO²L gives those values a name and a place in the declaration list, separate from the parameters.

Three benefits over hard-coded literals:

1. **A single source of truth.** A constant declared once is substituted everywhere it appears. Changing the brand name in the constant's declaration changes every substitution site in the prompt — and in any child or include that uses the same constant by reference.

2. **An uncluttered parameter dialog.** A parameter shows up in the instantiation UI; a constant does not. The caller of the prompt is not bothered with values they are not supposed to change.

3. **Immutability as documentation.** A reader of the prompt sees `@@@const BRAND_VOICE : text = "..."` and understands at a glance that the value is fixed by the author. A `param` with a default could be overridden by the caller; a `const` could not.

The split between parameter and constant is a recurring authoring decision. The rule of thumb: *if two operators of the prompt would disagree about the value, it is a parameter; if every operator should use the same value, it is a constant*. → §XI introduced this rule in its anti-pattern; this chapter cements it.

## 14.3 — Possibilities

**Resolution at Phase 6.** Constants are resolved in Phase 6, after inheritance (Phase 3) and before parameter substitution (Phase 9). The order matters: a parent prompt may declare a constant that a child prompt's body substitutes; inheritance has already merged the parent's constant declarations into the child by Phase 3, and by Phase 9 the constant is already bound.

**Immutability is about *instantiation*, not redeclaration.** The immutability rule says a constant cannot be *bound at instantiation time* — the caller has no slot for it. An instantiation that tries to bind a constant's name raises IMMUTABILITY_VIOLATION. This is the ONLY thing IMMUTABILITY_VIOLATION catches.

**Inheritance and constants — a child MAY re-declare a parent's constant.** Constant-on-constant override IS allowed: a child that declares a `const` with the same name as the parent's simply *replaces* the parent's value during inheritance resolution (Phase 3). This is not an immutability violation — the value is still fixed at definition time, just fixed by the child instead of the parent.

(Binding a constant at *instantiation* is what IMMUTABILITY_VIOLATION forbids.) A value that must vary per *caller* is still a parameter, not a constant.

**Constants in includes.** An included prompt's constants are bound when the included prompt is rendered. An include with parameter overrides cannot override constants ; the override syntax targets parameters only.

**Types and constraints, same as parameters.** A constant declares its type ; an implementation should validate that the value matches. A constant may carry the same constraint keys as a parameter — the validation runs at Phase 6 instead of Phase 9.

**Conversion to and from parameters.** The spec notes ( `§4.4.3` , *Relationship to parameters*) : « Conversion : A constant can be converted to a required parameter (and vice versa) ». This is an authoring move — change `const` to `param` (and reverse) — not a runtime mechanism.

**Naming convention.** Uppercase identifiers for constants are universal in the IziBuddy reference library and recommended by the spec — but only as a *convention* : like every POO²L identifier, constant names are case- and diacritic-insensitive, so `BRAND` and `brand` are the SAME constant. Lowercase or mixed-case constant names are syntactically valid but visually collide with parameters ; a team that drops the uppercase convention loses the at-a-glance disambiguation.

**Visibility in tooling.** Implementations may display the constant's *label* in documentation or developer tooling ; the label has no semantic role at runtime. A constant without a label is referenced in tooling by its name.

## 14.4 — Related features and why

**Parameters (Chapter XI).** Constants and parameters are the two sides of the same coin. Together they cover every « named value a prompt uses » — the parameter for what the caller decides, the constant for what the author decides.

**Constraints (Chapter XII).** Constraints work on constants exactly as they do on parameters ; the validation phase shifts from Phase 9 to Phase 6 because the value is known at definition time.

**System variables (Chapter X).** A system variable is a third species of named value : provided by the *processor* at render time. A team that needs « the rendering user's name » reaches for `USER` ; « the company's name » is a constant ; « the customer's name (asked per render) » is a parameter.

**Inheritance (Chapter XIX).** A child inherits the parent's constants and MAY re-declare one with a new value (constant-on-constant override is allowed ; Phase 3 replaces). What no one may do is *bind* a constant at instantiation — that is IMMUTABILITY_VIOLATION. A value that varies per *caller* is a parameter, not a constant.

**Parameter sets (Chapter XV).** A parameter set binds parameters ; it cannot bind constants (they are not bindable). The two constructs do not interact.

**Lifecycle markers (Chapters XXV, XXVI).** Aliases and deprecation markers apply to templates, not to constants. A constant cannot be deprecated ; the prompt that declares the constant can be.

## 14.5 — Practical examples

**Trivial — a brand name fixed by the author.**

```
@@@const COMPANY_NAME : text = "Acme Corp", "Company name"
```

```
@@@section role
You are a customer-support representative for @@@=COMPANY_NAME.
```

The substitution site renders `Acme Corp`. No caller can change this; the brand name is part of the prompt's identity, not its configurable surface.

**Idiomatic — multiple constants, one body.**

```
@@@template press_release_drafter
@@@name Press-Release Drafter
@@@description Drafts a press release in the brand voice

@@@const COMPANY_NAME : text = "Acme Corp", "Company name"
@@@const BRAND_VOICE : text = "confident, plain-spoken", \
   "Brand voice description"
@@@const SIGNATURE : text = "— The Acme Communications Team", \
   "Press-release sign-off"

@@@param announcement : textarea, "Announcement summary"

@@@section task
Draft a press release for @@@=COMPANY_NAME in a @@@=BRAND_VOICE
voice. Summary of the announcement:

@@@=announcement

@@@section sign_off
@@@=SIGNATURE
```

The brand assets are constants; the announcement is the per-render parameter. The split keeps the dialog clean for the caller and documents the brand-voice decisions in the prompt itself.

**Idiomatic — a constant with a constraint.**

```
@@@const MAX_RETRIES : number min=1 max=10 = 3, \
   "Default retry budget for downstream API calls"
```

```
@@@section instructions
If a downstream call fails, retry up to @@@=MAX_RETRIES times.
```

The constant carries the same constraint syntax as a parameter. The value `3` is validated at Phase 6 against the `1..10` range. If a future edit raises the value to `15`, the parser rejects the change.

**Anti-pattern — a constant that should be a parameter.**

```
@@@const CUSTOMER_TIER : select("free", "pro", "enterprise") \
   = "pro", "Customer tier"

@@@section instructions
Treat the customer as @@@=CUSTOMER_TIER tier.
```

If two operators of the prompt expect to address customers in different tiers, the value is *not* a constant. The fix :

```
@@@param customer_tier : select("free", "pro", "enterprise") \
   = "pro", "Customer tier"

@@@section instructions
Treat the customer as @@@=customer_tier tier.
```

The next chapter covers *parameter sets* — the named bundles of parameter bindings that turn the « ask the user one value at a time » model into « ask the user which set of values ».

# Chapter XV — Parameter Sets

> **`standard.md §5.9` — Parameter Set Syntax (excerpt):**
>
> *Format: `@@@parameter-set <name>` … `<param_name>: <value>` … `@@@end-parameter-set`. Rules: `name` is alphanumeric plus underscore; one binding per line as `<name>: <value>`; values are strings (double-quoted), numbers, or booleans. RESTRICTION: `parameter-set` blocks are ONLY allowed in templates that do NOT belong to a collection. A prompt belongs to a collection when its `@@@template` is qualified (`<collection_name>.<template_id>`, §5.2); such a member prompt MUST NOT contain embedded `parameter-set` blocks (use collection-level parameter sets, declared in the `@@@collection` definition block per §5.10, instead).*
>
> **`standard.md §4.4.5` — Parameter Sets (excerpt):**
>
> *Parameter sets are named collections of parameter bindings enabling prompt instantiation with specific values. Multiple parameter sets per prompt; runtime selection of parameter set; each binding is type-checked against the target parameter's declared type and constraints (a mismatch raises TYPE_MISMATCH / CONSTRAINT_VIOLATION, and a binding naming an undeclared parameter raises UNDECLARED_PARAMETER_SET_BINDING). Rationale for the collection restriction: standalone templates keep parameter sets self-contained, while collection templates manage parameter sets at the collection level — this avoids conflicts between template-level and collection-level sets.*

A parameter set is a named bundle of parameter bindings. Where a parameter is « ask the user for one value », a parameter set is « ask the user which collection of values » ; the caller selects a set by name and every parameter the set covers is bound in one move.

## 15.1 — Syntax

```
@@@parameter-set <name>
<param_name>: <value>
<param_name>: <value>
...
@@@end-parameter-set
```

Rules ( `§5.9` ) :

- `<name>` — a Unicode letter then letters, digits, underscore (NO hyphen) ; case- and diacritic-insensitive.
- Inside the block : one binding per line, `<param_name>: <value>`.
- Values are strings (double-quoted), numbers, or booleans.
- The block is closed by the compound reserved keyword `@@@end-parameter-set` .

A prompt may declare multiple parameter sets. Each is a complete or partial collection of bindings — partial sets leave some parameters unbound, to be supplied at instantiation or defaulted.

The **restriction** is the central rule of this construct : a parameter-set declared *inside a prompt* is allowed only when the prompt does **not** belong to a collection. A prompt belongs to a collection when its `@@@template` header is qualified — `@@@template <collection_name>.<template_id>` . For such a member prompt, embedded `parameter-set` blocks are FORBIDDEN ; the collection's own `@@@collection` definition block ( `§5.10` ) holds the parameter sets instead.

## 15.2 — Purpose

A prompt with eight parameters is hard to call. A user staring at eight input fields cannot remember which combinations work ; a caller scripting the prompt has to type eight key-value pairs every time. Parameter sets solve both problems : name the combination.

```
@@@parameter-set HighValueOutreach
tier: "enterprise"
budget: 250000
tone: "consultative"
focus_area: "strategic-fit"
@@@end-parameter-set
```

The caller picks `HighValueOutreach` and the four parameters bind together. The author has captured the *combination's* intent in its name ; the eight-parameter prompt becomes as easy to call as a zero-parameter one.

Parameter sets also serve as *documentation by example*. A reader of the prompt sees the parameter declarations *and* the typical combinations the author had in mind. The « HighValueOutreach » set tells the next author : this is what we mean by a high-value outreach. The names are themselves a glossary.

## 15.3 — Possibilities

**Phase 5 binding.** Parameter sets are selected and bound at Phase 5 (Parameter Binding), after inheritance is resolved (Phase 3) and before per-parameter substitution at Phase 9. A parameter set applied at Phase 5 supplies *default* bindings for the parameters it covers ; an explicit instantiation-supplied binding for the same parameter wins over the set's binding.

The order — set first, then instantiation — means a caller may select a set *and* override one of its values :

```
 caller selects: HighValueOutreach
caller overrides: budget=500000
result: tier=enterprise, budget=500000, tone=consultative, focus_area
```

The selected set establishes the baseline; explicit instantiation values take priority where they overlap.

**Multiple sets per prompt.** A prompt may declare any number of parameter sets. The caller selects one (or none) per instantiation. Sets are independent of each other; declaring several does not implicitly chain them.

**Partial sets.** A set need not bind every parameter. Unbound parameters fall back to their per-parameter defaults (or remain unbound if no default exists, in which case Phase 9 raises UNDEFINED_PARAMETER unless the parameter is optional).

**The values must satisfy constraints.** A set whose value violates a parameter's declared constraint is rejected. The phase at which the rejection surfaces is implementation-defined within Phase 5 / Phase 9.

**Values follow simple literal rules.** Strings use double quotes; numbers are bare; booleans are `true` or `false`. The spec does not define a richer literal grammar — no arrays, no objects, no expressions. A value the language cannot literal is a value that does not belong in a parameter set.

**The collection restriction.** Embedded parameter sets are *forbidden* in templates that belong to a collection (a prompt whose `@@@template` header is qualified, `<collection_name>.<template_id>`). The reason is that collection-scoped parameter sets must be declared at the collection level — in the `@@@collection` definition block (`§5.10`) — so they can be inherited and merged through the collection chain (Phase 2). Allowing both template-embedded and collection-scoped sets would open conflicts the language has chosen to outlaw rather than resolve.

**Naming sets is part of the design.** A `parameter-set` named `Default`, `Set1`, `Set2` is a code smell; a set named `HighValueOutreach`, `BugReportSummary`, `MarketingPitch` documents the prompt's typical use cases. The names live in the prompt source and travel with the prompt; pick them as carefully as identifier names.

## 15.4 — Related features and why

**Parameters (Chapter XI).** Parameter sets exist to make parameters easier to use in bulk. A prompt with one parameter does not need a set; a prompt with five parameters benefits greatly from one.

**Constants (Chapter XIV).** A parameter set cannot bind a constant — constants are bound at definition time and cannot be rebound. A team migrating a constant to a parameter (because two operators disagreed about its value) often adds a parameter set per operator-class at the same time, to keep the caller experience as simple as it was when the value was constant.

**Constraints (Chapter XII).** A parameter set's bindings must satisfy the parameter's constraints. The validation runs at the end of Phase 5 or in Phase 9.

**Collections (Chapter XXII).** Collections declare their own parameter sets in the `@@@collection` definition block (`§5.10`). The collection-scope sets are inherited along the collection chain (Phase 2, `§4.6.1`) and apply to every prompt in the chain that has matching parameters. Embedded sets in collection-member templates are forbidden.

> ***`standard.md §4.6.1` — Template Collections (excerpt):***
>
> *A prompt collection is the OO² (horizontal-axis) entity of POO²L. Its methods are the templates that belong to it. A collection may inherit from another collection, declare abstract (virtual) prompt members that derived collections MUST imple-*

> *ment, and a prompt rendered inside a derived collection may reference the parent collection's templates via the collection-axis reference `^parent.<template_id>`. Collections NEST by containment (a collection may contain sub-collections, to unbounded depth); a member's qualified id has as many segments as the containment run requires, and is addressed with `.`. An entity is a collection precisely because a `@@@collection` block declares it; its membership, inheritance, and visibility are all expressed in-language — nothing about a collection's composition lives in an out-of-text sidecar.*

**Inheritance (Chapter XIX).** A prompt's embedded parameter sets ARE part of its L3 data layer, so they are inherited through the template-level OO chain : Phase 3 merges the parent's whole L3 layer — embedded parameter sets included — into the child. A child may add or re-declare sets by name ; the parent's unaffected sets remain available. (Collection-scope sets, declared in the `@@@collection` block, inherit along the separate collection chain in Phase 2.)

**Includes (Chapter XVIII).** An `include` cannot select a target's parameter set — only individual parameter overrides. To compose « render prompt X with set Y », the calling code must select set Y at instantiation time ; there is no in-source way to embed the set selection in an include.

## 15.5 — Practical examples

**Trivial — two sets on a parameterized prompt.**

```
@@@template email_drafter
@@@name Email Drafter
@@@description Drafts emails in different tones and lengths

@@@param recipient : text, "Recipient name"
@@@param tone : select("formal", "casual") = "formal", "Tone"
@@@param length : select("short", "medium", "long") = "medium", \
```

```
   "Length"

@@@parameter-set QuickFormal
tone: "formal"
length: "short"
@@@end-parameter-set

@@@parameter-set ChattyLong
tone: "casual"
length: "long"
@@@end-parameter-set

Draft an email to @@@=recipient in a @@@=tone tone,
about @@@=length in length.
```

The caller picks `QuickFormal` or `ChattyLong` and the two tone and length parameters bind together. The `recipient` parameter is not in either set; the caller must still supply it.

**Idiomatic — partial sets that override a default.**

```
@@@template account_outreach
@@@name Account Outreach
@@@description Outreach email parameterized by tier and budget

@@@param tier : select("free", "pro", "enterprise") = "pro", \
   "Customer tier"
@@@param budget : number min=0 max=1000000 = 5000, \
   "Account budget in USD"
@@@param tone : select("warm", "consultative", "transactional") \
   = "warm", "Tone of the email"

@@@parameter-set HighValueOutreach
tier: "enterprise"
budget: 250000
tone: "consultative"
@@@end-parameter-set

@@@parameter-set EarlyStage
```

```
tier: "free"
budget: 0
@@@end-parameter-set
```

`HighValueOutreach` binds three parameters; `EarlyStage` binds two and leaves `tone` to its default of `warm`. The caller of `EarlyStage` can still override `tone` at instantiation.

**Real-world — a parameter set whose values are documented by its name.**

```
@@@template bug_report_triage
@@@name Bug Report Triage
@@@description Triage a bug report into severity, owner area, and SLA

@@@param report : textarea, "Original report"
@@@param severity : select("p0", "p1", "p2", "p3") = "p2", \
   "Triaged severity"
@@@param owner_area : select("backend", "frontend", "ops", \
   "extension") = "backend", "Owning team area"
@@@param sla_days : number min=1 max=30 = 7, "SLA in days"

@@@parameter-set Hotfix
severity: "p0"
sla_days: 1
@@@end-parameter-set

@@@parameter-set Backlog
severity: "p3"
sla_days: 30
@@@end-parameter-set
```

Reading the set names — `Hotfix`, `Backlog` — communicates the triage intent as clearly as the bindings themselves.

**Anti-pattern — a parameter set inside a collection-member prompt.**

```
@@@template reviewers.code_review
@@@name Code Review
@@@description Reviews code

@@@param language : text = "Python", "Language"
@@@param focus : text = "general", "Focus"

@@@parameter-set PythonPerf
language: "Python"
focus: "performance"
@@@end-parameter-set
```

The prompt's qualified header `@@@template reviewers.code_review` makes it a member of the `reviewers` collection *and* it carries an embedded `parameter-set`. The combination is forbidden by `§5.9`, *Restriction*. The fix is to move the set to the `reviewers` collection's `@@@collection` definition block (`§5.10`), so the set lives alongside the other collection-scoped sets and inherits through the collection chain.

The next chapter covers *attachments* — file and binary content that travels with a prompt either fixed at definition or provided at instantiation.

# Chapter XVI — Attachments

***`standard.md §5.8` — Attachment Syntax (excerpt):***

*An attachment is described by TWO ORTHOGONAL axes. WHEN it is bound: `fixed` (bound at definition) or the TO-INCLUDE binding (provided at instantiation; required by default, optional with `?`). SOURCE — only for `fixed`, WHERE the content lives: `file` (path), `blob` (data URI), or `record` (external store ref). A `to-include` attachment carries NO source. Forms:*
*`@@@attachment <name> : <type> fixed file "<path>"`, `@@@attachment <name> : <type> fixed blob "data_uri"`, `@@@attachment <name> : <type> fixed record "<external_ref>"`, `@@@attachment <name> : <type> to-include, "<description>"`, with the optional variant `@@@attachment? <name> : <type> to-include, "<description>"`. Rules: `name` is alphanumeric plus underscore; `when` is one of `fixed` / `to-include`; `source` is one of `file` / `blob` / `record`, REQUIRED with `fixed` and FORBIDDEN with `to-include`; the value is a quoted string in every case (path for `file`, data URI for `blob`, external-store reference for `record`); the `?` prefix on `attachment` denotes optional attachments and is semantically meaningful only with `to-include`.*

***`standard.md §3.7` — Attachment (excerpt):***

*Attachments are file or binary content associated with templates, described by two orthogonal axes. WHEN it is bound: `fixed` (bound at definition time) or `to-include` (provided at instantiation time — required by default, optional with `?`). SOURCE (only for `fixed`; where the content lives): `file` (a filesystem path), `blob` (an inline data URI), or `record` (a reference to an external store). A `to-include` attachment carries*

> *NO source — the caller supplies the content at instantiation. Attachments SHOULD be declared at the END of the prompt. Inline binary uses Data URI scheme (RFC 2397)* `data:<media_type>;base64,<encoded_data>` *; line breaks in the base64 string are stripped during processing.*

An attachment is file or binary content associated with a prompt. It is the construct for « the prompt needs a PDF », « the prompt needs an image », « the prompt needs a CSV ». An attachment is described by two orthogonal axes : WHEN it is bound — `fixed` (bound at definition) or the to-include binding (provided at instantiation) — and, for `fixed` attachments only, the SOURCE where the content lives — `file`, `blob`, or `record`. The first axis is the choice between « the author already has the file » (`fixed`) and « the caller provides one » (`to-include`) ; the second axis, meaningful only for `fixed`, is *where* the bytes live.

## 16.1 — Syntax

Two axes, five forms (`§5.8`) :

```
@@@attachment <name>  : <type> fixed file   "<path>"
@@@attachment <name>  : <type> fixed blob   "data_uri"
@@@attachment <name>  : <type> fixed record "<external_ref>"
@@@attachment  <name> : <type> to-include, "<description>"
@@@attachment? <name> : <type> to-include, "<description>"
```

Components :

- `<name>` — alphanumeric and underscore. Names the attachment slot.
- `<type>` — MIME type or file extension.
- `when` — exactly one of `fixed`, `to-include`. The binding TIME. Required.

- `source` — exactly one of `file`, `blob`, `record`. WHERE the content lives. REQUIRED with `fixed`, FORBIDDEN with `to-include`.
- value — a quoted string; shape depends on the source (see below). Required with `fixed`; absent for `to-include` (the caller supplies the content at instantiation).
- `?` prefix on `attachment` — makes the attachment optional. Only meaningful with `to-include`.

Recommended position: at the end of the prompt body (`§4.4.4`, *Position in prompt*), so the prompt text is immediately visible and the attachment declarations do not obscure it.

The two axes in action:

```
@@@attachment manual  : pdf fixed file   "docs/manual.pdf"
@@@attachment logo    : png fixed blob   "
@@@attachment dataset : csv fixed record "sqlite:8f3a2b1c..."
@@@attachment input   : txt to-include, "Upload your input file"
```

- `fixed file` references a path resolved relative to the prompt's storage (implementation-defined).
- `fixed blob` embeds a data URI (RFC 2397): `data:<media_type>;base64,<encoded_data>`. The URI may be arbitrarily long; line breaks within the base64 payload are stripped during processing.
- `fixed record` is an external store reference. The implementation resolves it at render time; common forms are `sqlite:<hash>`, `fs:<hash>`, or an opaque `att_<id>`.
- `to-include` declares a slot the caller must fill at instantiation time and carries no source. The value is the user-facing description shown in the instantiation UI.

## 16.2 — Purpose

A prompt is text. A prompt that needs a *non-text* input — a file the model must read alongside the prose — needs a way to declare that input. Attachments are that way.

The WHEN axis splits attachments into two groups:

- **`fixed`** (source `file` / `blob` / `record`) — the file is part of the prompt definition. The brand style guide PDF, the company logo, the reference data the prompt always uses. Every render of this prompt sees the same file. The three SOURCE options differ only in *where* the bytes live (filesystem, inline in the prompt JSON, or external store), not in *whose* responsibility they are.
- **`to-include`** — the file is supplied at instantiation time. The customer's code, the user's photo, the document under review. Every render of this prompt receives a different file. A `to-include` attachment carries no source.

The `fixed` group and `to-include` solve different problems and rarely substitute for each other.

## 16.3 — Possibilities

**Phase 7 binding.** Attachments are bound in Phase 7 (Attachment Binding), after constant resolution (Phase 6) and before include resolution (Phase 8). `fixed file` attachments are validated for path existence, `fixed record` for resolvability against the store; a `fixed blob`'s data-URI well-formedness is a static check at Phase 1. Required `to-include` attachments must be provided at instantiation, otherwise ATTACHMENT_REQUIRED. Optional `to-include` attachments may be omitted; the attachment slot is simply absent in the rendered output.

**The `?` prefix only makes sense with `to-include`.** A `fixed` attachment cannot meaningfully be optional — if the content is bound by the author, the attachment is part of the definition. Optionality applies only when the caller is the source of the content, which is to say only on the `to-include` binding time.

**Inline data URIs.** `fixed blob` accepts a data URI as the source. The URI scheme is RFC 2397 : `data:<media_type>;base64,<encoded_data>`. Implementations strip the line breaks inside the encoded data so authors may format the long string across multiple lines.

**The `<type>` slot.** The grammar carries a `<type>` slot (MIME type or file extension) between the name and the binding-time keyword. It declares the expected content type up front, regardless of axis : a `fixed file` attachment declares the type of the bound file, a `to-include` attachment declares the type the caller is expected to provide.

**Names are slot names, not file names.** The attachment's `<name>` identifies the *slot* the file fills, not the file's filename. A `to-include` attachment named `user_code` may receive a file actually named `module_a.py` at instantiation ; the prompt references the slot, not the filename.

**Positional convention.** Attachments at the end keeps the prompt's prose readable. The processor accepts attachments anywhere in the body, but readers benefit from the convention. Implementations may surface a « this attachment is declared mid- prose » warning during authoring.

**Sizes.** The language imposes no size limit. Implementations do, both for ergonomics and for the underlying model's input budget. A 100 MB CSV is syntactically a valid attachment ; practically it is unlikely to render.

**Storage backend.** The bytes referenced by `fixed blob` and `fixed record` are stored according to the implementation's selected backend. Common choices are : inline base-64 in the prompt

JSON (portable, bloats files), content-addressed SQLite with SHA-256 dedup, or content-addressed files on disk. The backend is an implementation concern — the grammar is the same regardless.

## 16.4 — Related features and why

**Parameters and constants (Chapters 11, 13).** Attachments are the *binary* siblings of these constructs. A parameter is for text the caller provides; a `to-include` attachment is for a file the caller provides. A constant is for text the author fixes; a `fixed` attachment (`file` / `blob` / `record`) is for a file the author fixes.

**Sections (Chapter VII).** A section's prose may refer to an attachment by name. The reference is informational — the attachment is delivered separately; the section's text simply tells the model what to do with it.

**Inheritance (Chapter XIX).** A child prompt inherits the parent's attachments. The child may declare an attachment that *overrides* a parent's attachment — for example, the parent declares a `to-include` attachment and the child provides a concrete `fixed file` or `fixed blob` one. The override mechanism is the same one parameters use.

**Virtual templates (Chapter XXIII).** A virtual prompt's contract includes its attachments. A concrete override must provide attachments for every required attachment the virtual declared, either as `fixed` with a `file` / `blob` / `record` source (filling the slot) or as `to-include` (passing the requirement through to the caller).

**Includes (Chapter XVII).** An `@@@{include }` does not transfer attachments by default — the included prompt's attachments belong to the included prompt's render, not to the including prompt's. Some implementations offer parameter overrides on includes; none extend the override syntax to attachments.

**Collections (Chapter XXII).** Collection-scoped attachments do not exist in v1.0 of POO²L. Attachments live on individual templates. A team that needs a « team-wide style guide PDF » attaches it to the relevant prompt and includes that prompt where needed.

## 16.5 — Practical examples

**Trivial — a single `fixed file` attachment.**

```
@@@template style_guide_reviewer
@@@name Style-Guide Reviewer
@@@description Reviews copy against the company style guide

@@@param copy_text : textarea, "Copy to review"

@@@section task
Review the copy against the attached style guide.

The copy:

@@@=copy_text

@@@attachment style_guide : pdf fixed file "docs/style-guide.pdf"
```

The style guide is the same file for every render. The reviewer parameter receives the copy under review.

**Idiomatic — required and optional `to-include` attachments.**

```
@@@template code_reviewer
@@@name Code Reviewer
@@@description Reviews user-supplied code with optional context

@@@param language : text = "Python", "Language"

@@@section task
Review the attached code. Use the optional context if provided.
```

```
@@@attachment user_code : py to-include, "Source file to review"
@@@attachment? context : txt to-include, "Optional readme or design doc
```

The caller must supply `user_code`; `context` is optional. A render without `user_code` raises ATTACHMENT_REQUIRED.

**Real-world — a `fixed blob` inline image used as a brand asset.**

```
 @@@template social_post_drafter
@@@name Social-Post Drafter
@@@description Drafts a social post with the brand logo attached

@@@param announcement : textarea, "Announcement text"

@@@section task
Draft a social-media post announcing the following:

@@@=announcement

Include the brand logo at the top of the post.

@@@attachment logo : png fixed blob "
AAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DH
   xgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg=="
```

The logo is inline as a data URI; the prompt has no external dependency on a file path.

**Storage-backed — a `fixed record` attachment for a large dataset.**

```
 @@@template quarterly_sales_analyst
@@@name Quarterly Sales Analyst
@@@description Q&A over the company's quarterly sales dataset

@@@param question : textarea, "Question about the data"
```

```
@@@section task
Answer the following question using the attached dataset.

@@@=question

@@@attachment dataset : csv fixed record "sqlite:8f3a2b1c0e5d4f6a..."
```

The 200 MB CSV does not bloat the prompt JSON ; the directive holds only the 70-byte ref, and the implementation's storage backend serves the bytes at render time.

**Anti-pattern — using an attachment for a small text value.**

```
@@@attachment tagline : txt fixed blob "data:text/plain;base64,VGhlI
```

The decoded text is a one-line tagline (« The future, rendered. ») that would be far more naturally written as a constant :

```
@@@const TAGLINE : text = "The future, rendered.", "Brand tagline"
```

Attachments are for content that does not fit in a `text` or `textarea` value — binary, multi-page, or natively non-text. For short prose, reach for a constant or parameter.

The next chapter — and the first of Part IV — covers *includes*, the construct that turns the language from one-template to many-template.

# Partie 4

*Composition (multi-template scope)*

# Chapter XVII — Includes

> ***`standard.md §5.4` — Include Syntax (excerpt):***
>
> *Line form: `@@@include <reference_path>`. Transparent line form: `@@@include (<reference_path>)` — the parentheses wrap the WHOLE reference, arguments included, and ask for the target to render with NO heading of its own. With parameters: `@@@include <reference_path>(<param>=<value>, ...)`. Inline (mid-text) form: `@@@{include <reference> ...}`. A reference layers two separators by the NATURE of what they join: `/` traverses FOLDERS to REACH a POO²L entity (pure location, no compositional meaning); `.` then qualifies the POO²L entity chain — namespace (library) → collection → sub-collection(s) → prompt → section → sub-section. Collections NEST, so the collection run has UNBOUNDED depth. There is NO bracket `[N]` indexing; an ordered item is a dotted **sub-section** (`prompt.section.sub`). Every identifier is case- and diacritic-insensitive. Parameterized includes pass values to the target — only parameters DECLARED in the target can be passed (passing an undeclared one is UNDECLARED_INCLUDE_PARAMETER, a semantic error detected at PHASE 8); values may be quoted strings, numbers, or booleans; a symbolic value is itself a substitution written `p=@@@=caller_param`.*
>
> ***`standard.md §4.3` — Level 2: Procedural (excerpt):***
>
> *At L2, templates support modular composition through external references, enabling reusability and DRY principles. Required elements: include mechanism, reference resolution, namespace or path specification. Characteristics: content reusability across templates, single source of truth for shared components, hierarchical organization of prompt libraries.*

The include is the simplest mechanism for *one prompt to reference another*. The move from L1 to L2 is the move from « named sections in one prompt » to « named templates that other templates can pull in by reference ». With includes alone, a team can replace duplication with reuse and turn a folder of templates into a *library*.

## 17.1 — Syntax

The basic form :

```
@@@include <reference_path>
```

Rules ( `§5.4` ) :

- `<reference_path>` — an optional `/`-separated folder path followed by a *qualified handle* whose POO²L-entity segments are separated by `.` (case- and diacritic-insensitive ; each segment starts with a Unicode letter, then letters / digits / `_` only — NO hyphen).
- A trailing `.<sub>` selects a **sub-section** of the referenced section ; there is no bracket `[N]` indexing.
- A handle may be bare ( `welcome` ) or qualified by collection ( `marketing.welcome` , and — since collections nest — deeper runs like `marketing.campaigns.welcome` ). Bare handles resolve through the cascade described in `§6.1` (PHASE 8) ; qualified handles pin the lookup to a specific collection.
- Template-axis reference forms — `self.<section>` , `parent` , `parent.<section>` — are available when the including prompt is in an inheritance chain. The collection-axis forms carry the `^` prefix — `^self.<template_id>` , `^parent.<template_id>` , `^parent.<template_id>.<section>` — and reach members of the current/parent collection. See Chapter XX.

The forms that pass parameter overrides — `@@@{include x(p="v")}` — are Chapter XVIII.

## 17.2 — Asking for a transparent render

`@@@section (<name>)` lets the AUTHOR of a section ship it without a heading. `@@@include (<reference>)` is the same request made from the other end, by whoever consumes it — for a target whose author did not declare it, and without editing that author's document.

```
@@@include contract.preamble           -- ## Preamble, then the body
@@@include (contract.preamble)         -- the body alone
@@@include (contract(lang=en).preamble)  -- arguments live INSIDE the
```

**The parentheses wrap the WHOLE reference, arguments included.** The arguments name WHICH instance is being included, so they are part of what is referenced — unlike a section's `: description`, which is a separate clause of the declaration and stays outside. `()` is the only family that marks transparency ; `[…]`, `{…}` and `<…>` are the equivalent families of the BOUNDED form ( `standard.md §3.17` ), which this is not.

**The two marks are ONE mark.** The transparency of a render is the LOGICAL OR of what the provider declared and what the consumer asked, so these three pairings render byte-identical text and asking twice is asking once :

| provider declares | consumer writes |
|---|---|
| `@@@section (preamble)` | `@@@include contract.preamble` |
| `@@@section preamble` | `@@@include (contract.preamble)` |
| `@@@section (preamble)` | `@@@include (contract.preamble)` |

NAMING the section at an include restores nothing — only a DESCENDANT redeclaring it without the parentheses brings the title back. A reference reads ; it does not redeclare.

The mark holds wherever an include resolves — a plain handle, `parent.…`, `self.…`, `^self.…` alike — and it is INERT where there is no heading of its own to withhold : on a target already declared transparent, and on material that MERGES into the host's list of sub-sections, whose headings belong to the host and never to the target.

On a WHOLE TEMPLATE it is refused : `TRANSPARENT_INCLUDE_ON_TEMPLATE`. An included prompt emits no heading of its own to begin with, so the mark would either say nothing or silently mean « make every one of its sections transparent » — a wider request, written section by section or not at all.

## 17.3 — Purpose

A prompt without includes is *self-contained* : every paragraph it needs lives inside its own body. A prompt with includes is a *composer* : it assembles its body from named fragments living elsewhere.

The benefits are the same benefits any modular system gives a codebase :

1. **Single source of truth.** The shared safety preamble lives in one prompt. Editing that prompt edits every render of every caller, in one move.

2. **Smaller surfaces to read.** A 200-line prompt assembled from eight 25-line included fragments is easier to navigate than the same 200 lines in one file.

3. **Independent evolution.** The glossary block can be edited without touching the templates that include it. The templates that include it inherit the edit automatically.

The cost is that the rendered behaviour now depends on templates the reader of one file does not see. Tooling helps — IziBuddy's inheritance-chain viewer renders the include graph — but the cost is real and is the reason L2 is the level where a *library* (as opposed to a folder) starts to make sense.

## 17.4 — Possibilities

**Phase 8 resolution.** Includes resolve in Phase 8, after inheritance (Phase 3) but before parameter substitution (Phase 9). The included prompt's body is substituted into the including prompt's body in place of the `@@@include` directive.

**Cycle detection.** An include that, directly or transitively, references the including prompt is CIRCULAR_INCLUDE. The processor detects cycles by tracking the include stack during Phase 8.

**Alias resolution.** Includes that name an alias are registered in Phase 1 and resolved to the canonical id during segmentation at Phase 8; the canonical id is what gets included.

**Section and sub-section selection.** A path of the form `<prompt>.<section>` selects a named section of the referenced prompt; a further `.<sub>` (`<prompt>.<section>.<sub>`) selects one of its sub-sections. There is no bracket `[N]` indexing — ordered items are named sub-sections (`steps.one`, `steps.two`).

**Lookup cascade (`§6.1` PHASE 8).** A bare handle resolves through an ordered cascade: qualified handle (8.a) → inheritance chain of the caller's collection (8.b) → root, the templates that belong to no collection (8.c). The first non-empty match wins, silently; there is no global by-id fallback beyond root. Ambiguity at a single stage is `AMBIGUOUS_INCLUDE`; qualifying the handle disambiguates.

**The included prompt's own constructs apply.** The included prompt has its own parameters, constants, sections, and attachments. The including prompt does not see those declarations directly — the included prompt is rendered as a self-contained unit, and its rendered output is the text spliced into the including prompt. Parameters can flow *into* the included prompt via the override form (Chapter XVIII); they do not flow *out*.

**Include is not inheritance.** An include pulls *rendered text*; inheritance pulls *structure and contract*. The difference is explored in detail in Chapter XIX. The short version: include is reuse of *content*; inherit is reuse of *form*.

**Verbatim suspends include resolution.** An `@@@{include }` directive inside a `@@@verbatim` block (Chapter IX) is not resolved. The literal directive renders in the output.

**No control flow.** An include is unconditional. The processor includes the referenced prompt every time. A library that needs « sometimes include A, sometimes include B » must produce two templates — one that includes A, one that includes B — because the language has no `if`. Selecting between them belongs to the caller.

## 17.5 — Related features and why

**Includes with parameter overrides (Chapter XVIII).** The same keyword (`include`) accepts a parameter override list at the call site, letting the includer bind the includee's parameters. The override syntax is the only way to influence the includee's substitution from the include site.

**Inheritance (Chapter XIX).** Includes and inheritance are related but distinct. An inherits relationship is *vertical* — the child *is a* parent. An include relationship is *horizontal* — the includer *uses a fragment of* the included. Choosing between them is the most common authoring decision in Part IV.

**`parent` and `self` references (Chapter XX).** When the including prompt is in an inheritance chain, an `@@@{include parent}` or `@@@{include parent.<section>}` walks the chain. The `self.<section>` form retrieves the most specialized version of a named section across the chain. These forms compose with includes naturally; they are includes whose target is « the parent » rather than a named prompt.

**Sections (Chapter VII).** A path of the form `<prompt>.<section>` selects a *section* of the referenced prompt. Without sections, includes can only pull entire prompt bodies; the section-selection form turns the include into a fine-grained reference.

**Aliases (Chapter XXV).** An include may name an alias. The processor registers the alias in Phase 1 and resolves it during segmentation at Phase 8; the canonical id is what gets included.

**Deprecation (Chapter XXVI).** An include of a deprecated prompt emits a DEPRECATED_TEMPLATE warning. The include still resolves; the warning is informational.

## 17.6 — Practical examples

**Trivial — a shared safety preamble.**

```
@@@template customer_reply
@@@name Customer Reply
@@@description Customer-support reply with safety preamble

@@@section (preamble)
@@@include common/safety_preamble

@@@section role
You are a customer-support agent.

@@@section task
Reply to the message below.
```

The safety preamble lives in its own prompt, `common/safety_preamble`. A change to that file changes the preamble in every prompt that includes it.

**Idiomatic — a glossary block included into several callers.**

```
@@@template acronym_reviewer
@@@name Acronym Reviewer
@@@description Reviews copy for unexplained acronyms

@@@section role
You are a copy editor.

@@@include glossary/company_acronyms

@@@section task
Identify any acronym in the attached text that does not appear
in the glossary above.
```

The glossary is a fact about the company; it does not belong to any single prompt. Storing it as its own prompt, includable from several callers, is the L2 pattern.

**Real-world — section-selection from a multi-section prompt.**

```
@@@template onboarding_email
@@@name Onboarding Email
@@@description Onboarding email for new customers

@@@section opener
@@@include templates/email.opener

@@@section body
Welcome to Acme. The next few days will...

@@@section closer
@@@include templates/email.closer
```

The `templates/email` prompt declares an `opener` and `closer` section; the onboarding email reuses *just those sections*, leaving the body unique to itself.

**Anti-pattern — using include where inheritance was needed.**

```
@@@template python_code_reviewer
@@@name Python Code Reviewer
@@@description Reviews Python code

@@@section (role_from_reviewer)
@@@include code_reviewer.role
@@@include code_reviewer.format

@@@section task
Review the attached Python code with emphasis on type hints
and PEP-8 conformance.
```

The author has manually copied two sections from `code_reviewer` because they wanted to inherit *everything* but override only `task`. The construct for « inherit everything, override one section » is inheritance, not a list of includes. The right shape :

```
@@@template python_code_reviewer inherits code_reviewer
@@@name Python Code Reviewer
@@@description Reviews Python code

@@@section task
Review the attached Python code with emphasis on type hints
and PEP-8 conformance.
```

The next chapter covers the parameter-override form of the same keyword : `@@@{include x(p="v")}`.

# Chapter XVIII — Includes with Parameter Overrides

> ***`standard.md §5.4` — Parameterized includes (excerpt):***
>
> *Includes MAY pass parameter values to the referenced prompt using the form `@@@{include <reference>(<param1>=<value1>, ...)}`. Only parameters DECLARED in the target prompt can be passed; passing an undeclared parameter is an `UNDECLARED_INCLUDE_PARAMETER` error — a SEMANTIC binding error (the construct parses fine), detected during include resolution (§6.1, PHASE 8) once the target is resolved, with a Damerau-Levenshtein «did you mean» suggestion against the target's declared names. Values may be quoted strings, numbers, or booleans, and parameter names must match exactly (case-sensitive). All `self` and `parent` forms accept the same parenthesized bindings and raise the SAME error for a parameter the target does not declare.*

The plain `@@@{include path}` of Chapter XVII pulls the included prompt's *current* rendering — whatever the included prompt produces with its own default parameter bindings. The parameterized form lets the *includer* bind the *includee's* parameters at the include site, producing a different rendering of the same included prompt.

## 18.1 — Syntax

```
@@@include <reference_path>(<param1>=<value1>, <param2>=<value2>, ...)
```

Rules (`§5.4`, *Parameterized includes*):

- The override list is a parenthesized, comma-separated sequence of `name=value` bindings.
- Only parameters declared by the *target* prompt may be passed. Passing an undeclared parameter is an `UNDECLARED_INCLUDE_PARAMETER` error (a semantic binding error, not malformed syntax; Chapter XXXI) — and the inheritance-aware `self.` / `parent.` forms raise the same error, they do not silently ignore it.
- Parameter names are matched case- and diacritic-insensitively (like every identifier).
- Values follow the same literal rules as parameter sets (Chapter XV): double-quoted strings, bare numbers, bare booleans. A *symbolic* value — passing one of the caller's own parameters through — is itself a substitution and must be written `name=@@@=caller_param`, never a bare `name=caller_param`.

The override syntax extends to the `self`/`parent` family too (Chapter XX); the same parenthesized binding list applies.

## 18.2 — Purpose

A library prompt is most useful when it accepts parameters. An includer who wants to *call* the library prompt with specific values — without binding those values for every other includer — needs a way to pass them at the include site. The parameterized include is that way.

```
@@@include templates/greeting(name="John", formal=true)
```

The `templates/greeting` prompt declares the `name` and `formal` parameters. The include passes the values at the call site; the greeting renders as if the caller had supplied those values at instantiation. Twelve calls with twelve different names produce twelve different renderings of the same prompt — without storing twelve variants.

Think of the parameterized include as a *function call*. The included prompt is the function; its parameters are the function's signature; the override list is the argument tuple. POO²L lacks the broader function-call machinery (no return values beyond the rendered text, no scoping rules richer than substitution), but within those limits, the include-with-overrides is the language's function call.

## 18.3 — Possibilities

**Partial bindings.** The override list may bind a subset of the target's parameters. Unbound parameters fall back to their declared defaults; if a required parameter has no default and the override list does not bind it, Phase 9 raises UNDEFINED_PARAMETER on the *included* render.

**Type and constraint validation.** The values in the override list must satisfy the target parameter's type and constraints. Phase 9 validates as it would for any other binding source.

**Aliases work.** Overrides on an include that names an alias target are passed to the canonical prompt. Phase 1 has already registered the alias; Phase 8 resolves the canonical reference and binds the parameters declared on the canonical prompt.

**Includes are still includes.** Phase 8 substitutes the included prompt's *rendered* output into the includer's body. The override affects only the included render; the includer's own parameters are not influenced.

**Scope of the bindings.** The override binds the includee's parameters for *that include only*. Two consecutive includes of the same target with different override lists produce two different renderings; the bindings do not leak between them.

**Combination with `parent` / `self`.** The `parent` and `self` family (Chapter XX) accepts the same parenthesized override syntax. A common pattern at L3+:

```
@@@include parent(lang="fr")
```

The parent prompt is rendered with `lang="fr"`, overriding the parent's own default. The override applies to the parent rendering only ; the current prompt's parameters are unaffected.

**No referent in the includee.** The override binds *the includee's parameters*. There is no way to reference the includer's parameters from inside the includee — the includee is a self- contained prompt with its own declaration space.

**Combination with parameter sets.** An include cannot select a parameter set on the includee — only individual parameter overrides. If a librarian wants a « the includee is rendered with parameter set X » semantics, the right shape is two layers : the includee declares no sets, and a *wrapper* prompt declares the set and includes the inner prompt with the right override list.

**Verbatim suspends overrides too.** Includes inside a verbatim block render literally, including their override lists.

## 18.4 — Related features and why

**Includes (Chapter XVII).** The basic include without overrides is a special case of this chapter's construct with an empty override list. The same Phase 8 resolution applies.

**Parameters (Chapter XI).** The target's parameter declarations are what the override list binds against. A target without parameters cannot be called with overrides — the override list must be empty, and the parenthesized syntax is conventionally omitted in that case.

**Parameter sets (Chapter XV).** Parameter sets and overrides are both bulk-binding mechanisms. The set lives in the target's source and is named ; the override lives at the call site and is anonymous. Use a set when the same combination is reused ; use overrides when each call site has a unique combination.

**`self` and `parent` (Chapter XX).** Both forms accept override lists in the same parenthesized syntax. `@@@{include parent(p="v")}` renders the parent with the override binding; analogous for the section forms.

**Constraints (Chapter XII).** Constraints on the target's parameters apply to override values. A value that violates a constraint is rejected with CONSTRAINT_VIOLATION.

**Inheritance (Chapter XIX).** Inheritance pulls the parent's contract; an include with overrides invokes a callable target. The two compose: a child may *inherit* a parent and *include* a sibling library prompt with overrides.

## 18.5 — Practical examples

**Trivial — calling a library greeting with a name.**

```
@@@template welcome_email
@@@name Welcome Email
@@@description Sends a welcome email to a named recipient

@@@param recipient_name : text, "Recipient name"

@@@section greeting
@@@include templates/greeting(name=@@@=recipient_name, formal=false)

@@@section body
Welcome to Acme.
```

The welcome email includes the library `templates/greeting` and passes its own `recipient_name` parameter through as the greeting's `name`. The greeting prompt does not see the welcome email's other parameters; it sees only `name` and `formal`.

**Idiomatic — three call sites, three combinations of the same template.**

```
@@@template quarterly_newsletter
@@@name Quarterly Newsletter
@@@description Composes a quarterly newsletter from three blocks

@@@section opener
@@@include blocks/seasonal_opener(season="winter", year=2027)

@@@section feature
@@@include blocks/feature_story(topic="api-v3", length=500)

@@@section closer
@@@include blocks/standard_closer(signoff="The Acme Team")
```

The same three library templates (`seasonal_opener`, `feature_story`, `standard_closer`) are reused across newsletters; each include passes the per-issue values.

**Real-world — propagating a locale parameter through an included library.**

```
@@@template localized_reviewer
@@@name Localized Reviewer
@@@description Reviewer that includes locale-aware checklists

@@@param lang : select("en", "fr", "de") = "en", "Locale"

@@@section role
You are a reviewer.

@@@section checklist
@@@include shared/legal_checklist(lang=@@@=lang)
@@@include shared/style_checklist(lang=@@@=lang)
```

The reviewer's `lang` parameter is passed through to both included library templates. Both library templates render in the chosen locale; the same review prompt produces three different checklists depending on the caller's choice.

**Anti-pattern — overriding a parameter the target does not declare.**

```
@@@include templates/greeting(undefined_param="x")
```

The target `templates/greeting` does not declare `undefined_param`. Include resolution (Phase 8) raises `UNDECLARED_INCLUDE_PARAMETER` (« `undefined_param` is not declared in `templates/greeting` »), typically with a « did you mean » suggestion. The fix is to read the target's declarations and override only the parameters that exist.

**Anti-pattern — using an override where a wrapping prompt was needed.**

```
@@@section opener
@@@include templates/greeting(name="John")

@@@section body_one
@@@include templates/greeting(name="Jane")

@@@section body_two
@@@include templates/greeting(name="Mike")
```

Three includes of the same target with three different names. That is exactly what the override is for, and the example is *not* an anti-pattern in itself. It *becomes* one when the names are all *the same value as the includer's parameters* — at which point the right shape is to declare the names as parameters on the includer and pass them through symbolically, as in the « Localized Reviewer » idiomatic example above.

The next chapter covers *inheritance* — the construct that specializes templates rather than reusing fragments.

# Chapter XIX — Inheritance

> ***`standard.md §5.5` — Inheritance Syntax (excerpt) :***
>
> *Inheritance is declared by an inline clause on the entity-opening header: `@@@template <id> inherits <parent_template_reference>`. There is NO standalone `@@@inherits` directive — written as a bare directive (`@@@inherits …`) it matches no line-directive production and is rejected as INVALID_SYNTAX. Only one parent per prompt — single inheritance. The `inherits` reference follows the same namespace rules as includes.*
>
> ***`standard.md §4.5` — Level 4 : Object-Oriented (excerpt) :***
>
> *Templates support inheritance, enabling specialization and derivation of prompt families. This is the first Object-Oriented axis (OO #1). A child prompt inherits all sections from its parent (and the parent's L3 data layer), can override parent sections, and can add new sections. Multiple levels of inheritance are supported. Circular inheritance is prohibited.*

Inheritance is POO²L's first object-oriented construct. A prompt whose header carries `inherits parent_id` — `@@@template child inherits parent_id` — is a *specialization* of the parent : it shares the parent's identity-and-content surface, may override individual sections, may add new sections, and reports its own identity to the rest of the system. Where an include reuses *content*, inheritance reuses *form*.

## 19.1 — Syntax

```
@@@template <id> inherits <parent_template_reference>
```

Inheritance rides on the `@@@template` header as an inline `inherits` clause — there is no separate `@@@inherits` directive (a bare `@@@inherits …` is INVALID_SYNTAX).

Rules (`§5.5`):

- Single inheritance only: at most one `inherits` clause per prompt header.
- `<parent_template_reference>` follows the same path conventions as an include's reference path; aliases work transparently (an alias used as the parent reference resolves to the canonical id as inheritance is resolved, at PHASE 3 — identically to using the canonical id directly).
- The clause lives on the entity-opening `@@@template` header, which MUST come first in the prompt; the parent reference is therefore declared ahead of any section definitions.
- Multiple *levels* of inheritance are supported. A → B → C is valid; the chain is walked root-first during PHASE 3.
- Circular inheritance is forbidden (CIRCULAR_INHERITANCE).

## 19.2 — Purpose

A library of templates often has a *family* — a base pattern shared across several variants. A `code_reviewer` base prompt describes how to review code; the variants — `python_reviewer`, `go_reviewer`, `prose_reviewer` — describe how to review a specific medium. Without inheritance, the base pattern must be copied into every variant or pieced together with includes. With inheritance, each variant declares its parentage and overrides only what differs.

Three benefits:

1. **Reuse of contract.** The parent's parameters, constants, and sections become the child's. A child does not need to redeclare them; it gets them for free.

2. **Specialization by override.** The child overrides parent sections by name. The overriding section replaces the parent's content for that slot ; new sections in the child are appended.

3. **A type-like relationship.** The child *is a* parent. Anywhere the system reasons about templates of a parent's family, the child is a member of that family. Collections (Chapter XXIII) exploit this when treating the family as a polymorphic group.

The cost is the cost OO charges everywhere : a reader of the child cannot understand the child without the parent. The hierarchy must be readable ; a chain four deep is at the edge of comprehensibility.

## 19.3 — Possibilities

**PHASE 3 resolution.** Inheritance resolves in PHASE 3. The processor walks the chain root-first : starting from the topmost ancestor, then each descendant in turn, merging sections by name. A descendant section with the same name as an ancestor's replaces the ancestor's content ; new descendant sections are appended.

**Single inheritance.** A prompt may declare at most one `inherits`. POO²L does not support multiple inheritance — the language is committed to determinism and resolution simplicity, and multiple inheritance is famously complicated to resolve. Where a child needs structure from two unrelated sources, the right shape is *include* (Chapter XVII), not multiple inheritance.

**Cycle detection.** A cycle in the chain (A → B → A, or A → B → C → A) is CIRCULAR_INHERITANCE. The processor builds the chain by following each ancestor's `inherits` ; the first time it revisits a prompt, the cycle is detected.

**Identity stays the child's.** A child inherits the parent's *content* and *contract*, not its *identity*. The child's `name`, `id`, and `description` are the child's own. The processor never treats the child as if it were the parent at the identity level.

**Parameter and constant inheritance.** A child inherits the parent's parameter and constant declarations. The child may add its own parameters and constants; it may re-declare a parent's parameter to change its default (the type cannot change), but it cannot remove a parent's parameter — the parent's body presumably substitutes it. A child MAY also re-declare a parent's *constant* with a new value: constant-on-constant override is allowed, and Phase 3 simply replaces the parent's value. (What IMMUTABILITY_VIOLATION forbids is *binding* a constant at instantiation — not a child re-declaring it.)

**Section overrides.** A child section with the same name as a parent's section *replaces* the parent's content for that section. A child section with a new name is *appended* in declaration order (or positioned by an explicit `first` / `last` / `before` / `after` placement clause). Sub-sections (`steps.one`) are matched by their dotted name.

**Sealed sections (`final` section).** A parent may declare a section as `@@@section final <name> ...` to forbid any descendant from redeclaring it. Any attempt to override a final section — even just to change its presence toggle (`?` / `!`) or its description, with no body change — raises OVERRIDE_OF_FINAL_SECTION at PHASE 3. The complementary modifier `@@@section virtual <name>` declares an abstract slot the descendant MUST implement; both modifiers are covered in detail in Chapter VII.

**Attachments are inherited.** The parent's attachments — both binding times (`fixed`, with a `file` / `blob` / `record` source, and `to-include`) — appear in the child unless the child declares an attachment with the same name (which overrides the parent's, in the same way sections override).

**`parent` and `self` references.** Inside the child's body, the parent's content is reachable through `parent` (full parent content) and `parent.<section>` (named parent section). The `self` form retrieves the most specialized version of a section across the chain. Chapter XX is the full treatment.

**Aliases work for the parent reference.** An `inherits` clause that names an alias is resolved to the canonical id as inheritance is resolved (PHASE 3). The inheritance link is to the canonical prompt regardless of how it was named at the call site.

**Deprecation warns through inheritance.** If the parent is declared deprecated, every child triggers a DEPRECATED_TEMPLATE warning at render. The warning encourages migration without breaking the rendered output.

## 19.4 — Related features and why

**Includes (Chapter XVII).** Include reuses *content*; inheritance reuses *form*. The two are not interchangeable. The rule of thumb: if the child's behaviour is *the same kind of thing*, only specialized, use inheritance. If the child uses some of the target's *output* and otherwise has its own behaviour, use include.

**`parent` and `self` (Chapter XX).** Inheritance and these references are inseparable. `parent` and `self.<section>` only make sense when the prompt is in an inheritance chain.

**Section composition semantics (Chapter XXIV).** The rules that determine whether a child section *overrides*, *extends*, or *merges* a parent's section are formalized in Chapter XXIV. The short version: replacing without `parent` is *override*; using `parent.<section>` at the start of the child's section is *extend*; using it in the middle is *merge*.

**Virtual and final sections.** A section may carry a `virtual` or `final` modifier (Chapter VII). A `virtual` section is an abstract slot the descendant MUST implement (otherwise UNIMPLEMENTED_VIRTUAL_SECTION at PHASE 3); a `final` section is sealed and MUST NOT be redeclared (otherwise OVERRIDE_OF_FINAL_SECTION at PHASE 3). The two modifiers are mutually exclusive; combining them raises INVALID_SYNTAX at

PHASE 1. The collection-level counterparts — the `virtual` and `final` modifiers on a prompt's `@@@template` header — are in Chapter XXIII.

**Collections (Chapter XXII).** Inheritance is the OO #1 axis; collection inheritance (Chapter XXIII) is OO #2. The two axes compose: a derived prompt in a derived collection participates in both chains at render time (§4.6.2).

> ***`standard.md §4.6.2` — Composition Semantics (excerpt):***
>
> *OO² enables three axes of composition: vertical (prompt inheritance hierarchies — OO #1), horizontal (collection inheritance hierarchies — OO #2 — and parameter-set binding), and orthogonal (a derived prompt living inside a derived collection participates in both chains). The resolution order interleaves them: PHASE 2 applies the collection context (collection chain, effective prompt set, virtual checks); PHASE 3 then runs the prompt inheritance chain (parent to child); section overrides and includes follow.*

**Aliases (Chapter XXV) and deprecation (Chapter XXVI).** Both travel through inheritance. A renamed parent must declare its old `id` as an alias to keep child references resolving; a deprecated parent surfaces its migration message every time a child renders.

**Parameter sets (Chapter XV).** A prompt's embedded parameter sets are part of its L3 data layer, so they ARE inherited through the template-level OO chain — Phase 3 merges the parent's L3 layer, embedded sets included. A child may add or re-declare sets by name.

## 19.5 — Practical examples

**Trivial — a child that overrides one section.**

```
@@@template strict_reviewer inherits code_reviewer
@@@name Strict Reviewer
@@@description Reviewer with a strict format rule

@@@section format
Reply in three bullet points, each at most fifteen words.
```

The child inherits `role` and `task` from `code_reviewer` and overrides `format`. The rendered output is the parent's `role` followed by the parent's `task` followed by the child's `format`.

**Idiomatic — a three-level chain.**

```
Base: reviewer
  @@@template reviewer
  @@@name Reviewer
  @@@description Generic reviewer base
  @@@section role
  You are an expert reviewer.
  @@@section format
  Reply in bullet points.

Middle: code_reviewer
  @@@template code_reviewer inherits reviewer
  @@@section task
  Review code for correctness, performance, and style.

Leaf: python_code_reviewer
  @@@template python_code_reviewer inherits code_reviewer
  @@@section task
  Review Python code, emphasizing type hints and PEP-8.
```

The leaf's render concatenates the three contributions: `role` from `reviewer`, `task` from `python_code_reviewer` (the child's override), and `format` from `reviewer`. The middle prompt's `task` is replaced by the leaf's override; it is invisible to the leaf's renderer.

**Real-world — a child that adds a parameter and a section.**

```
@@@template python_reviewer_typed inherits code_reviewer
@@@name Python Reviewer (typed)
@@@description Reviewer for Python code, emphasizing the typing system

@@@param strict_typing : boolean = true, "Treat typing issues as errors"

@@@section task
Review Python code with emphasis on type hints.

@@@section typing_rules
When `strict_typing` is true, treat missing `-> ReturnType`
annotations as errors. When false, treat them as suggestions.
The current value is @@@=strict_typing.
```

The child adds a parameter and a section. The parent's `role` and `format` are inherited unchanged.

**Anti-pattern — using inheritance for content reuse.**

```
@@@template customer_reply inherits email_opener
@@@name Customer Reply
@@@description Customer reply with shared opener

@@@section body
Thanks for reaching out. ...
```

The author wanted the opener prompt's prose at the start of the customer reply. The opener is not a *kind of* customer reply; it is a shared fragment. The right shape is include, not inherits:

```
@@@template customer_reply
@@@name Customer Reply
@@@description Customer reply with shared opener

@@@section (opener)
@@@include email_opener
```

```
@@@section body
Thanks for reaching out. ...
```

The next chapter covers the `parent` and `self` references that make inheritance practical at any depth.

# Chapter XX — `parent` and `self`

> ***`standard.md §5.4` — Inheritance-aware references (excerpt):***
>
> *When a prompt uses inheritance, special reference patterns enable access to sections from the current prompt and its ancestors. `self.<section>` returns the MOST SPECIALIZED version (current prompt or nearest ancestor); `self.<prompt>.<section>` selects a specific named ancestor. `parent` returns the ENTIRE resolved content of the parent prompt; `parent.<section>` returns the parent's version, skipping the current prompt's override; `parent.<prompt>.<section>` selects a specific ancestor's section. All forms accept parenthesized parameter bindings; only parameters DECLARED by the target may be passed — passing a parameter the target does NOT declare is an UNDECLARED_INCLUDE_PARAMETER error (6.2), exactly as for a regular include (`self` and `parent` use the SAME error code).*
>
> ***`standard.md §4.6.1` — Collection-axis references from a prompt in a derived collection (excerpt):***
>
> *When a prompt P1 is rendered as a member of collection C1, and C1 inherits from C0, the COLLECTION axis (OO #2) is written with the unary `^` operator: `^parent.<template_id>` pulls the full rendered content of `<template_id>` as it would resolve in C0; `^parent.<template_id>.<section>` pulls a named section; `^self.<template_id>` is the current collection's effective member. TWO DISJOINT AXES — NO PRECEDENCE, no silent fallback: `self` / `parent` (no `^`) are the TEMPLATE axis ONLY, `^self` / `^parent` (with `^`) are the COLLECTION axis ONLY. `parent.X` is ALWAYS a section of the parent prompt and `^parent.X` is ALWAYS a member of the parent col-*

> *lection; adding or removing a same-named section or member can never change which one a reference denotes. If the named target does not exist on the stated axis, the result is a clean UNDEFINED_REFERENCE.*

Inheritance gives a child its parent's sections. `parent` and `self` give the child a way to *reach into* the parent — to extend rather than replace, to retrieve a specific ancestor's contribution, and (at L5) to navigate the second OO axis. The two reference families are inseparable from inheritance; without an `inherits` chain (the inline `inherits` clause on a `@@@template` header), they have nothing to resolve against.

## 20.1 — Syntax

Both families use the include keyword as the host:

**Template axis (OO #1) — no `^`:**

```
@@@{include self}                          (rarely used)
@@@include self.<section>
@@@include self.<prompt>.<section>       (a named ancestor's section)
@@@include parent
@@@include parent.<section>
@@@include parent.<prompt>.<section>      (a named ancestor's section
```

**Collection axis (OO #2) — the unary `^` operator (L5):**

```
@@@{include ^parent.<template_id>}                  (parent collecti
@@@{include ^parent.<template_id>.<section>}       (a section of it)
@@@{include ^self.<template_id>}                  (current collectio
```

All forms accept the parenthesized parameter-override syntax of Chapter XVIII:

```
@@@include parent(lang="fr")
@@@include parent.task(format="markdown")
@@@include parent.code_review.task(lang="py")
@@@include self.constraints(strict=true)
@@@{include ^parent.role(lang="fr")}
```

There is NO disambiguation to perform : the axis is fixed SYNTACTICALLY by the presence of the `^` operator. `parent.<name>` (no `^`) is ALWAYS the prompt axis ; `^parent.<template_id>` (with `^`) is ALWAYS the collection axis. The two axes are DISJOINT, with NO precedence and NO silent fallback — a `parent.<name>` that matches no ancestor section is a clean UNDEFINED_REFERENCE, never a collection lookup.

## 20.2 — Purpose

A child that *replaces* a parent's section forfeits the parent's content. Sometimes that is the right move ; often it is not. The child may want to *extend* the parent's content (add a paragraph after), *merge* it (wrap content around the parent's), or simply *quote* it from a sibling location elsewhere in the chain. `parent` and `self` cover those cases.

- `parent` returns the parent's content *skipping the current prompt's override*. The child can compose its body around the parent's contribution.
- `self` returns the *most specialized* version available across the chain. Useful when a child wants to defer to its own overrides if any, falling back to the parent only when none exist.
- The `.` form addresses a *specific named ancestor* in the chain. Useful when the chain is more than two levels deep and the child wants to reach a particular middle level.

At L5 with collections (Chapter XXII), the `^`-prefixed collection axis adds navigation *between members* : `^parent.<template_id>` is the *parent collection's* prompt of that id, as that collection would

render it. The prompt axis (`parent.<section>`) and the collection axis (`^parent.<template_id>`) are disjoint surfaces — the `^` names the axis, so there is no ambiguity to resolve.

## 20.3 — Possibilities

**PHASE 3/8 resolution.** Template-scope `parent` and `self` resolve during PHASE 3 (inheritance) for the section-merging step, and during PHASE 8 (includes) for the explicit include forms. The collection-axis `^parent.<template_id>` resolves during PHASE 8 against the effective prompt set computed in PHASE 2.

**`parent` alone (no `.`) returns the full parent.** A child that writes `@@@{include parent}` pulls the parent's entire resolved content — every section, in parent declaration order — into the spot where the include appears. Useful when the child wants to embed the entire parent and add content around it.

**`parent.<section>` skips the current prompt's override.** If the child overrides a section and *also* references `parent.<that-section>`, the include sees the *parent's* content, not the child's override. This is what makes « extend rather than replace » possible : the child redeclares the section but pulls the parent's content into it before adding its own.

```
@@@section task
@@@include parent.task
Additional Python-specific note: prefer dataclasses.
```

The child's `task` section contains the parent's `task` content followed by the additional sentence.

**`self.<section>` walks the chain.** The most-specialized version of the named section wins. For a chain A → B → C (C is current), `self.task` returns C's `task` if defined, else B's, else A's. Useful when an include wants whatever the chain has settled on for that section.

**`<prompt>.<section>` selects a specific ancestor.** When a chain has three or more levels, `parent.A.task` returns A's `task` specifically, ignoring any intermediate overrides. If `<prompt>` is not in the inheritance chain, the processor raises UNDEFINED_REFERENCE.

**Parameter overrides on these forms.** All five surface forms accept the parenthesized override list:

```
@@@include parent(lang="fr")
@@@include parent.task(format="markdown")
@@@include parent.code_review.task(lang="py")
@@@include self.constraints(strict=true)
@@@include self.base.intro(audience="expert")
```

Parameters not declared on the target raise `UNDECLARED_INCLUDE_PARAMETER` — exactly as a plain include does; `self` and `parent` use the same error code (`standard.md §5.4`, Chapter XXXI). There *is* a leniency, but it concerns *under*-provision: a parameter the target declares but never references in its rendered closure may be omitted without error. Passing a *name the target does not declare* is always an error.

**L5 collection axis `^parent`.** When the current prompt is a member of a derived collection (Chapter XXIII), `^parent.<template_id>` returns the rendered content of the parent collection's prompt of that id. The `.<section>` suffix selects a named section of that prompt. The header form `@@@template P1 inherits ^parent.P` extends template-level OO #1 across the collection axis: the current prompt inherits from the parent-collection's `P`. Section overrides and virtual sections apply on top.

**No cross-axis ambiguity.** A section name and a prompt id can coincide without any collision, because the two axes have DISJOINT surfaces: `parent.<token>` (no `^`) is always read as a section on the prompt axis; `^parent.<token>` (with `^`) is always read as a

member on the collection axis. Neither ever recaptures a target from the other axis; if the named target is absent on the stated axis, the result is a clean UNDEFINED_REFERENCE.

**Verbatim suspends resolution.** A `@@@{include parent}` inside a verbatim block renders literally. This is rare in practice but useful when documenting POO²L itself.

**Qualified ancestor targets.** The `<prompt>` segment in `self.<prompt>.<section>` and `parent.<prompt>.<section>` may be a qualified handle. `parent.marketing.welcome.intro` pins the ancestor lookup to the `welcome` prompt living in `Marketing/welcome`; the processor canonicalises the dotted prefix through the cascade (`§6.1` PHASE 8) before walking the inheritance chain. Without qualification, two ancestors with the same bare name in different collections would be ambiguous.

**No single-segment fallback across axes.** `@@@{include parent.<X>}` (no `^`) is ALWAYS the prompt axis: `<X>` is a section of the parent prompt, and a `<X>` that matches no ancestor section is a clean UNDEFINED_REFERENCE — it never re-reads as a collection member. To reach a parent-collection member, write the collection-axis form explicitly: `@@@{include ^parent.<X>}` (§4.6.1 OO #2). There is no silent fall-through between the two axes.

## 20.4 — Related features and why

**Inheritance (Chapter XIX).** `parent` and `self` have no meaning outside an `inherits` chain. Their resolution (during prompt inheritance, Phase 3, and include resolution, Phase 8) depends on the chain that inheritance establishes.

**Sections (Chapter VII).** All three forms reference named sections. The most-specialized form (`self.<section>`) and the parent-only form (`parent.<section>`) both require the section to be named.

**Includes (Chapter XVII).** `parent` and `self` are sub-cases of include with special target families. The same PHASE 8 mechanism substitutes their content.

**Section composition semantics (Chapter XXIV).** The choice between override, extend, and merge — three distinct ways a child can relate to a parent's section — is the choice between not using `parent`, using `parent.<section>` at the start of the child's override, and using it in the middle. Chapter XXIV names these patterns.

**Collections and collection inheritance (Chapters XXII, XXIII).** The L5 form `^parent.<template_id>` is the second axis. The `^` operator names the axis syntactically, so both axes coexist with no precedence and no ambiguity.

**Virtual templates (Chapter XXIII).** When a virtual prompt's parame-
ters are the abstract signature, a concrete override typically *invokes* the virtual through a derived `^parent.<template_id>` reference. The composition pattern is « override-in-derived calls SUPER-of-virtual-in-base », same as in conventional OO.

## 20.5 — Practical examples

**Trivial — extending a parent section.**

```
@@@template friendly_reviewer inherits code_reviewer
@@@name Friendly Reviewer
@@@description Reviewer that softens the standard format

@@@section format
@@@include parent.format
Use friendly language; address the author by name when possible.
```

The child's `format` section contains the parent's `format` followed by the additional sentence. The pattern is *extend*.

**Idiomatic — wrapping a parent section.**

```
@@@template annotated_reviewer inherits code_reviewer
@@@name Annotated Reviewer
@@@description Reviewer with explicit before/after framing

@@@section task
Begin by stating your overall impression in one sentence.

@@@include parent.task

End by listing the top three improvements in order of impact.
```

The child wraps the parent's `task` between two of its own contributions. The pattern is *merge*.

**Real-world — reaching a specific ancestor in a three-level chain.**

```
@@@-- Chain: reviewer → code_reviewer → python_security_reviewer (cu

@@@template python_security_reviewer inherits code_reviewer
@@@name Python Security Reviewer
@@@description Python reviewer focused on security

@@@section task
Review the code for security issues specifically.

@@@section format
@@@include parent.reviewer.format
For each finding, classify severity using CWE identifiers.
```

The current prompt's `format` extends the *base* reviewer's `format` directly, skipping any intermediate override on `code_reviewer`. The `.` form makes the choice explicit.

**Real-world — L5 collection-scope `parent` (preview).**

```
@@@-- Collection C0 (base) has a member prompt `role`.
@@@-- Collection C1 inherits from C0 and has its own member prompt `role
@@@-- The C1.role prompt:

@@@template C1.role
@@@name Role
@@@description Senior role override in C1

@@@section seniority
principal

@@@section context
Was @@@{include ^parent.role.seniority} in the parent collection.
```

The `^parent.role.seniority` reference (collection axis) returns the *C0* `role`'s `seniority` section content. The rendered `context` section reads : « Was senior in the parent collection. » (if C0's `role.seniority` was « senior »). The L5 chapter (22) covers this pattern in depth.

**Anti-pattern — using `parent` outside an inheritance chain.**

```
@@@template standalone
@@@name Standalone Template
@@@description Not part of any inheritance chain

@@@section role
@@@include parent.role
```

The prompt's header carries no `inherits` clause. There is no parent for `parent` to resolve against ; the include raises UNDEFINED_REFERENCE in PHASE 8. The fix is either to add an `inherits` clause to the `@@@template` header or to use a plain include with a path :

```
@@@section role
@@@include shared/standard_role
```

The next chapter covers *folders* — the cross-level entity that shares the membership keyword with collections but carries no OO semantics.

# Chapter XXI — Folders

> **`standard.md §4.7` — Folders (cross-level) (excerpt):**
>
> *A folder is a pure organizational grouping of templates (and sub-folders), analogous to a file-system directory. Folders carry NO compositional semantics: no inheritance, no virtual members, no `parent` resolution. Folders are an organizational primitive available at ALL LEVELS (L0–L5), not tied to any specific MATURITY LEVEL. A folder is addressed with `/` and a prompt's placement inside it is PHYSICAL and IMPLICIT (where the file is stored) — it is never written in prompt text. A folder is distinguished from an OO² collection structurally: there is no `kind` flag. An entity is a collection precisely because a `@@@collection` block declares it (Level 5, addressed with `.`); it is a folder when it is a storage location with no such declaration.*
>
> **`standard.md §5.10` — Collection Syntax (excerpt):**
>
> *A collection is defined in-language by a `@@@collection <name>` block; a prompt is a member when its `@@@template` id is QUALIFIED — `<collection_name>.<template_id>`. Collections NEST by containment (a collection may contain sub-collections, to unbounded depth); a member's qualified id carries as many `.`-segments as the containment run requires. Each identifier is a Unicode letter followed by letters, digits, and underscore only — NO hyphen. A collection is a POO²L entity addressed with `.` (even when it lives at a folder path); physical folders are addressed with `/` and are never named by `@@@collection`. Folder membership is never declared in prompt text; collection membership lives on the prompt's qualified `@@@template` header.*

A folder is a pure organizational grouping of templates, analogous to a file-system directory. Folders are available at *every* POO²L level — L0 through L5. They carry no compositional semantics : no inheritance, no virtual members, no `parent` resolution. A folder is addressed with `/` (a collection, by contrast, is a POO²L entity addressed with `.`); a prompt's placement inside a folder is PHYSICAL and IMPLICIT — it follows from WHERE the prompt file is stored, NOT from any in-text declaration.

## 21.1 — Syntax

There is NO in-text directive for folder membership. A prompt's folder is the `/`-addressed location where its file is stored :

```
folder path:  team/marketing/campaigns
```

Rules ( `§5.10`, §4.7) :

- A folder path is one or more segments separated by `/`. Each segment follows the identifier lexeme : letters, digits, and underscore only — NO hyphen.
- The path is hierarchical ; `team/marketing/campaigns` places the prompt inside the `campaigns` sub-folder of the `marketing` folder of the `team` folder.
- `@@@collection` is NOT used for folders — it declares a COLLECTION (a POO²L entity, addressed with `.`), and collection membership rides on a prompt's qualified `@@@template` id. There is no `kind` flag : an entity is a collection because a `@@@collection` block declares it, and a folder when it is a storage location with no such declaration. Folder placement is never declared in prompt text.

A folder is purely a storage location : it has no in-language declaration at all. An entity becomes a *collection* — and gains OO² semantics — only when a `@@@collection` block declares it. Until then it is a folder.

> **`standard.md §4.7` — Folder distinction (excerpt) :**
>
> *A folder is a pure physical location ; a collection is a declared POO²L entity (a `@@@collection` block, section 5.10 / 4.6.1). There is no `kind` flag : an entity is a collection because a `@@@collection` block declares it, and a folder because it is a storage location with no such declaration. A folder is addressed with `/` , a collection with `.` . A prompt's placement inside a folder is PHYSICAL and IMPLICIT ; collection membership, by contrast, is carried by the prompt's qualified `@@@template` — never by folder placement.*

A folder has none of the OO² mechanisms — no `inherits`, no virtual members, no `parent`. An entity that needs them must be declared a collection with a `@@@collection` block ( `§4.7` , *Note* at the end).

> **`standard.md §4.7` — Conformance note on OO² constructs (excerpt) :**
>
> *An implementation conforming below LEVEL 5 MUST NOT silently give a recognized construct a DEGRADED meaning. Faced with a `@@@collection` block's `inherits` clause it does not implement, it MUST either reject the input, or preserve it unrendered and refuse to render that collection — never accept the clause and quietly treat the members as a flat namespace. A folder, having no `@@@collection` block, carries none of these constructs ; its semantics remain purely organizational.*

## 21.2 — Purpose

A library of templates grows ; at some point the flat list becomes unmanageable. Folders are the immediate, no-commitment answer : group related templates under a path, like files in a directory. The prompt *content* is unchanged ; only its *location* moves.

Three reasons to reach for a folder :

1. **Reduce list noise.** A sidebar of forty templates becomes a sidebar of seven folders, expandable on demand. The same forty templates are still there ; they are just hidden behind a parent.

2. **Express intent.** A folder named `customer_support` says what the templates inside it are *for*. The templates themselves already carry their `description` ; the folder name is the one-word answer that the description elaborates.

3. **Defer the collection decision.** A team that is not sure yet whether the group is a collection (Chapter XXII) — i.e., whether the members share a contract that should be enforced — can start as a folder and promote to a collection later. The move adds a `@@@collection` block and qualifies the members' ids with the collection name ; folder placement on disk is otherwise unaffected.

A folder is the *neutral* shape. The decision to *not* promote it to a collection is itself information : the templates inside have no shared abstract contract ; they merely happen to be related by topic.

## 21.3 — Possibilities

**Cross-level availability.** Folders work at L0 through L5. Unlike collections, folders are not gated by conformance level — every implementation that claims POO²L conformance supports folders.

**Hierarchy.** Folders nest by path segment. A prompt with membership `a/b/c` (folder path) places it in the `c` sub-folder of the `b` sub-folder of the `a` folder. Implementations typically render this as a tree.

**No inheritance, no virtuals, no `parent`.** A folder has none of the OO² mechanisms. A folder is never the target of an `inherits` clause : because a folder has no `@@@collection` block, naming it as a parent resolves to no declared collection and raises UNKNOWN_COLLECTION.

**Independent metadata.** Folders may carry application-level display attributes — tags, descriptions, colours — and may participate in implementation-defined display and search machinery. None of that is part of the language, and none of it affects the rendered output of templates inside the folder.

**Drag-and-drop semantics.** Folders are typically moved through the implementation's UI. Moving a prompt from one folder to another changes its stored `/`-location ; the prompt's text (including any collection membership on its `@@@template` header) is unchanged.

**Promotion to a collection.** Adding a `@@@collection` block for the entity promotes it. The members qualify their `@@@template` ids with the collection name, *gaining* access to OO² semantics. The reverse — demoting a collection to a folder — removes the block and the prefix on member ids ; any `inherits` or virtual semantics become inert, but the prompt bodies are otherwise unchanged.

**Folder placement is out-of-text.** Because a folder is a pure `/`-addressed location, a prompt's folder membership is not written in its source at all — it is the physical storage path. Only COLLECTION membership (a POO²L entity) is declared in-text, by QUALIFYING the prompt's `@@@template` id ( `@@@template <collection_name>.<template_id>` ). Moving a prompt between folders therefore never edits its source.

**No effect on the processor's resolution phases.** Phase 2 (Collection Context Application) treats a folder as a no-op for compositional purposes (the inheritance and virtual sub-steps are skipped); a prompt that merely lives in a folder, with no collection membership, passes through unchanged. Folders are transparent to POO²L semantics.

## 21.4 — Related features and why

**Collections (Chapter XXII).** A collection is the POO²L-entity counterpart of a folder: addressed with `.`, declared in-language by a `@@@collection <name>` block (with membership carried on each member's qualified `@@@template` id), and carrying OO² semantics. A folder is the same physical storage shape without those semantics; the boundary between the two is whether a `@@@collection` block declares the entity.

**Collection inheritance and virtual templates (Chapter XXIII).** None of those mechanisms apply to folders. An attempt to use them on a folder is either silently ignored or raised as an error; the spec leaves the choice to the implementation.

**Includes and inheritance (Chapters XVII, XIX).** Both work identically regardless of folder membership. A prompt's includes and inherits reference other templates by id; the referenced templates may live in any folder.

**`^parent` (Chapter XX).** The L5 collection-axis form `^parent.<template_id>` does *not* resolve against a folder. A prompt that merely lives in a folder is a member of no collection, so a `^parent` reference has no collection to resolve against and raises UNDEFINED_REFERENCE. The prompt axis (`parent.<section>`, no `^`) is unaffected — it depends only on the prompt's own `inherits` chain.

> ***`standard.md §4.6.1` — Collection axis and folders (excerpt):***

> *The two OO axes are DISJOINT and named SYNTACTICALLY: `self` / `parent` (no `^`) is the TEMPLATE axis; `^self` / `^parent` (with `^`) is the COLLECTION axis. There is NO precedence and no silent fallback between them. A prompt whose membership is a folder belongs to no collection, so a `^parent` reference has no collection to resolve against; the template-axis `parent` remains available and depends only on the prompt's `inherits` chain.*

**The collection restriction on parameter sets (Chapter XV).** The restriction triggers whenever a prompt is a member of a collection — i.e. whenever its `@@@template` id is qualified with a collection name. A prompt that merely lives in a folder (a bare, unqualified `@@@template` id) is NOT subject to the restriction: folder placement is physical and carries no membership declaration, so embedded parameter sets remain allowed there.

**Promotion/demotion in tooling.** Some implementations expose folder/collection toggles to authors; some do not. The IziBuddy reference implementation exposes the folder/collection toggle as a feature gated by the *containerKindUI* feature flag; flipping it adds or removes the `@@@collection` block behind the scenes.

## 21.5 — Practical examples

**Trivial — a single folder grouping three templates.**

```
Folder "customer_support"  (a storage location — no @@@collection block)

Template 1 — bug_report_triage  (stored under customer_support/):
  @@@template bug_report_triage      (bare id — folder placement is phys

Template 2 — refund_reply_template  (stored under customer_support/):
  @@@template refund_reply_template  (bare id — folder placement is phys

Template 3 — escalation_summary  (stored under customer_support/):
  @@@template escalation_summary     (bare id — folder placement is phys
```

Three unrelated templates share a folder for navigation only. None inherits from another; no shared parameters; no virtual members. The folder reduces sidebar noise; it does not encode a contract.

**Idiomatic — nested folders.**

```
Folder "team"                            (a storage location)
Folder "team/engineering"               (a storage location)
Folder "team/marketing"                 (a storage location)
Folder "team/marketing/campaigns"       (a storage location)

Template summer_campaign_brief  (stored under team/marketing/campaign
  @@@template summer_campaign_brief      (bare id — folder placement
```

Four levels of nesting. Each segment is just a folder. Promotion of any level to a collection (say, declaring a `@@@collection campaigns` block and qualifying the prompt as `@@@template campaigns.summer_campaign_brief`) would activate OO² semantics for that group without touching the sibling levels. Note that the collection name is flat — the `/`-path organizes storage; the `.`-prefix names the collection.

**Real-world — folder-to-collection promotion.**

The team starts with a `reviewers` folder (a storage location holding three unrelated reviewer templates, each with a bare `@@@template` id). As the library grows, the team realizes the three reviewers share a contract: the base collection should expose `summarize`, `score`, and `suggest` as abstract members that every derived collection must implement. Promoting the folder to a collection makes that contract expressible — declare a `@@@collection` block and qualify the members:

```
@@@collection reviewers
@@@template summarize
@@@template score
```

```
@@@template suggest
@@@end-collection
```

Declaring the `@@@collection` block (and re-qualifying the member ids as `reviewers.summarize`, etc.) *is* the whole of the promotion. The contract itself is then expressed by marking each abstract member with the `virtual` modifier on its `@@@template` header (Chapter XXIII) — virtuality is never a collection-level field. Those members gain access to the collection-axis `^parent.<template_id>` and become subject to the UNIMPLEMENTED_VIRTUAL_TEMPLATE check.

**Anti-pattern — using a folder for a contract.**

```
Folder "reviewers"  (a storage location — no @@@collection block)

Templates inside use a hand-rolled convention: each declares the
same three sections (`summarize`, `score`, `suggest`).
```

The convention is informal. New reviewer templates may forget to declare a section; no error fires. The fix is to promote the folder to a collection with the corresponding virtual templates, so the contract is enforced.

The next chapter introduces *collections* — the OO² entity that this chapter has been carefully distinguishing from.

# Chapter XXII — Collections

> ***`standard.md §4.6.1` — Template Collections (excerpt):***
>
> *A prompt collection is the OO² (horizontal-axis) entity of POO²L. Its methods are the templates that belong to it. A collection may inherit from another collection, declare abstract (virtual) prompt members that derived collections MUST implement, and a prompt rendered inside a derived collection may reference the parent collection's templates via the collection-axis reference `^parent.<template_id>`. A prompt declares membership by QUALIFYING its `@@@template` id — `@@@template <collection_name>.<template_id>`; the collection itself is a declared in-language entity (a `@@@collection <name>` block), not a sidecar. Collections NEST by containment (a collection may contain sub-collections, to unbounded depth). When collection C1 declares `inherits C0`, the effective prompt set of C1 is computed root-first: start with C0's effective set, then add or override (by id) with C1's own members.*
>
> ***`standard.md §5.10` — Collection Syntax (excerpt):***
>
> *A collection is defined in-language by a `@@@collection <name>` block: the header carries the collection's own data — its optional `inherits` parent and its collection-level parameter sets — and lists the collection's public members via `@@@template <member_id>` lines. Collections NEST by containment (unbounded depth); each identifier segment is a Unicode letter then letters, digits, and underscore only — NO hyphen. A collection is a POO²L entity addressed with `.` (`<outer>.<inner>.<template_id>` when it nests); physical folders*

> *are addressed with `/` and are never named by `@@@collection`. There is no `kind` flag: an entity is a collection precisely because a `@@@collection` block declares it.*

A collection is the L5 entity that treats a group of related templates as a *class*. Its members — the templates that declare membership — are the *methods*. A collection may inherit from another collection; it may declare virtual members that derived collections must implement; a prompt rendered inside a derived collection may navigate the collection chain through the collection-axis `^parent.<template_id>` references. This chapter covers the entity itself; the next chapter covers its inheritance machinery.

## 22.1 — Syntax

**Definition** — the collection is declared in-language by a `@@@collection <name>` block that lists its public members:

```
@@@collection <collection_name> [inherits <parent_collection>]
@@@template <member_id>                    -- zero or more — the public su
@@@parameter-set <name>                  -- optional; collection-level set
...
@@@end-parameter-set
@@@end-collection
```

**Membership** — carried by QUALIFYING a prompt's `@@@template` header id; there is no separate membership directive:

```
@@@template <collection_name>.<template_id>
```

The `<collection_name>` prefix names the collection the prompt belongs to; the final segment is the prompt's id. (Compare with a folder from Chapter XXI, whose placement is physical and never written in prompt text.)

There is no `kind` flag and no sidecar : an entity is a collection precisely *because* a `@@@collection` block declares it. The block's optional `inherits` clause names the parent collection (Chapter XXIII) ; collection-level parameter sets live in the block too. Virtual templates are *not* listed in the block — they are declared by the `virtual` modifier on an individual member's `@@@template` header (Chapter XXIII).

Rules :

- Collections NEST by containment. A member's qualified id carries as many `.`-segments as its containment run requires (`<outer>.<inner>.<template_id>` when a collection contains a sub-collection), to unbounded depth. A collection is a POO²L entity addressed with `.` (physical folders, addressed with `/`, are a separate, implicit layer). A *library* (`@@@library`, §5.16) is a distinct namespace layer that bundles collections ; it is not a substitute for containment.
- A collection's parent (the `inherits` clause on its `@@@collection` header) must itself be a declared collection. A prefix that resolves to no declared collection raises UNKNOWN_COLLECTION ; one that resolves to two or more raises AMBIGUOUS_COLLECTION.
- A member's qualified `@@@template` header opens its entity and so MUST appear first in that prompt (`§5.2`, *the `@@@template` header is a structural exception to free declaration order*) ; the remaining declarations may be reordered.

## 22.2 — Purpose

A collection answers a question that folders cannot : *what does it mean to be a member of this group ?* For some groups the answer is « nothing — these templates merely live in the same drawer » ; that is a folder. For other groups the answer is « every member must implement these three sections, with this parameter signature, and may be substituted for any other member » ; that is a collection.

The substitution principle is the headline. A collection is a *type*; its concrete members are the *values* of that type. A caller of the collection's `reviewer.summarize` does not care which concrete reviewer answers — `code_reviewer`, `prose_reviewer`, or `python_security_reviewer`. They all implement `summarize`; they all have the parameters the abstract `summarize` declared. Polymorphism at the prompt level.

The benefits:

1. **Contract enforcement.** Virtual templates declare an abstract signature; concrete members must implement it. The processor catches missing implementations (UNIMPLEMENTED_VIRTUAL_TEMPLATE).

2. **Family-wide refinement.** A derived collection can override *any* base collection's member by id without touching the bases. Two reviewer families share the base; one of them sharpens the `format` member; the other does not. No prompt in either family duplicates the unchanged bits.

3. **A second OO axis.** The first axis (`inherits` between templates) specializes a single prompt. The second axis (`inherits` between collections) specializes the *family* of templates. Together they cover the cases real teams hit when their prompt library mirrors their organizational structure (Chapter XXIV).

## 22.3 — Possibilities

**Phase 2 resolution.** The collection chain is resolved in Phase 2 (Collection Context Application), before any template- level work. The processor walks the chain root-first, computes the *effective prompt set* (the deduplicated union of members across the chain, keyed by id), checks for cycles and folder- parent mismatches, and validates virtual implementations.

**The effective prompt set.** For a leaf collection C1 that inherits from C0, the effective prompt set is:

```
{ every member of C0 (recursively, by id) } ∪
{ every member of C1, overriding by id }
```

When a member id appears in both, the C1 declaration *replaces* the C0 declaration (member override). The effective set is what the rest of the pipeline operates on: includes, inheritance chains, `^parent.<template_id>` references all use it.

**Member override.** A derived collection's prompt with id `X` overrides the parent collection's prompt with id `X`. Matching is by *canonical id* — a member override never matches by a parent's alias. The override is *whole-template*: the derived `X` replaces the parent's `X` entirely. To extend rather than replace, the derived `X` uses the collection-axis `^parent.<template_id>` to retrieve the parent's version and compose around it.

**Virtual templates.** A member whose `@@@template` header carries the `virtual` modifier is abstract: its parameters constitute the signature, and it has NO rendered body. A derived collection must provide a non-virtual prompt with the same id; otherwise, rendering that virtual raises UNIMPLEMENTED_VIRTUAL_TEMPLATE. Virtual templates and their resolution rules are the subject of Chapter XXIII.

**Parameter sets at collection scope.** When a prompt belongs to a collection, parameter sets are declared in the collection's `@@@collection` block, not embedded in the prompt source (Chapter XV's restriction). The collection-scope sets are inherited root-first along the chain.

**Visibility (public / private).** Visibility is structural, not a keyword. A member listed in its collection's block (via `@@@template <template_id>`) is **public** — addressable from outside the collection. A member that is *not* listed is **private**: it exists and is reachable from within the collection and its descendants (intra-collection includes, `parent.`), but an external reference to it raises NOT_EXPORTED.

**Application-level metadata.** Tags, descriptions, colours, and other display attributes are orthogonal to the OO² machinery. They are useful for tooling — search, display, search-by-tag, color-coded sidebars — and have no effect on rendering. An implementation *may* add such metadata outside prompt text; doing so is an application-level extension (allowed by `§9.3`), not a language extension.

> ***`standard.md §9.3` — Implementation extensions (excerpt):***
>
> *POO²L distinguishes LANGUAGE-LEVEL (syntax, semantics) — which is CLOSED — from APPLICATION-LEVEL (tooling, storage, metadata) — which is OPEN. Implementations MAY freely add features that do not modify the language itself: additional tooling, alternative storage backends, UI/UX enhancements, and application metadata stored OUTSIDE prompt text (user-specific parameter sets, tags, flags, access control, version history, usage statistics). The distinction ensures POO²L templates remain portable across implementations while implementations can innovate at the application level.*

**Promotion from a folder.** Adding a `@@@collection` block for an entity that previously had none promotes it from a folder to a collection. Its members qualify their `@@@template` ids with the collection name, gaining access to collection-scope features (virtual templates in members, the collection-axis `^parent.<template_id>` from members, collection inheritance, member override).

**Demotion to a folder.** Removing the `@@@collection` block — and the collection prefix from the members' ids — demotes the entity back to a folder. Any `inherits` or virtual semantics become inert. An implementation may emit a warning if the entity had non-trivial collection-scope state (a parent chain, virtual members) that is about to be lost.

**Nesting by containment.** Collections nest : a collection may contain sub-collections, to unbounded depth, and a member's qualified id carries one `.`-segment per containment level (`<outer>.<inner>.<template_id>`). A *library* (`@@@library`, §5.16) is a separate namespace layer that bundles collections ; it is packaging, not a containment level. Physical folders (`/`) remain available for storage organization and are transparent to addressing.

**Capacity.** A collection has no maximum size in the language. Implementations may impose UI-level limits for ergonomics.

**Two composing relations.** Lookup uses two relations. *Containment* — the outer-to-inner nesting run — answers « where can I find that name ? » ; `inherits` — the collection chain — answers « which definition wins when several exist ? ». They compose : the `inherits` chain is walked to compute the effective prompt set, and containment qualifies which collection a name is sought in. An unqualified handle resolves through the cascade of §6.1 PHASE 8 ; ambiguity at one level raises AMBIGUOUS_INCLUDE (or AMBIGUOUS_COLLECTION / AMBIGUOUS_ENTITY for the collection itself).

## 22.4 — Related features and why

**Folders (Chapter XXI).** A folder is the same physical storage shape without OO² semantics. The only language-level difference is whether a `@@@collection` block declares the entity : a collection has one, a folder does not. Promotion adds the block (and qualifies member ids) ; demotion removes it.

**Collection inheritance and virtual templates (Chapter XXIII).** The detailed machinery of `inherits` between collections, the abstract-member contract, the override semantics, and the UNIMPLEMENTED_VIRTUAL_TEMPLATE error. Chapter XXIII is the full treatment of what this chapter sketches.

**Section composition semantics (Chapter XXIV).** OO #1 (template- level) and OO #2 (collection-level) compose orthogonally. A prompt may be a child of another prompt *and* a member of a derived collection; the two chains resolve independently in their respective phases.

**`^parent` references (Chapter XX).** The L5 collection-axis form `^parent.<template_id>` returns the *parent collection's* prompt of that id, as that collection would render it. The `^` operator names the axis syntactically, so it never collides with the template-axis `parent.<section>` — two disjoint axes, no precedence.

**Parameter sets (Chapter XV).** Collection-scope sets live in the `@@@collection` block; the embedded form is forbidden in collection-member templates.

**Includes (Chapter XVII).** Includes resolve their references against the *effective prompt set* computed in Phase 2, so a member of a derived collection can reference a member of the *parent* collection by id transparently. If the parent had a member named `glossary` that the derived collection did not override, an include of `glossary` finds the parent's version.

**Aliases and deprecation (Chapters XXV, XXVI).** Aliases are LOCAL and non-qualified. The effective set keys on *canonical* ids, so a member override matches a parent member by canonical id — NOT by a parent's alias. A member's own alias makes it addressable under that name locally, but it does not let a derived member override a parent by the parent's alias.

## 22.5 — Practical examples

**Trivial — a single collection with two members.**

```
@@@collection reviewers
@@@template code_reviewer
```

```
@@@template prose_reviewer
@@@end-collection

@@@template reviewers.code_reviewer
@@@name Code Reviewer
@@@description Reviews code for best practices
...

@@@template reviewers.prose_reviewer
@@@name Prose Reviewer
@@@description Reviews prose for clarity
...
```

The collection's block lists its two public members; each member qualifies its own `@@@template` id with the collection name. The OO² machinery is available — the team could declare virtual members next — but at this point the collection is structurally indistinguishable from a folder. The difference becomes visible only when virtual templates or collection inheritance enters.

**Idiomatic — promoting a folder to a collection.**

The team starts with a `team/marketing` folder (placement is physical; nothing is written in prompt text). Three templates inside use the convention that every marketing prompt has a `tone`, a `brand_voice`, and a `target_audience` parameter. The convention is fragile; new templates may forget. Promotion declares a collection in-language and qualifies the members:

```
@@@collection marketing
@@@template summer_campaign
...
@@@end-collection
```

Now the team declares a *virtual base member* whose parameters state the contract, and every concrete member overrides it with its own body. Chapter XXIII covers the virtual mechanism; this example sketches the *trigger*.

**Real-world — a collection inheriting from another.**

```
@@@collection reviewers
@@@template task
@@@end-collection

@@@collection code_reviewers inherits reviewers
@@@template task
@@@end-collection
```

`code_reviewers` is a derived collection : it inherits the members of `reviewers` and overrides or adds its own. Note that *inheritance* (the `inherits` clause) and *containment* (a collection nested inside another) are different relations : inheritance expresses specialization, containment expresses grouping. A collection may inherit from any declared collection, as long as the rules (no cycle, parent exists) are respected.

**Anti-pattern — using a collection where a folder was needed.**

```
@@@collection scratch
@@@end-collection
```

The « scratch » entity holds half-finished templates that share nothing in common. Declaring a `@@@collection` block for it puts every member under the OO² resolution path in Phase 2 even though no member uses any collection-scope feature. The fix is to demote it to a folder — remove the block and the collection prefix from the member ids.

The next chapter walks the inheritance and virtual-template machinery introduced in passing here.

# Chapter XXIII — Collection Inheritance, Virtual and Final Templates

> ***`standard.md §4.6.1` — Collection inheritance, virtual and final templates (excerpt):***
>
> *When collection C1 declares `inherits C0` on its `@@@collection` header, C1's effective prompt set is computed root-first: start with C0's effective set, then add or override (by CANONICAL id) with C1's own members. A collection MUST NOT inherit from itself or participate in a cycle (CIRCULAR_COLLECTION_INHERITANCE), and its declared parent MUST resolve to a `@@@collection` block — a name that names no collection (e.g. a plain folder) raises UNKNOWN_COLLECTION. Collections NEST by containment to unbounded depth. Virtual templates are declared at the **prompt level** via the `virtual` modifier on the (collection-qualified) `@@@template` header — the prompt's `@@@param ...` declarations constitute the abstract signature. A descendant collection MUST provide a non-virtual prompt with the same id for every virtual ancestor; otherwise rendering raises UNIMPLEMENTED_VIRTUAL_TEMPLATE. A descendant MAY re-declare a member as `virtual` to escalate abstraction. A member of a derived collection reaches the parent collection's member with the collection-axis reference `^parent.<template_id>`. Symmetric to `virtual`, a prompt may carry the `final` modifier to declare itself **sealed** — no descendant collection MAY override the prompt by id (OVERRIDE_OF_FINAL_TEMPLATE). `virtual` and `final` are mutually exclusive on the same prompt (a header bearing both is INVALID_SYNTAX).*

This is the chapter where OO²'s second axis becomes concrete. A collection may *inherit* from another collection — an `inherits` clause on its `@@@collection` header — making the derived collection a specialization of the parent. A prompt member may be declared *virtual* — the `virtual` modifier on its `@@@template` header — making it an abstract contract that descendant collections must implement. A prompt member may equivalently be declared *final* — the `final` modifier on its `@@@template` header — sealing it against any descendant override. Together, these mechanisms make the collection a *type*: callers obtain a prompt of a given role without knowing which concrete implementation answers, and the author of a collection can decide which members are open to specialization and which are not.

## 23.1 — Syntax

**Collection inheritance** — an `inherits` clause on the `@@@collection` header:

```
@@@collection <collection_name> inherits <parent_collection>
@@@template <member_id>
...
@@@end-collection
```

**Virtual templates** — the `virtual` modifier on the abstract member's (collection-qualified) `@@@template` header:

```
@@@template virtual <collection_name>.<template_id>
```

The `virtual` modifier takes no arguments. The prompt's parameters (declared via `@@@param ...`) constitute its *abstract signature* — the contract every concrete override must match. A virtual prompt carries NO rendered body; the modifier, like the rest of the identity header, is stripped from the rendered output. The

modifier is valid ONLY on a collection-qualified `@@@template`: a prompt belonging to no collection has no OO² chain to override it, so `virtual` (and `final`) on a bare id is INVALID_SYNTAX.

**Final templates** — the `final` modifier on the sealed member's `@@@template` header:

```
@@@template final <collection_name>.<template_id>
```

The `final` modifier takes no arguments. It declares the prompt **sealed** as a collection member: no derived collection MAY override the prompt by id. The modifier is stripped from the rendered output. `virtual` and `final` are mutually exclusive on the same header — an abstract member cannot also be sealed.

Rules (`§4.6.1`):

- A collection MUST NOT inherit from itself. (Self-cycle: error.)
- A collection MUST NOT participate in an inheritance cycle. (Cycle: CIRCULAR_COLLECTION_INHERITANCE.)
- The `inherits` clause's target MUST be a declared collection. A name that resolves to no `@@@collection` block (e.g. a plain folder, which has no block) raises UNKNOWN_COLLECTION; a name that resolves to two or more collections raises AMBIGUOUS_COLLECTION.
- A descendant collection MUST provide a non-virtual prompt with the same id for every virtual ancestor; otherwise rendering raises UNIMPLEMENTED_VIRTUAL_TEMPLATE.
- A descendant collection MAY re-declare a member as `virtual` to escalate abstraction; further descendants must then re-implement.
- A descendant collection MUST NOT redeclare a prompt with the same id as one marked `final` upstream in the chain. (Override of sealed member: OVERRIDE_OF_FINAL_TEMPLATE.)

- `virtual` and `final` MUST NOT appear on the same `@@@template` header. (A header bearing both — `@@@template virtual final X` — is INVALID_SYNTAX.)
- A descendant MAY take a `virtual` member, provide a concrete override, and mark the override `final` — fixing the implementation for all further descendants.

## 23.2 — Purpose

The first OO axis (Chapter XIX) specializes *one prompt*. The second axis specializes *a family*. A team that maintains three different reviewer flavours — `code_reviewer`, `prose_reviewer`, `legal_reviewer` — and wants each flavour to expose the same three methods, with the same signature, in a way the rest of the system can rely on, needs collection inheritance and virtual templates.

Without these two mechanisms, the team is stuck with conventions : « every reviewer should have a `summarize`, a `score`, and a `suggest` section, with the same parameters ». Conventions decay. The fourth reviewer added six months later forgets the `score` section ; nothing fires. With collection inheritance and virtual templates, the convention becomes a *contract* : the parser refuses to render a reviewer that fails to implement all three.

The benefits compound. A piece of caller code that asks for « `reviewers.summarize` » does not need to know which concrete reviewer answers ; the collection chain resolves to the appropriate implementation. A new reviewer added later — sharing the same virtual base — is automatically substitutable for any of its siblings.

## 23.3 — Possibilities

**Phase 2 sub-steps.** The processor at Phase 2 walks the collection chain root-first :

1. **Resolve the chain.** Start at the leaf, follow each collection's `inherits` toward the root. Detect cycles (CIRCULAR_COLLECTION_INHERITANCE).

2. **Compute the effective prompt set.** Take the union of all member templates along the chain, keyed by prompt id; for each id, the closest declaration wins (member override).
3. **Validate virtuals.** For every prompt id marked `virtual` anywhere in the chain, ensure a non-virtual prompt with that id exists in the effective set. Otherwise, UNIMPLEMENTED_VIRTUAL_TEMPLATE.
4. **Merge collection-level configurations.** Parameter sets, constraints, and permissions are merged root-first.

**Member override.** A derived collection's prompt with id `X` overrides the parent collection's prompt with id `X`, matched by CANONICAL id (never by a parent's alias). The override is whole-template — replacing the entire member. To preserve and extend, the derived `X` uses the collection-axis `^parent.<template_id>` reference (Chapter XX) to reach the parent's `X`.

**Virtual templates carry their signature.** A virtual prompt declares its parameters; the parameter list is the abstract signature. A concrete override must declare the same parameters (or a compatible superset — adding optional parameters is allowed; removing or changing types is a contract break).

**Why template-level declaration of virtuality.** The specification puts the `virtual` modifier *on the prompt's own* `@@@template` *header*, not on the collection (as a `virtual_templates: [id, ...]` list). The reasoning (`§4.6.1`, *Rationale for template-level declaration*): a virtual prompt's contract is its parameter signature. A collection-level list captures only the id, losing the parameters. Putting the marker on the prompt also preserves the symmetry with OO #1, where `@@@section virtual <name>` lives inside the prompt, where the section's contract lives.

**Escalation.** A descendant collection may re-declare a member as `virtual` again, returning it to abstract status for *its* descendants. The pattern is unusual but valid; the spec explicitly permits it. The effect: the prompt is concrete in the intermediate chain but abstract in the next layer down, requiring re-implementation.

**Sealing with `final`.** The complement to escalation is *sealing*. A prompt marked `final` is closed against override at the collection axis: a derived collection that declares a member with the same id raises OVERRIDE_OF_FINAL_TEMPLATE at Phase 2. The rationale mirrors the *sealed class* of class-based OO — a base author who has good reasons to forbid specialization (a security-critical prompt, an audited prompt, a piece of vetted legal boilerplate) can seal the member and rely on the conformance checker to enforce the rule. `final` is also the natural endpoint of an abstraction lifecycle: an ancestor declared the member `virtual`, an intermediate descendant provided the concrete body, and a leaf descendant — or the intermediate itself — marks the override `final` to freeze it against further drift. The two modifiers are mutually exclusive on a single `@@@template` header (both on the header is INVALID_SYNTAX).

**The `^parent.<template_id>` reference.** When a derived collection's member references `^parent.<template_id>` (collection axis, `^` prefix), it reaches the *parent collection's* prompt of that id, rendered in the parent's scope. The form `.<section>` selects a named section of that prompt. The header form `@@@template P inherits ^parent.P` declares the current prompt's OO #1 parent to be the parent collection's `P`; the template-level chain extends across the collection axis.

**Cross-axis composition.** A prompt may simultaneously participate in both axes:

```
@@@template C1.P1 inherits P    -- membership in C1 (OO #2, where C1 inh
                               -- plus parent prompt P (OO #1 — Chapter <
```

The qualified id `C1.P1` carries collection membership (OO #2 — this chapter), where C1 inherits from C0, while the inline `inherits P` clause names the OO #1 parent prompt. Phase 2 runs the collection chain first; Phase 3 runs the prompt chain afterward. The two chains do not interact except through `^parent.<template_id>` references that bridge them.

**Parameter-set inheritance.** Collection-scope parameter sets are inherited along the same root-first chain (Chapter XV). A derived collection adds or overrides sets by name; the parent's unaffected sets remain available.

**No collection-level virtuality field.** Virtuality is declared *only* at the prompt level, via the `virtual` modifier on the `@@@template` header. The `@@@collection` block (which carries the collection's `inherits` parent and its parameter sets, and lists its public members) has **no** `virtual_templates` field — the specification explicitly states that virtual templates are NOT declared at collection scope (§4.6.1), because a list of ids would lose the parameter signature that constitutes the contract (*Rationale for template-level declaration*). A tool MAY *derive* a read-only list of a collection's virtual members from the template-level modifiers for display, but that is a tooling artefact, not an authoring surface.

## 23.4 — Related features and why

**Collections (Chapter XXII).** The host entity. Inheritance and virtual templates add meaning to a collection that a folder does not have.

**Inheritance (Chapter XIX).** The first OO axis. Both axes use the keyword `inherits` — the first as a clause on the `@@@template` header, the second as a clause on the `@@@collection` header — and the two are independent.

**`^parent` references (Chapter XX).** The L5 collection-axis form `^parent.<template_id>` is the navigation primitive within the collection chain. The `^` operator names the axis syntactically, so it never collides with the template-axis `parent.<section>` — two disjoint axes, no precedence.

**Section composition semantics (Chapter XXIV).** Override, extend, and merge — Chapter XXIV's three patterns — apply at both axes. The collection axis adds member-override; the section patterns then apply within each member.

**Parameter sets (Chapter XV).** Collection-scope sets inherit through the chain. Each level may add or override sets by name.

**Aliases (Chapter XXV).** Aliases are LOCAL and non-qualified. The effective set keys on *canonical* ids, so a member override matches a parent member by canonical id — NOT by a parent's alias. A member's own alias makes it locally addressable under that name; it does not let a derived member override a parent by the parent's alias.

## 23.5 — Practical examples

**Trivial — collection inheritance with no virtuals.**

```
@@@collection C0
@@@template P
@@@end-collection

@@@template C0.P
@@@name P
@@@description Base P prompt
Body of P.

@@@collection C1 inherits C0
@@@template Q
@@@end-collection

@@@template C1.Q
@@@name Q
@@@description Q prompt added by C1
Body of Q.
```

Effective prompt set of C1 : { P (inherited from C0), Q (own) }. A reference to P from within C1 finds P; the reference resolves through the chain transparently.

**Idiomatic — member override.**

```
@@@collection C0
@@@template report
@@@end-collection

@@@template C0.report
Base report.

@@@collection C1 inherits C0
@@@template report
@@@end-collection

@@@template C1.report
Specialized report.
```

The effective prompt set of C1 binds id `report` to C1's declaration. A render of `report` in C1's scope produces « Specialized report. » The parent's content is unreachable except via `parent.report` from a member of C1.

**Real-world — virtual contract enforcement.**

```
@@@collection reviewers
@@@template task
@@@end-collection

@@@template virtual reviewers.task
@@@name Review Task
@@@description Abstract review task — implement in derived
               reviewers
@@@param topic : text, "What to review"
(no body — the parameter list is the signature)

@@@collection code_reviewers inherits reviewers
@@@template task
@@@end-collection

@@@template code_reviewers.task
@@@name Code Review Task
```

```
@@@description Reviews code
@@@param topic : text, "What to review"
Review the code @@@=topic for correctness and style.
```

Rendering `task` via `code_reviewers` succeeds ; rendering it via `reviewers` alone fails (UNIMPLEMENTED_VIRTUAL_TEMPLATE). The virtual contract requires the `topic` parameter ; the concrete override has it.

**Real-world — collection-scope `parent`.**

```
@@@collection C0
@@@template role
@@@end-collection

@@@template C0.role
@@@section seniority
senior

@@@collection C1 inherits C0
@@@template role
@@@end-collection

@@@template C1.role
@@@section seniority
principal
@@@section context
Was @@@{include ^parent.role.seniority} in the parent collection.
```

Rendering `role` via C1 : `seniority` is `principal` ; `context` reads « Was senior in the parent collection. » (because the collection-axis `^parent.role.seniority` returns C0’s `role`’s `seniority` section content).

**Real-world — sealing a member with `final`.**

```
@@@collection reviewers
@@@template legal_notice
```

```
@@@end-collection

@@@template final reviewers.legal_notice
@@@name Legal Notice
@@@description Audited legal notice — sealed
This output is provided under the terms of the Acme Reviewer
Agreement; see legal@acme.example for details.

@@@collection internal_reviewers inherits reviewers
@@@template legal_notice
@@@end-collection

@@@template internal_reviewers.legal_notice
Internal-only legal notice, friendlier tone.
```

The base collection seals `legal_notice` with the `final` modifier. The descendant collection tries to override the sealed member; Phase 2 raises OVERRIDE_OF_FINAL_TEMPLATE. The fix is structural: either the legal notice is allowed to vary across collections (drop the `final` modifier on the base) or the descendant must accept the sealed content (remove the override).

**Anti-pattern — a collection inheriting from a folder.**

```
Folder "drafts"  (a storage location — no @@@collection block)

@@@collection marketing inherits drafts
@@@end-collection
```

The named parent `drafts` is a folder: it has no `@@@collection` block, so the `inherits` clause resolves to no declared collection and raises UNKNOWN_COLLECTION. The fix is to promote `drafts` to a collection if its semantics warrant it (give it a `@@@collection` block), or to remove the `inherits` and let `marketing` be a base collection.

**Anti-pattern — a circular collection chain.**

```
@@@collection A inherits B
@@@end-collection

@@@collection B inherits A
@@@end-collection
```

Phase 2 detects the cycle and raises CIRCULAR_COLLECTION_INHERITANCE. The fix is structural : one direction must be the true ancestor ; the other should not inherit at all, or both collections should derive from a common base C.

The next chapter formalizes the three patterns by which a child relates to a parent's content — *override*, *extend*, and *merge* — across both OO axes.

# Chapter XXIV — Section Composition Semantics

> ***`standard.md §4.6.2` — Composition Semantics (excerpt):***
>
> *OO² enables vertical composition (prompt inheritance hierarchies, OO #1), horizontal composition (collection inheritance hierarchies, OO #2, and parameter-set binding), and orthogonal composition (a derived prompt living inside a derived collection participates in both chains). The resolution order is: PHASE 2 applies the collection context (collection chain, effective prompt set, virtual checks); PHASE 3 runs prompt inheritance (parent to child) and applies section overrides (child overrides parent); PHASE 8 resolves includes; PHASE 9 substitutes parameters and validates constraints.*

A child prompt that inherits from a parent has three available relationships to any given parent section: it can *replace* it, it can *extend* it, or it can *merge* its own content around it. POO²L does not introduce new keywords for these three patterns — they emerge from the combinations of section overrides (Chapter VII) and `parent.<section>` references (Chapter XX) you already know. This chapter names the three patterns so they can be talked about, chosen between, and read in someone else's prompt at a glance.

## 24.1 — Syntax

The three patterns share a host section in the child and differ in whether and where the child uses `parent.<section>`:

**Override** — the child declares the section, the parent's content is unreachable from the rendered output of *this* section:

```
@@@section task
Replacement content. The parent's task content does not appear.
```

**Extend** — the child declares the section, pulls the parent's content first via `parent.<section>`, then appends its own :

```
@@@section task
@@@include parent.task
Additional content the child wants to add at the end.
```

**Merge** — the child declares the section, places the parent's content somewhere *inside* its own contribution :

```
@@@section task
Opening contribution from the child.

@@@include parent.task

Closing contribution from the child.
```

The three are syntactically simple ; the names exist to capture the *authoring intent* behind each combination.

## 24.2 — Purpose

Inheritance without these patterns is a blunt instrument : the child either inherits the parent's section unchanged or replaces it wholesale. Most real specializations want something between the two. The three named patterns cover the in-between space :

- **Override** is for the cases where the child's specialization fundamentally changes the section. A reviewer that targets a different output format does not extend the parent's format section ; it replaces it.

- **Extend** is for the cases where the parent's section is *generally right* and the child has *additional* concerns. A Python reviewer extends the base reviewer's `task` with Python-specific notes ; the parent's general instructions are still right.

- **Merge** is for the cases where the parent's section contributes *part* of the child's content — perhaps the middle, perhaps the end — and the child wants to frame it.

The three names also serve as *code review heuristics*. A reviewer looking at a child section can ask : is this override, extend, or merge ? If the answer is « I cannot tell », the section needs clarification (a comment, a restructuring) before it ships.

## 24.3 — Possibilities

**Override is the default when `parent` is absent.** The child section without any `parent.<section>` reference is an override. The parent's content is unreachable from the child's section *as rendered*. It can still be reached from *other* sections in the child (a different section can include `parent.<that-section>`).

**Extend places `parent` at the start.** A section that opens with `@@@{include parent.<self>}` and then adds content is an extension. The parent's content appears first ; the child's addendum follows.

**Merge places `parent` inside or at the end.** Any other position of `parent.<section>` is a merge. The child contributes content both before and after, or only before, the parent's contribution.

**The patterns work across multi-level chains.** In a three- level chain A → B → C, a section's `parent.<section>` in C returns the *most-specialized parent's* version — typically B's. To reach A's directly, use `parent.A.<section>` (the explicit-ancestor form from Chapter XX).

**Composition with `self.<section>`.** The `self.<section>` form is rarely used inside a section of the same name (`self.task` inside the child's `task` section would be a recursive reference, which the processor's PHASE 8 detects and errors on as a CIRCULAR_INCLUDE). The `self` form makes sense in *other* sections that want to retrieve the most-specialized version of a given section.

**Repeated overrides at multiple chain levels.** When more than one level of the chain overrides the same section, the rendered output reflects the *leaf's* override. Intermediate overrides are visible only if a level explicitly references them with `parent.<intermediate>.<section>`.

**Empty sections.** A child override that is empty produces an empty section in the rendered output. This is sometimes used as a « delete the parent's section » idiom — there is no explicit « delete » syntax in POO²L ; an empty override is the closest approximation. Tooling may emit a warning when an override is suspiciously empty.

**Sections vs whole-template extend.** `@@@{include parent}` (no `.<section>`) returns the *entire* parent's content — every section, in parent declaration order. The child uses it at the start to embed the whole parent before its own sections, or in a single section to wrap the parent's full body. The specification refers to this as « the child wants to embed the entire parent and add content around it » (`§5.4`).

> ***`standard.md §5.4 — parent` alone (excerpt):***
>
> *`parent` alone (without a section) includes the FULL parent content. This is useful when a child prompt wants to embed the entire parent and add content around it, rather than selectively overriding individual sections. The form returns the parent's full resolved content (every section, in parent declaration order), including content inherited from further ancestors.*

## 24.4 — Related features and why

**Inheritance (Chapter XIX).** The three patterns only exist within an inheritance chain. Without a parent, there is no `parent.<section>` to compose with.

**`parent` and `self` (Chapter XX).** The reference family that makes the patterns possible. Chapter XX catalogues every form the patterns use.

**Sections (Chapter VII).** The named units the patterns manipulate. A prompt without sections has no override/extend/ merge surface ; the patterns require Level 1 or higher.

**Includes (Chapter XVII).** The host directive of all three patterns. `@@@{include parent.<section>}` is what splices the parent's content ; without it, only override is possible.

**Member override at L5 (Chapter XXIII).** The OO² axis extends the override pattern from sections to whole templates. A collection member with id `X` in a derived collection overrides its parent collection's `X` exactly the way a child section overrides its parent's section.

**Collections (Chapter XXII).** A prompt that participates in both OO axes may use override/extend/merge for OO #1 sections *and* for OO #2 members. The two axes compose orthogonally ; the patterns are independent on each.

## 24.5 — Practical examples

**Trivial — straight override.**

```
@@@template marketing_reviewer inherits reviewer
@@@name Marketing Reviewer
@@@description Reviewer with a brand-tone format

@@@section format
```

```
Reply as a confident, plain-spoken brand voice would. Two
paragraphs maximum.
```

The parent's `format` section is replaced wholesale. The child does not need the parent's format-section content; it has its own.

**Idiomatic — extend.**

```
@@@template annotated_reviewer inherits reviewer
@@@name Annotated Reviewer
@@@description Reviewer that appends a justification rule

@@@section format
@@@include parent.format
After each bullet, append a one-sentence justification.
```

The parent's `format` content appears first; the child's addendum follows. The parent's instruction is still in force; the child sharpens it.

**Idiomatic — merge.**

```
@@@template framed_reviewer inherits reviewer
@@@name Framed Reviewer
@@@description Reviewer that frames the standard format

@@@section format
Begin every review with the line "Overall impression: <one
sentence>". Then follow the standard format below.

@@@include parent.format

Close with the line "Top three improvements: <a numbered list>".
```

The parent's `format` content is sandwiched between the child's framing. The reader sees: an opening line, the parent's instructions, a closing line.

**Real-world — multi-level merge.**

```
Chain: reviewer → code_reviewer → python_code_reviewer (current)

@@@section task
Before reviewing, note the language version in use.

@@@include parent.task

Conclude with a Python-specific note: prefer dataclasses over
NamedTuple.
```

The current prompt's `task` merges its own framing around the *middle* parent's (`code-reviewer`'s) `task` content. To reach the base reviewer's content directly, the child would write `parent.reviewer.task`.

**Anti-pattern — accidental override.**

```
@@@section format
Use a friendly tone.
```

The author intended to *extend* the parent's `format` (which specifies bullet structure, length, and so on) with a tone note. The override discarded the parent's content entirely. The fix is to use the extend pattern :

```
@@@section format
@@@include parent.format
Use a friendly tone.
```

Such accidental overrides are common ; a « diff against parent » view in tooling helps catch them.

**Anti-pattern — empty override as a delete.**

```
@@@section format
```

The child wanted to *remove* the parent's `format` section. The empty override accomplishes the goal *in the rendered output*, but the section is still declared, and downstream references (`parent.format`, `self.format`) still resolve to an empty string. If the goal is structural — a derived prompt that *truly has no format constraint* — the parent's section shouldn't exist in the first place; the right fix is upstream.

The next chapter — and the first of Part V — covers *aliases*, the mechanism that lets a prompt's `id` evolve without breaking references.

# Partie 5

*Lifecycle and governance*

# Chapter XXV — Aliases

> ***`standard.md §5.14` — Alias Syntax (excerpt):***
>
> *Format: `@@@alias <alternative_identifier>`. The alternative identifier follows the same rules as `id` (no spaces, unique in scope). Multiple alias declarations are allowed (one per line). Aliases MUST NOT conflict with any existing prompt id or other alias. Alias resolution is transparent: includes and inherits using an alias resolve identically to using the canonical id — `@@@{include code_review_v2}` resolves to the prompt whose canonical id is, say, `code_review_v3` if that prompt declares `@@@alias code_review_v2`.*
>
> ***`standard.md §4.4.7` — Alias (rationale, excerpt):***
>
> *An alias is an alternative identifier for a prompt, enabling backward compatibility during ID migrations. A prompt MAY declare one or more aliases; each is registered independently. Aliases are secondary — the canonical ID remains the primary identifier and is the only one that appears in identity metadata exports. Implementations SHOULD track alias usage and MAY emit a WARNING so teams can know when an alias can safely be removed.*

An alias is an alternative identifier for a prompt. It exists for one purpose: backward compatibility when a prompt's canonical `id` changes. References to the old `id` continue to resolve, transparently, to the new one. Without aliases, every rename would be a breaking change; with them, a rename is a non-event for the rest of the system.

## 25.1 — Syntax

```
@@@alias <alternative_identifier>
```

Rules (`§5.14`):

- `<alternative_identifier>` follows the same identifier rules as `id` (no spaces; letters, digits, underscore only — NO hyphen).
- A prompt MAY declare *multiple* aliases — one declaration per alias, repeated as many times as needed.
- Aliases MUST be unique across all templates and aliases in scope. An alias MUST NOT conflict with any prompt's primary id (this is the ALIAS_CONFLICT error from `§6.2`).

> ***`standard.md §6.2` — ALIAS_CONFLICT (excerpt):***
>
> *ALIAS_CONFLICT is an error condition: an alias conflicts with an existing prompt id or another alias. The processor maintains the alias table across the prompt's scope (group, project, or directory). An attempt to declare an alias that is already in use — either as another prompt's `id` or as another alias — raises ALIAS_CONFLICT at parsing time (Phase 1), before any compositional resolution. — Aliases are secondary: the canonical `id` remains the primary identifier.*

Aliases live at the *identity* layer, alongside the `@@@template` header (which carries the `id`), `name`, and `description`. The recommended writing order places them immediately after `description` (`§4.5`, *Standard declaration order*), so the prompt's identity-and-history block is visually clustered.

> ***`standard.md §4.5` — Standard declaration order (excerpt):***

> *POO²L generators SHOULD organize declarations in this order: (1) Identity (`@@@template` header first, then `name` and the optional `description` — at the very beginning); (2) Lifecycle (`alias`, `deprecated` — identity-related metadata); (3) Parameters; (4) Constants; (5) Content; (6) Attachments (at the end). This ensures prompt identity is immediately visible and lifecycle metadata (aliases, deprecation) follows identity. Apart from the `@@@template` header, which MUST open the entity, processors MUST accept declarations in ANY order — order has NO semantic significance for processing.*

## 25.2 — Purpose

An `id` is the prompt's stable handle. The entire compositional half of the language — includes, inherits, collection members, `^parent.<template_id>` references — refers to templates by their `id`. Renaming a prompt is therefore a breaking change unless something makes the *old name* keep working.

Aliases are that something. The prompt declares its old `id` as an alias; the rest of the system can keep using either the old or the new name; the processor builds the alias table at Phase 1 and resolves alias references to their canonical id as part of reference resolution, so a caller may use either name interchangeably.

Three benefits:

1. **Renaming is reversible cheaply.** A team that committed to `code_review_v2` six months ago can rename it to `code_reviewer` without auditing the entire library for references.
2. **Versioned migrations.** A `code_review_v2 → code_review_v3` migration declares `v2` as an alias of `v3` for the transition window, then removes the alias once callers have moved.
3. **Typo fixes that do not break callers.** A misspelled `id` can be corrected at any time: the misspelling becomes an alias of the corrected name.

The mechanism is intentionally narrow. Aliases solve renames; they do not solve « make this prompt available under several unrelated names for thesaurus reasons », and using them that way is the chapter's anti-pattern.

## 25.3 — Possibilities

**Alias table at Phase 1; resolution at reference time.** The processor builds the alias table at Phase 1 (Parsing). Resolution itself — replacing an alias with its canonical id — happens as part of reference resolution: an `inherits` parent reference is resolved at Phase 3 (inheritance), an `@@@include` target at Phase 8 (includes, where the standard lists « apply alias resolution » as the first step). Either way the lookup consults the same table, so an alias resolves identically to using the canonical id.

**Multiple aliases per prompt.** A prompt may declare any number of aliases. Each is registered independently. A migration that has gone through three names — `v1`, `v2`, `v3` — may have `v3` as the canonical id with `v1` and `v2` as aliases.

**Aliases are unique in scope.** The implementation maintains the alias table across the prompt's *scope* — typically the group, project, or directory in which the prompt lives. An attempt to declare an alias already used by another prompt (or already used as another prompt's `id`) raises ALIAS_CONFLICT.

**Aliases do *not* affect identity exports.** When the processor outputs the prompt's identity (in a metadata export, in a dashboard listing), only the canonical id appears. Aliases are an *input* convenience for references; they are not part of the prompt's *output* identity.

**Implementations may track usage.** `§4.4.7` notes that implementations SHOULD track alias usage and MAY emit a warning. A team that wants to know whether an alias is still being referenced in the wild can ask the implementation for usage statistics and decide when to drop the alias safely.

**Includes and inheritance accept aliases transparently.**

```
@@@template code_review_v3
@@@alias code_review_v2
@@@alias code_review

(elsewhere)
@@@{include code_review_v2}              → resolves to code_review_v3
@@@template child inherits code_review     → resolves to code_review_
```

Both references reach the canonical prompt. The author of the including/inheriting prompt does not need to know which name is canonical.

**Aliases are LOCAL — member override matches by canonical id.** An alias is a non-qualified, local alternative name for its own prompt. It does NOT let a derived collection's member override a parent member by the parent's *alias*: member override in the effective prompt set matches by CANONICAL id only. A member's own alias makes it addressable under that name locally; a parent's alias is invisible as an override key.

> ***`standard.md §4.6.1` — Effective prompt set keying (excerpt):***
>
> *When collection C1 inherits from C0, the effective prompt set of C1 is computed root-first and keyed by CANONICAL prompt id. A prompt in C1 with the same canonical id as one inherited from C0 OVERRIDES the inherited prompt (member override). Aliases are local and non-qualified: a member's alias makes it addressable under that name within its own scope, but a member override is never established through a parent's alias.*

**Aliases do not propagate through inheritance.** A child prompt does not automatically gain its parent's aliases. If the parent has aliases `v1` and `v2` that the child also wants to be addressable by, the child declares them itself.

**Deprecation typically pairs with aliases during a rename.** When `v2` is being phased out in favour of `v3`, the migration declaration is :

```
@@@template code_review_v3
@@@name Code Review v3
@@@alias code_review_v2
@@@description Latest reviewer — supersedes v2
```

and the *old* `v2` prompt — if it still exists — is marked deprecated with a pointer to `v3`. Aliases and deprecation (Chapter XXVI) work together ; neither replaces the other.

## 25.4 — Related features and why

**Identity (Chapter VI).** Aliases extend the prompt's identity at the input side without changing it at the output side. The canonical `id` remains the single identifier in every output channel.

**Deprecation (Chapter XXVI).** Aliases solve the *forward* problem (old references keep working) ; deprecation solves the *backward* problem (callers learn that they should move to a new name). The two combine in every real migration : the new prompt declares the old name as an alias, the old prompt's identity is marked deprecated.

**Includes (Chapter XVII) and inheritance (Chapter XIX).** Both mechanisms resolve aliases transparently. An author renaming a heavily-referenced prompt does not need to touch the callers ; the alias declaration on the renamed prompt does the work.

**Collections (Chapter XXII) and member override (Chapter XXIII).** Member override matches by CANONICAL id, never by a parent's alias. Aliases are local; the effective prompt set keys on canonical ids only.

**Versioning (Chapter XXVII).** A v2 → v3 rename is the prototypical alias use case. The versioning policy of `§7.1` makes aliases the primary mechanism for non-breaking renames.

> ***`standard.md §7.1` — Template versioning (excerpt):***
>
> *POO²L templates SHOULD support versioning with semantic versioning (MAJOR.MINOR.PATCH) and immutable versions (no in-place modification). There is NO version-reference syntax in `include` / `inherits` in v1 — a reference names a prompt, not a version. The combination with aliases is the canonical mechanism for non-breaking renames: when a prompt's `id` evolves, declaring the prior id as an alias lets the rest of the library continue to resolve references transparently.*

**Access control (Chapter XXIX).** A prompt that is reachable under several names — its canonical id and one or more aliases — is subject to the same access policy on every name. The policy keys on the canonical id internally; the implementation ensures alias paths cannot bypass the policy.

## 25.5 — Practical examples

**Trivial — a typo fix.**

```
Original prompt with the typo:
  @@@template cod_review        ← typo
  @@@name Code Review
  @@@description Reviews code

After the fix:
  @@@template code_review
```

```
  @@@name Code Review
  @@@alias cod_review
  @@@description Reviews code
```

Existing references to the misspelled `cod_review` continue to resolve to the corrected prompt. Once the team has audited and cleaned up references, the alias can be removed.

**Idiomatic — a v2 → v3 migration with multiple aliases.**

```
 @@@template code_review_v3
@@@name Code Review
@@@alias code_review_v2
@@@alias code_review
@@@description Latest reviewer; supersedes v2 and the unversioned
                original
```

Three names reach the same prompt. The team can use any of them in includes and inherits ; the canonical name is the v3 one.

**Real-world — override matches by canonical id, not by a parent's alias.**

```
 Parent collection "reviewers", member task:
  @@@template reviewers.task
  @@@alias review_task
  Body of the parent's task.

Derived collection "reviewers_code" (inherits reviewers):
  @@@template reviewers_code.task           -- overrides the parent's `t
  Body of the derived task.
```

To override the parent's `task`, the derived member must declare the CANONICAL id `task`. Declaring the terminal id `review_task` (the parent's *alias*) would NOT establish an override — aliases are local and non-qualified, and member override keys on

canonical ids only. The derived member would instead be a new, distinct member. Aliases solve renames of a prompt's own id ; they are not an override key across a collection chain.

**Anti-pattern — aliases as a synonym dictionary.**

```
@@@template customer_reply
@@@alias respond_to_customer
@@@alias answer_client
@@@alias reply_message
@@@alias customer_response
```

The aliases are not historical names — they are aspiring synonyms. The system has to keep them unique forever ; new templates cannot use any of those names ; the alias table grows without serving an actual migration. Aliases are for *real* renames ; thesaurus entries belong to documentation, not to the alias mechanism.

The next chapter covers the lifecycle marker that pairs with aliases in every real migration : *deprecation*.

# Chapter XXVI — Deprecation

> ***`standard.md §5.13` — Deprecation Syntax (excerpt):***
>
> *Format: `@@@deprecated "<migration_message>"`. The message is REQUIRED and MUST be enclosed in double quotes; it SHOULD indicate the replacement prompt or migration path. The declaration may appear anywhere in the prompt (before or after other declarations); at most ONE deprecated declaration per prompt. Processing: implementations SHOULD emit a WARNING (not error) when the deprecated prompt is instantiated, included, or inherited, and SHOULD surface the migration message in tooling/UI.*
>
> ***`standard.md §4.4.6` — Deprecation (rationale, excerpt):***
>
> *A deprecation marker is a lifecycle signal that a prompt is scheduled for removal and should be replaced by an alternative. It is purely informational — it does NOT prevent instantiation. Implementations SHOULD emit a WARNING when a deprecated prompt is used, inherited, or included by another prompt. Deprecated templates remain fully functional until removed; the warning channel — distinct from the error channel — lets callers learn about the migration without breaking the rendered output.*

A deprecation marker is a piece of metadata that announces a prompt's future removal and points callers to the replacement. It is *purely informational*: every conformant processor emits a warning when a deprecated prompt is instantiated, included, or inherited, but the prompt still renders normally. The mechanism is a *migration signal*, not an enforcement.

## 26.1 — Syntax

```
@@@deprecated "<migration_message>"
```

Rules (`§5.13`):

- The migration message is REQUIRED and MUST be enclosed in double quotes.
- The message SHOULD indicate the replacement prompt or the migration path.
- A prompt declares *at most one* `deprecated` marker.
- The declaration may appear anywhere in the prompt — before or after other declarations.

Examples (from `§5.13`):

```
@@@deprecated "Use code_review_v3 instead"
@@@deprecated "Replaced by analysis.comprehensive — removal scheduled
```

## 26.2 — Purpose

A prompt that should not be used anymore needs a way to *say so*. Three audiences benefit:

- **The human reading the prompt** sees the marker and the message in the editor. They understand at a glance that this prompt is on the way out and that there is a successor.
- **The system rendering the prompt** sees the marker at Phase 1 and emits a warning. Tooling — dashboards, audit logs, CI scripts — can surface the warning so callers learn that the prompt they relied on is scheduled for removal.
- **The author who eventually removes the prompt** has a record of the intended successor in the message itself. The decision log lives next to the decision.

Crucially, deprecation does *not* break anything. The deprecated prompt continues to render exactly as before. The warning is the *soft* enforcement ; the *hard* removal is a separate decision that happens when the team is ready.

Pairing with aliases (Chapter XXV) is universal. A rename is two moves :

1. The *new* prompt declares the *old* `id` as an alias, so old references keep working.
2. The *old* prompt — if it is still kept around as a stub — declares deprecation with a pointer to the new id.

Together these two mechanisms turn a rename into a non-event for callers while signalling the migration intent clearly.

## 26.3 — Possibilities

**Phase 1 emission.** The processor emits a DEPRECATED_TEMPLATE warning at Phase 1, as it parses the marker. The warning is on the *warning* channel, not the *error* channel — rendering continues normally.

**Three trigger paths.** A warning fires when the deprecated prompt is :

- **Instantiated.** A user or system renders the deprecated prompt directly. The deprecation message reaches the renderer alongside the rendered output.
- **Included.** Another prompt has `@@@{include <deprecated>}`. The warning fires for the *included* prompt, not the including one — the includee is what is deprecated.
- **Inherited.** Another prompt has the inline clause `@@@template child inherits <deprecated>`. Same logic — the warning is on the parent, not the child.

**Message format.** The spec does not mandate a structure for the migration message — it is free text inside double quotes. The convention that works in practice is `"<reason> — <next step>"` :

*« Use code_review_v3 instead »*, *« Replaced by analysis. comprehensive — removal scheduled 2026-06 »*. The message is shown to humans ; write it in their voice.

**Removal scheduling.** The spec does not formalize a « removal date » field. Authors who want a deadline embed it in the message. Some implementations parse simple date patterns out of the message for tooling ; the parsing is implementation-defined.

**No cascading deprecation.** A prompt that inherits from a deprecated parent is *not* itself deprecated. The warning fires on the parent at render time, but the child remains non-deprecated. To deprecate the child as well — for example because the entire chain is being retired — the child declares its own `deprecated` marker.

**Behaviour during the transition window.** Phase 1 emits the warning every time the prompt is rendered. Implementations may de-duplicate within a single rendering session (the same deprecated prompt referenced ten times produces one warning, not ten) ; the de-duplication is implementation-defined.

**Deprecation and virtual templates (Chapter XXIII).** A virtual prompt may be deprecated. The warning fires ; concrete overrides in derived collections do not inherit the deprecation. A team deprecating a virtual ancestor typically deprecates the concretes as well, in lockstep.

**Removal.** When the deprecation transition is complete, the prompt is removed entirely — its source is deleted from the library. Aliases that pointed to it (if a *replacement* prompt declared the deprecated id as an alias) cease being needed at the same time. The final cleanup is a separate authoring operation ; the language does not automate it.

## 26.4 — Related features and why

**Aliases (Chapter XXV).** Aliases and deprecation are the two halves of every migration. The alias keeps old references resolving ; the deprecation marker signals that callers should move. Use both.

**Identity (Chapter VI).** Deprecation is an *identity-layer* marker. It does not change the prompt's `id`, `name`, or `description`; it adds a piece of metadata about the identity's future.

**Inheritance (Chapter XIX) and includes (Chapter XVII).** Both trigger DEPRECATED_TEMPLATE warnings when the parent or included prompt is deprecated. Tooling typically presents the warnings *at the call site*, not at the deprecated prompt, so the caller knows which of their includes points at something on the way out.

**Versioning (Chapter XXVII).** Deprecation is the canonical mechanism for retiring an old version while a newer version takes over. The versioning policy of `§7.1` makes deprecation a recommended step in the v1 → v2 transition.

> ***`standard.md §7.1` — Template versioning (excerpt) :***
>
> *POO²L templates SHOULD support versioning with semantic versioning (MAJOR.MINOR.PATCH) and immutable versions (no in-place modification). There is NO version-reference syntax in `include` / `inherits` in v1. Deprecation pairs naturally with the versioning policy : when a new version supersedes an older one, the older prompt's `@@@deprecated` marker tells callers that the prompt is scheduled for removal and points them to the replacement, while immutability preserves the older version's content for any caller still referencing it.*

**Dependency management (Chapter XXVIII).** Implementations that track inter-template dependencies SHOULD surface deprecated ancestors and dependencies as part of their audit output. A team that wants to clean up a library can ask « show me every prompt that depends on something deprecated » and act on the answer.

**Access control (Chapter XXIX).** Deprecation does not affect permissions. A deprecated prompt with read access is still readable; a deprecated prompt the caller cannot access is still inaccessible. The migration message is shown to whoever has the access to render the prompt.

## 26.5 — Practical examples

**Trivial — deprecate with a pointer to a successor.**

```
@@@template code_review_v2
@@@name Code Review v2
@@@description Older code_reviewer, kept for backward compatibility
@@@deprecated "Use code_review_v3 instead"

(rest of the prompt unchanged)
```

Every render emits a DEPRECATED_TEMPLATE warning with the message; the prompt still functions normally. Callers that still reference `code_review_v2` continue to work; they also see the warning and learn about v3.

**Idiomatic — deprecation with a removal date.**

```
@@@template analysis_legacy
@@@name Analysis (legacy)
@@@description Legacy analysis prompt
@@@deprecated "Replaced by analysis.comprehensive —
                  removal scheduled 2026-06"
```

The migration window is explicit. Tooling can sort templates by deprecation deadline; teams can schedule the removal sprint.

**Real-world — deprecation paired with alias on the successor.**

The *old* prompt:

```
@@@template code_review_v2
@@@name Code Review v2
@@@description Older reviewer
@@@deprecated "Use code_review_v3 instead"
```

The *new* prompt :

```
@@@template code_review_v3
@@@name Code Review v3
@@@alias code_review_v2
@@@description Latest reviewer
```

Callers can use either `code_review_v2` or `code_review_v3` interchangeably (the alias resolves to v3) ; calls that still name v2 produce a deprecation warning (because the alias resolution does not suppress it — the *named-as-v2* alias of v3 is *itself* a deprecated reference, on the assumption that the v2 stub is still around).

If the team has *already removed* the v2 stub and kept only v3 with the alias, the warning channel goes quiet — there is no `deprecated` marker on v3 because v3 is not deprecated. The alias by itself does not trigger a warning.

**Anti-pattern — deprecation as a comment.**

```
@@@template code_review_v2
@@@-- DEPRECATED: please migrate to code_review_v3

@@@name Code Review v2
...
```

The information lives in a comment that the tooling cannot read. No warnings fire ; the dashboards cannot list « deprecated templates » ; CI cannot fail builds that still reference deprecated templates. The fix is to use the proper marker :

```
@@@deprecated "Use code_review_v3 instead"
```

The next chapter covers the *versioning policy* — the guidance the specification gives implementations on tracking prompt evolution over time.

# Chapter XXVII — Versioning Policy

> ***`standard.md §7.1` — Template Versioning (excerpt) :***
>
> *POO²L templates SHOULD support versioning with : semantic versioning ( `MAJOR.MINOR.PATCH` ) ; immutable versions (no in-place modification). Both clauses are SHOULD-level recommendations — the language gives the vocabulary but defers the storage model and the policy for when a `PATCH` becomes a `MINOR` to the implementation. There is NO version-reference syntax in `include` / `inherits` in v1 : a reference names a prompt, never a specific version. (Version-pinned references are a roadmap item, not part of v1.)*

POO²L describes a *language*, not a storage system. The versioning policy of `§7.1` is correspondingly short : two SHOULD-level recommendations about how implementations are expected to track a prompt's evolution over time. This chapter unpacks the two recommendations, explains why the language stops where it does, and covers the authoring patterns that make versioning practical at L4.

## 27.1 — The two recommendations

`§7.1` reads, in full :

> *POO²L templates SHOULD support versioning with : — Semantic versioning (MAJOR.MINOR.PATCH) — Immutable versions (no in-place modification)*

Two pieces :

1. **Semantic versioning.** A prompt's history is a sequence of versions identified by a `MAJOR.MINOR.PATCH` triple.
2. **Immutable versions.** Once a version is published, its content does not change. New work produces a new version.

Neither is mandated as a normative requirement — both use SHOULD. Implementations are expected to follow the spirit ; they may diverge on the details (the mechanism of storage, the policy for when a `PATCH` becomes a `MINOR` ).

**There is no version-reference syntax in v1.** A `@@@include` or an `inherits` clause names a *prompt*, not a specific version — POO²L v1 has no `@1.2.0` pin. Version-pinned references are a roadmap item, not part of the language ; a reference always resolves to the target as the implementation currently stores it.

## 27.2 — Why the spec stops here

Versioning is fundamentally an *application-level* concern. The language describes how a *single* prompt's source is parsed and rendered ; it does not describe how multiple versions of that source are stored, served, or selected. Pushing versioning into the language would mean defining the storage model — which is the opposite of the closed-extension policy's intent. Implementations are free to use Git, a database, a flat-file scheme, or anything else.

What the language *does* provide is the *vocabulary* that versioning needs :

- **Identity** (Chapter VI) — the stable handle that survives across versions.
- **Aliases** (Chapter XXV) — the mechanism for the v1 → v2 rename that does not break callers.
- **Deprecation** (Chapter XXVI) — the marker for the v1 retirement message.

The combination of these constructs is sufficient to express most versioning policies. The remaining choices — semver vs date vs sequential number, immutable vs mutable, branch-based vs tag-based — are implementation policy. Note that pinning a reference to a specific version is NOT among the language's constructs in v1.

## 27.3 — Semantic versioning in POO²L

The recommended scheme is the standard `MAJOR.MINOR.PATCH` discipline familiar from software :

- **MAJOR** increments on breaking changes — a parameter removed, a constraint tightened, a section renamed.
- **MINOR** increments on additive changes — a new optional parameter, a new section, a clearer description.
- **PATCH** increments on cosmetic or behavioural fixes that do not change the contract — a typo correction, a clarification.

The spec does not formalize what counts as « breaking » — that is the implementation's audit responsibility. The practical rule of thumb : a change is breaking if a caller that was correct against the previous version is no longer correct against the new one.

## 27.4 — Immutable versions

A published version's source does not change. Edits produce a new version. This is the same discipline used in semver-published libraries, npm packages, etc.

Two consequences :

- **Audit.** Every render is reproducible from the version identifier alone. Running the prompt next year produces the same text it produced today.
- **Trust in references.** A child prompt that pins to a parent version trusts that version's contract not to change behind its back. Without immutability, the language's compositional guarantees would all be on shifting sand.

Implementations may *delete* old versions (a retention policy), but they may not *modify* them in place. Deletion is a destructive operation ; modification is the dangerous one.

## 27.5 — References always resolve to the current target (no pinning in v1)

POO²L v1 has NO version-reference syntax. A reference — whether an `inherits` clause or an `@@@include` — names a *prompt*, and resolves to that prompt as the implementation currently stores it :

```
@@@template child inherits code_reviewer      -- resolves to code_rev
@@@include glossary                          -- resolves to glossary
```

There is no `code_reviewer@1.2.0` pin in the language. Every reference is « unpinned » by construction : upstream improvements to the target propagate to callers automatically, and so do upstream breakages. Teams that need the stability of a frozen dependency get it at the *storage* layer, not in the prompt source — by keeping the depended-on prompt itself immutable and evolving successors under new ids (with aliases and deprecation for the transition).

Version-pinned references (a `@1.2.0`-style suffix) are a **roadmap item**, tracked for a future revision ; they are not part of v1 and no conformant v1 implementation accepts such syntax.

## 27.6 — Possibilities

**Where versioning lives.** The spec leaves storage to the implementation. IziBuddy stores versions in a sidecar metadata record per prompt and (optionally) mirrors them to Git for audit. Other implementations may use a database with a `version` column, a flat-file naming scheme, or any other mechanism.

**Reading a specific version from tooling.** Implementations SHOULD provide a way to render a specific version, not just the latest. The reference implementation surfaces this in the dashboard's version history.

**Diff between versions.** Implementations SHOULD make version-to-version diffs available. The granularity is the implementation's choice — line-level, section-level, or declaration-level diffs are all useful, and many tools provide all three.

**Rollback.** Rolling back to a previous version produces a *new version* with the content of the old one. The previous *latest* remains in history ; it is not overwritten. The discipline keeps the audit trail intact.

**Deprecation across versions.** A deprecation marker is per-prompt, not per-version. A team can deprecate the entire prompt identity (signalling « this prompt is going away, regardless of version ») by adding the marker to the latest version ; the marker's effect is on every future render.

**Branching.** The spec does not address branching. An implementation may support parallel branches (a `feature/foo` branch alongside `main` ) ; the language does not standardize them because they are storage-level, not language-level.

## 27.7 — Related features and why

**Identity (Chapter VI).** The `id` is the version-invariant handle. Versions belong to a prompt identity ; the `id` does not change as the version changes.

**Aliases (Chapter XXV).** Aliases bridge identity changes *across* the language's version-tracking machinery. A renamed prompt declares its old `id` as an alias, regardless of which version of the new prompt is being rendered.

**Deprecation (Chapter XXVI).** A deprecation marker on the latest version effectively deprecates the entire prompt — every future render sees the warning. To deprecate one specific older version (rare), the marker lives on that version's source.

**Dependency management (Chapter XXVIII).** The dependency tracking of `§7.2` operates at the version level. Implementations SHOULD track which version of a parent or included prompt each caller depends on, so the audit can answer « what calls `code_review_v2` 1.2.0 ? »

> ***`standard.md §7.2` — Dependency Management (excerpt) :***
>
> *Implementations SHOULD track : prompt dependencies (includes, inheritance) ; dependency versions ; breaking changes across versions ; attachment dependencies and versions. The clauses are SHOULD-level — version-aware dependency tracking is recommended, not mandated.*

**Access control (Chapter XXIX).** Permissions usually apply per-template-identity, not per-version. An implementation MAY have version-level permissions if it has a use case (a private beta of v3 visible only to some users) ; the spec does not require it.

## 27.8 — Practical examples

**Idiomatic — a plain include resolves to the current target.**

```
@@@template quarterly_report
@@@name Quarterly Report
@@@description Q4 report draft

@@@section (glossary_block)
@@@include glossary
```

```
@@@section findings
...
```

The report includes `glossary` as the implementation currently stores it — there is no version pin in the language. If the team needs the report frozen against a specific glossary content, they freeze the glossary prompt itself (immutability) or point the report at a successor id ; the prompt source carries no `@version` suffix.

**Idiomatic — an inherits resolves to the current parent.**

```
@@@template python_reviewer inherits code_reviewer
@@@name Python Reviewer
@@@description Specialization of code_reviewer for Python

@@@section task
Review Python code with attention to type hints.
```

The Python reviewer inherits from whatever the latest version of `code_reviewer` is. Upstream improvements to `code_reviewer` propagate automatically. The trade-off is risk : an upstream breakage does too.

**Real-world — a v2 → v3 migration with aliases and deprecation.**

```
v2 prompt (latest known version, kept as a stub):
  @@@template code_review_v2
  @@@name Code Review v2
  @@@deprecated "Use code_review_v3 instead — removal 2027-01"
  ...

v3 prompt:
  @@@template code_review_v3
  @@@name Code Review v3
  @@@alias code_review_v2
  ...
```

Callers using either id continue to work; the alias resolves to v3, the deprecated v2 stub fires its warning. Once the team has finished the migration, the v2 stub is deleted (a single authoring operation); the alias on v3 remains until the team chooses to remove it.

**Anti-pattern — modifying a published version in place.**

A team edits an existing version's content directly, without incrementing the version number. Existing callers that relied on the previous behaviour now silently see different output. The audit trail is gone. The fix is procedural: every edit to a published version produces a new version, period.

The next chapter covers *dependency management* — the audit machinery that complements versioning.

# Chapter XXVIII — Dependency Management

> ***`standard.md §7.2` — Dependency Management (excerpt):***
>
> *Implementations SHOULD track: prompt dependencies (includes, inheritance); dependency versions; breaking changes across versions; attachment dependencies and versions. None of the four is mandated — every clause is SHOULD-level. The language supplies the dependency-producing constructs (`inherits`, `include`, `fixed file` attachment paths, `fixed record` attachment refs); the implementation supplies the tracking machinery, the reverse-dependency view, and any breaking-change tooling.*

A POO²L library is a graph. Templates include other templates; templates inherit from other templates; collections inherit from other collections; attachments live in storage that may be shared. Dependency management is the discipline of *seeing the graph* — knowing what depends on what, what would break if a target disappeared, and what is downstream of any given change.

The specification's `§7.2` is short. This chapter unpacks the recommendations and explains the patterns implementations have adopted in practice.

## 28.1 — What the spec recommends

`§7.2`, in full:

> *Implementations SHOULD track:* — *Template dependencies (includes, inheritance)* — *Dependency versions* — *Breaking changes across versions* — *Attachment dependencies and versions*

Four things to know:

1. **The dependency graph itself.** For each prompt, what does it reference? An implementation SHOULD be able to answer: « show me everything `code_reviewer` inherits, includes, or refers to via `^parent.<template_id>`. »

2. **The version each dependency uses.** Pinned references name a version; unpinned references resolve to latest. The dependency record tracks which.

3. **Breaking changes.** When the dependency graph contains a reference to a version that no longer exists or that has been superseded with a breaking change, the implementation SHOULD surface it.

4. **Attachments.** Files referenced by `fixed file` attachments and bytes referenced by `fixed record` attachments are also dependencies. A prompt that references `docs/manual.pdf` depends on that file existing in the storage backend; if it moves, the prompt breaks. A `record` ref depends on the store entry; if the entry is GC'd, the prompt breaks.

None of the four is *mandated* — every clause uses SHOULD. The language gives the vocabulary; the *implementation* gives the machinery.

## 28.2 — Why this matters

A POO²L library at L4 with collection inheritance and many includes is a non-trivial graph. The graph determines what an author can change without breaking callers, what an audit can trust, and what an automated test can cover.

Without dependency tracking, every change is a leap of faith : *who else uses this ?* The author guesses, edits, and waits to find out. With dependency tracking, the same question has an answer before the edit happens.

Three operations that dependency management makes practical :

- **Impact analysis.** Before bumping `code_reviewer` from 1.2 to 2.0, list every prompt that inherits from it and every caller that includes it.
- **Cleanup audits.** Find every prompt that depends on a deprecated ancestor. The list is the migration backlog.
- **Reverse navigation.** Given a leaf prompt that misbehaves, trace upward through its dependencies to find which inherited section actually produced the problem.

## 28.3 — Possibilities

**The graph nodes.** Each node is a prompt (or, at L4, a collection). The edges are typed :

- `inherits` (template-level OO #1).
- `inherits` (collection-level OO #2, the `inherits` clause on the `@@@collection` header).
- `include` (any of the include forms — plain, parameterized, `parent` / `self` ).
- `attachment : <type> fixed file` (a path to a file in storage) or `attachment : <type> fixed record` (a ref into the external store).

A complete dependency record per prompt lists every outgoing edge with the version (pinned or latest-at-render-time) and the edge type. Reversing the records yields the *reverse-dependency* view : who depends on this prompt.

**Phase relationship.** Dependencies are *resolved* during Phase 1 (aliases) and Phase 2–6 (collection chain, inheritance, includes). The processor can emit a dependency record as a *side product* of rendering ; many implementations cache the record so subsequent queries do not require a re-render.

**Snapshot-time vs render-time dependencies.** An *unpinned* reference's resolved version depends on when the render happens — a later render may resolve to a newer version of the target. Implementations that want a static dependency record snapshot the resolution at a specific moment (typically at save time).

**Cross-axis dependencies.** A prompt that participates in both OO axes has dependencies on both. Its template-axis parent and its collection-axis parent are both edges in the graph. The graph remains acyclic at each axis individually ; cross-axis cycles are possible if (and only if) the language allowed them, which it does not (the two `inherits` keywords are distinct).

**Attachment dependencies.** A `fixed file` attachment depends on the referenced path existing. A `fixed blob` (inline data URI) is self-contained and has no external dependency. A `fixed record` attachment depends on the ref resolving through the configured store. A `to-include` attachment has no dependency in the prompt source — the attachment slot is filled at instantiation.

**Tooling.** Implementations may provide :

- A dependency-graph viewer (the IziBuddy *inheritance chain* view is a slice of this).
- A reverse-dependency search (« who includes this glossary ? »).
- An automated breaking-change check (« the parameter `tone` was removed in v3 — list every caller pinned to ≥3.0 »).

- A pre-deletion audit (« you are about to delete `safety_preamble` ; the following 14 templates depend on it »).

The richness of the tooling is part of the implementation's value proposition ; the language does not standardize it.

## 28.4 — Related features and why

**Includes (Chapter XVII) and inheritance (Chapter XIX).** Both produce dependency edges. The dependency graph is, at its heart, the union of every prompt's `inherits` and `include` edges.

**Aliases (Chapter XXV).** Aliases complicate the graph : a prompt that references `code_review` may actually depend on the prompt whose canonical id is `code_review_v3` . The implementation's dependency record keys on canonical ids, resolving aliases first.

**Versioning (Chapter XXVII).** Dependency records carry version information. A library that audits its dependencies typically sorts them by version freshness — the « you have callers pinned to ancient versions » report.

**Deprecation (Chapter XXVI).** A deprecated dependency is a candidate for migration. Tooling typically combines deprecation warnings with reverse-dependency lookups to produce the migration backlog.

**Collections (Chapters XXII, XXIII).** Collection-axis edges are part of the graph. A derived collection depends on its parent ; a virtual prompt's concrete implementations depend on the virtual.

**Access control (Chapter XXIX).** Dependency tracking and access control interact : a prompt that depends on a target the caller cannot access still has the dependency, but the caller cannot render the parent's content. Tooling may surface the permission issue as part of the dependency view.

## 28.5 — Practical examples

**A small library's dependency graph.**

```
 safety_preamble (no dependencies)
glossary (no dependencies)

code_reviewer
  inherits: reviewer
  includes: safety_preamble, glossary

python_code_reviewer
  inherits: code_reviewer

prose_reviewer
  inherits: reviewer
  includes: glossary

Reverse view:
  reviewer ← code_reviewer, prose_reviewer
  code_reviewer ← python_code_reviewer
  safety_preamble ← code_reviewer
  glossary ← code_reviewer, prose_reviewer
```

Six templates, eight edges. Tooling can render this as a graph ; a CLI can answer « who depends on safety_preamble ? » (« code_reviewer ») ; an automated test can verify that every listed dependency resolves.

**Impact analysis before a parameter rename.**

The team wants to rename the `topic` parameter of `reviewer` to `subject`. The dependency tooling reports :

- 4 templates inherit from `reviewer` directly.
- 12 templates use `@@@{include reviewer.role}` (no parameter override ; the rename is safe for them).

- 3 templates use `@@@{include reviewer(topic=...)}` with override — these will break.

The team renames `topic` to `subject` in `reviewer`, declares an alias on the parameter (the spec does not formalize parameter aliases; some implementations support them as an extension), and updates the 3 callers in the same release.

**Reverse navigation for a render error.**

A user reports that `python_code_reviewer` produces a missing-parameter error. Tooling traces the dependency chain:

```
python_code_reviewer
  ← inherits code_reviewer
    ← inherits reviewer (latest, 2.0.0)
      → parameter `topic` is REQUIRED with no default
```

The error originates in `reviewer` 2.0.0, which made `topic` required. The fix is either to make `topic` optional in `reviewer` 2.0.1 (a non-breaking change) or to bind it in `python_code_reviewer` directly.

The next chapter covers *access control* — the third of the three governance topics from `§7`.

# Chapter XXIX — Access Control Across Groups

> ***`standard.md §7.3` — Access Control (excerpt):***
>
> *Collections MAY enforce: read/write permissions; inheritance restrictions; include scope limitations; attachment access restrictions. All four are MAY-level — an implementation without group-scoped access control is conformant. The language describes what could be enforced (the four boundary types attached to the collection scope); it does not mandate enforcement nor specify a permission model.*

Access control is the third of the three governance topics in `§7`. The specification's treatment is brief — three sentences — and intentionally narrow: the language defines what access control *applies to*, not how access control *works*. The implementation owns the policy, the storage, and the enforcement. This chapter explains what POO²L gives implementations to work with and how authoring patterns interact with access boundaries.

## 29.1 — What the spec says

`§7.3`, in full:

> *Collections MAY enforce: — Read/write permissions — Inheritance restrictions — Include scope limitations — Attachment access restrictions*

Four MAY-level recommendations :

1. **Read/write permissions** — who may *see* and who may *modify* a prompt.
2. **Inheritance restrictions** — whether a prompt outside a given scope may name `<restricted>` in the inline `inherits` clause on its `@@@template` header.
3. **Include scope limitations** — whether a prompt outside a given scope may declare `@@@include <restricted>`.
4. **Attachment access restrictions** — whether a caller may read an attachment associated with a prompt.

The MAY framing matters : an implementation that does not need group-scoped access control is conformant without these features. The language describes what *could* be enforced ; it does not mandate enforcement.

## 29.2 — Why the spec stops here

Access control is application-level by design. The spec's job is to describe how text becomes rendered text — a per-template operation that knows nothing about *who* is doing the rendering. The *who* lives in the implementation : which user is logged in, which group they belong to, which permissions the policy applies.

Pushing access control into the language would mean defining the permission model, the identity model, the group model. None of those are *language* concerns — they are infrastructure concerns the implementation owns.

What the language does provide is the *vocabulary* that access control attaches to :

- **Identity** (Chapter VI) — the canonical handle the policy applies to.
- **Collections** (Chapter XXII) — the scope the policy typically groups by.

- **Cross-scope references** — the boundary the policy decides whether to allow.

## 29.3 — Possibilities

**The reference implementation: IziBuddy groups.** IziBuddy's group model — users belong to one or more named groups, each with role-based membership — maps directly onto the four `§7.3` axes:

- **Read/write permissions** are role-based at the group level: *admin*, *editor*, *viewer*. A viewer may render group templates; an editor may modify them; an admin may add members and change roles.

- **Inheritance restrictions.** A prompt in one group may inherit from a prompt in another group only if the caller has read access to the parent. The check happens at Phase 3 — unresolvable parent (because of access denial) raises UNDEFINED_REFERENCE from the caller's perspective.

- **Include scope limitations.** Same as inheritance: an include of a prompt the caller cannot read raises UNDEFINED_REFERENCE.

- **Attachment access restrictions.** A `fixed` attachment is bound at render time using the caller's permissions; an attachment the caller cannot read causes the render to fail (ATTACHMENT_NOT_FOUND in Phase 7, or a more specific permission error in some implementations).

Other implementations are free to choose a different policy model. The language places no constraints on what the model must look like, only on what kinds of boundary it may enforce.

**Aliases respect the policy.** A caller cannot bypass the policy by referencing a target through its alias rather than its canonical id. The implementation resolves the alias in Phase 1; the permission check happens on the *resolved* target.

**Cross-axis interactions.** The two OO axes do not bypass the policy. A prompt inheriting from a parent across a group boundary triggers the same access check ; a member of a derived collection that references `^parent.<template_id>` triggers the check against the parent collection's scope.

**Rendering as the rights check.** A practical pattern : a prompt is *readable* if and only if it can be *rendered*. A caller without rights to a parent's content sees an error when trying to render the child ; the child does not silently fall back to partial output. The choice keeps the rendering behaviour deterministic across access policies.

**Snapshot of who has what.** Implementations SHOULD provide a way to audit who has what access. Auditing is itself part of governance — the team must be able to answer « who can read this prompt ? » as easily as « what does this prompt depend on ? »

**Static vs dynamic policies.** A static policy assigns permissions once and enforces them on every render. A dynamic policy may grant temporary access (timed shares, reviewer mode, audit windows). The language has no opinion ; both work within the four `§7.3` axes.

## 29.4 — Related features and why

**Collections (Chapter XXII).** Collections are the typical *unit* of access control. A team's templates live in a collection ; the team's group has permissions on that collection ; outside the group, the templates are not visible.

**Identity (Chapter VI).** The policy keys on the canonical `id`. Aliases redirect at Phase 1 to the canonical id ; the permission check then runs against that.

**Inheritance (Chapter XIX) and includes (Chapter XVII).** Cross-scope references are the bandwidth across which policy must be enforced. A child in group A inheriting from a parent in group B requires the policy to decide.

**Attachments (Chapter XVI).** A `fixed` attachment's path exists in the implementation's storage ; the storage layer typically has its own access controls that must align with the template-level policy.

**Dependency management (Chapter XXVIII).** The dependency graph interacts with access control : a reverse-dependency lookup that crosses a group boundary may legitimately be empty from the caller's perspective even though edges exist. Tooling should surface « you have no access to see who depends on this » rather than report a misleading zero.

**Versioning (Chapter XXVII).** The policy usually applies per-template-identity, not per-version. A team that needs version- level access (a private beta visible only to some users) is asking the implementation for an extension.

## 29.5 — Practical examples

**Trivial — a single-group library.**

A solo author has all their templates in one personal group. No cross-group references, no permission decisions. The implementation's access control is in effect but never triggered.

**Idiomatic — shared library across two groups.**

```
Group "shared", collection "common":
  - templates: safety_preamble, glossary

Group "team-a", collection "reviewers":
  - templates: code_reviewer, prose_reviewer

Permissions:
  shared       — readable by everyone in any team-a or team-b group
  team-a       — readable and writable by team-a members only
```

A `code_reviewer` in `team-a` may include `safety_preamble` from `shared` because team-a has read access. A user in team-b who has no access to team-a cannot render `code_reviewer` (UNDEFINED_REFERENCE in Phase 3/8 from their perspective).

**Real-world — an inheritance restriction across an internal boundary.**

```
Group "platform", collection "base":
  - prompt: standard_role
  - permissions: read-only outside platform

Group "growth", attempted prompt:
  @@@template growth_role inherits platform/base/standard_role
```

The implementation's policy says any team may *read* a platform/base prompt, but only platform may *modify*. Inherits from `standard_role` succeeds — the growth-team prompt inherits the role's contract. The growth team cannot fork `standard_role` into their own scope; doing so would require write access in the platform group.

**Real-world — an attachment behind a permission boundary.**

```
Group "legal", collection "contracts":
  - prompt: contract_reviewer
    - attachment: contract_clauses_2027 (file / blob / record — bound at
  - permissions: render-only by legal, no external rendering

User outside the legal group attempts to render
contract_reviewer.
```

The render fails: the attachment is not readable by the caller. The implementation surfaces a clear permission error (implementation-defined; some raise a dedicated PERMISSION_DENIED warning, some return ATTACHMENT_NOT_FOUND).

**Anti-pattern — assuming the language enforces a policy that the implementation does not.**

A team writes a `@@@deprecated "Internal only — do not share externally"` marker hoping that the language will hide the prompt outside their group. The marker is purely informational (Chapter XXVI); it has no permission effect. External callers who have read access to the prompt render it normally, see the deprecation warning, and proceed.

The fix is to use the *implementation's* permission system — move the prompt into a group with appropriate access controls — rather than relying on a comment-like marker to enforce policy.

The next chapter opens Part VI with the most detailed view of the *resolution phases* a reader has seen so far: a phase-by- phase reference for implementers.

# Partie 6

*Reference*

# Chapter XXX — Resolution Phases in Depth

> ***`standard.md §6.1` — Resolution Phases (excerpt):***
>
> *POO²L processors SHALL execute eleven phases in order: (1) PARSING — tokenize, build AST, extract inline declarations, register aliases, check deprecation, validate static syntax; (2) COLLECTION CONTEXT APPLICATION — resolve collection inheritance chain, compute effective prompt set, validate virtuals, merge collection-level configurations; (3) TEMPLATE INHERITANCE RESOLUTION — merge parent sections and L3 data layer, apply overrides (3.b validates the call contract); (4) SECTION SELECTION — apply the `?/!` toggle map, mark OFF sections inert; (5) PARAMETER BINDING — constants are resolved at the start of this phase; (6) CONSTANT RESOLUTION; (7) ATTACHMENT BINDING; (8) INCLUDE RESOLUTION — resolve external references, validate interfaces; (9) PARAMETER SUBSTITUTION — substitute values, compute the required-parameter set, validate constraints; (10) ASSERTION EVALUATION — validate scope nesting, evaluate `@@@assert` predicates against rendered content, block on failure; (11) OUTPUT GENERATION — assemble final text, strip every directive, emit the structure as markdown headings (`##` section, `### <n>.` sub-section; the prompt itself contributes none). The reference implementation delegates rendering to the Rust POO²L engine (`src/engine/crates/pool-core`), consumed through its bindings.*

Chapter III introduced the eleven phases as a reading map. This chapter revisits them at the level an implementer needs: inputs, outputs, errors, and the operational details that turn the spec's

prose into running code. The audience is the engineer building (or auditing) a POO²L processor; the prose stays close to the language the spec uses.

## 30.1 — Phase 1 — Parsing

**Inputs**: raw prompt source (a string), the source's storage identifier, the alias table for the current scope, the deprecated prompt registry.

**Outputs**: a parsed tree of directive nodes and text nodes; an alias-resolved canonical id; a (possibly empty) list of deprecation warnings; the prompt's declared metadata (identity, parameters, constants, attachments).

**Operations**:

- Tokenize: scan for `@@@=` and `` `` boundaries; build a token stream of directives interleaved with literal text.
- Identify the keyword inside each directive. Reject unknown keywords (INVALID_SYNTAX). Reject reserved keywords used as identifiers (INVALID_SYNTAX).
- Track the depth of the self-nesting block constructs (block comments `@@@comment ... @@@end-comment` and verbatim blocks `@@@verbatim ... @@@end-verbatim`). The line and bounded comment forms (`@@@-- ...`, `@@@(-- ... )`) do not nest. The nesting prohibition (`§5.1.3`) forbids the body of a parsed bounded construct from containing the sigil `@@@`.
- Build the abstract syntax tree.
- Extract inline declarations: parameter declarations, constant declarations, attachment declarations, identity declarations. These are not part of the rendered body but are needed by later phases.
- Register aliases and detect local conflicts. The EFFECTIVE rewrite of an alias reference to its canonical id happens later, during include resolution (Phase 8), where the registry is fully

known; Phase 1 only records the alias table for every `inherits`, `include`, and `collection` reference.

- Check deprecation: every reference whose target carries a `@@@deprecated "..."` marker generates a DEPRECATED_TEMPLATE warning. The current prompt's own deprecation marker also generates one.
- Detect ALIAS_CONFLICT when an alias clashes with an existing prompt id or alias in scope.
- Validate section and template-header declarations: a `@@@section` line — or a `@@@template` header — bearing both `virtual` and `final` raises INVALID_SYNTAX, as does a `virtual` / `final` modifier on a bare (unqualified) `@@@template`; a `virtual` section followed by non-empty content before the next section marker (anything other than whitespace and comments) raises VIRTUAL_SECTION_HAS_BODY, and a `virtual` prompt carrying any rendered body raises VIRTUAL_TEMPLATE_HAS_BODY.

> ***`standard.md §5.1.3` — Nesting prohibition (excerpt):***
>
> *The body of a parsed bounded construct MUST NOT contain the sigil `@@@`, and bounded constructs MUST NOT be nested. The opaque bodies (`verbatim`, `comment`, and the `--` bounded comment) are the sole exception: there `@@@` is inert. The self-nesting block constructs track depth internally — block comments `@@@comment ... @@@end-comment` and verbatim blocks `@@@verbatim ... @@@end-verbatim`; the line (`@@@-- ...`) and bounded (`@@@(-- ... )`) comment forms do not nest. Reserved keywords MUST NOT be used as identifiers; doing so is an ERROR.*

**Errors raised**: `INVALID_SYNTAX`, `ALIAS_CONFLICT`, `INVALID_SYNTAX`, `VIRTUAL_SECTION_HAS_BODY`. **Warnings raised**: `DEPRECATED_TEMPLATE`.

## 30.2 — Phase 2 — Collection Context Application

**Inputs**: the parsed tree, the current prompt's collection membership declaration (if any), and the `@@@collection` blocks that declare the collections in the chain (their `inherits` clauses, members, and collection-level parameter sets).

**Outputs**: the effective prompt set of the leaf collection (if applicable), the merged collection-level configurations, a validation that all virtual templates have non-virtual implementations.

> ***`standard.md §4.6.1` — Template Collections (excerpt):***
>
> *A prompt collection is the OO² (horizontal-axis) entity of POO²L. Its methods are the templates that belong to it. A collection MAY inherit from another collection (the inline `inherits` clause on its `@@@collection` header), declare abstract (virtual) prompt members (the `virtual` modifier on a member's `@@@template` header) that derived collections MUST implement, and a prompt rendered inside a derived collection MAY reference the parent collection's templates via the collection-axis reference `^parent.<template_id>`. Collections NEST by containment (unbounded depth). A collection is opened by a `@@@collection <name>` block listing its public members; a prompt belongs to it by qualifying its header id (`@@@template <name>.<id>`).*

**Operations** (collection case, `§4.6.1` and `§6.1` PHASE 2):

- **2.a.** Walk the collection inheritance chain root-first: start at the leaf, follow each collection's inline `inherits` clause until reaching a root (a collection without `inherits`). Detect cycles (CIRCULAR_COLLECTION_INHERITANCE). Reject inheritance from a folder (UNKNOWN_COLLECTION). Reject references to non-existent collections (UNDEFINED_REFERENCE).
- **2.b.** Compute the effective prompt set: union of member templates along the chain, keyed by canonical prompt id, with

member override applied in favour of the most-derived collection. Aliases are resolved. A descendant member whose id matches a prompt whose `@@@template` header carries the `final` modifier upstream in the chain raises OVERRIDE_OF_FINAL_TEMPLATE.

- **2.c.** Validate virtuals : every id whose `@@@template` header carries the `virtual` modifier somewhere in the chain must have a non-virtual concrete prompt with that id in the effective set. Raise UNIMPLEMENTED_VIRTUAL_TEMPLATE for the first violation the current render would touch.
- **2.d.** Apply collection-level configurations : parameter sets, constraints, permissions merged root-first along the chain.

For a prompt whose `@@@template` header is unqualified (it belongs to no collection) : steps 2.a–2.c are skipped (a prompt outside any collection has no OO² chain) ; its physical folder, if any, carries no compositional semantics.

**Errors raised** : `CIRCULAR_COLLECTION_INHERITANCE`, `UNKNOWN_COLLECTION`, `UNIMPLEMENTED_VIRTUAL_TEMPLATE`, `OVERRIDE_OF_FINAL_TEMPLATE`, `UNDEFINED_REFERENCE`.

## 30.3 — Phase 3 — Template Inheritance Resolution

**Inputs** : the parsed tree, the parent prompt's parsed tree (transitively, via the inline `inherits` clause).

**Outputs** : a flat effective definition — the OO #1 chain has been collapsed : merged sections plus the merged L3 data layer.

**Operations** ( `§6.1` PHASE 3) :

- Walk the inline `inherits <parent>` chain (declared on the `@@@template` header) root-first. Detect cycles (CIRCULAR_INHERITANCE). Detect references to non-existent parents (UNDEFINED_REFERENCE).

- Merge the parent's COMPOSITIONAL definition into the child: its sections and its L3 data layer (parameters, constants, embedded parameter sets, attachments). Identity, `@@@alias`, and `@@@deprecated` are LOCAL and never merged.
- Merge sections by name. A child's section with the same name as a parent's section *replaces* the parent's content for that section. New child sections are *appended* in their declaration order after all inherited sections (unless an explicit placement clause relocates them).
- Match sections (and dotted sub-sections) by name. A `@@@section steps.one` in the parent is matched by a `@@@section steps.one` in the child; `steps.two` is a different sub-section. (There is no `[N]` numbering.)
- Honour section-level inheritance modifiers. A descendant section declaration whose name matches a section marked `final` somewhere upstream in the chain raises OVERRIDE_OF_FINAL_SECTION — even if the descendant only changes the toggle or the description, with no body change. After all overrides are applied, if any section in the merged set is still `virtual` (no concrete descendant override anywhere in the chain), rendering raises UNIMPLEMENTED_VIRTUAL_SECTION. The effective modifier, toggle, description, and body of each section come from the *deepest non-virtual* declaration along the chain.

Phase 3.b then validates the **call contract** on every public override — section or prompt — once the effective signatures exist.

**Errors raised**: `CIRCULAR_INHERITANCE`, `UNDEFINED_REFERENCE`, `OVERRIDE_OF_FINAL_SECTION`, `UNIMPLEMENTED_VIRTUAL_SECTION`.

## 30.4 — Phase 4 — Section Selection

**Inputs**: the effective definition from Phase 3, the caller's `?` / `!` section toggle map.

**Outputs**: the effective definition with each `?`/`!` section marked ON or OFF; OFF sections (and their sub-sections) flagged inert.

**Operations** (`§6.1` PHASE 4):

- Apply the toggle map. Each `?`/`!` section is ON or OFF (default from its suffix, overridden by the user's choice); a plain `@@@section` is always ON.
- An OFF section and all its sub-sections become *inert*: their body text, includes, and assertions take no further part. Selection happens here — before any binding — so inert material never triggers Phase 8 (includes) or Phase 10 (assertions), and substitutions confined to inert material never enter the Phase 9 required-parameter set.

**Errors raised**: none specific to this phase (a toggle naming a section that does not exist surfaces as the implementation's toggle-validation error).

## 30.5 — Phase 5 — Parameter Binding

**Inputs**: the prompt's parameter declarations, the selected parameter set (if any), the collection-scope parameter sets merged in Phase 2, the instantiation values, the constant declarations.

**Outputs**: a parameter-binding table where each parameter is bound to a value (from a set, an explicit value, or its default); the constant table, resolved and type-validated.

**Operations** (`§6.1` PHASE 5):

- **Constants are resolved at the START of this phase** (hoisted from Phase 6): a constant's value depends on no instantiation input, and an include binding value may reference a caller constant, so it must be available here. A wrong-typed constant is TYPE_MISMATCH, reported here.
- Compute each parameter's bound value in its own prompt frame: apply the ONE selected parameter set (template-embedded or collection-level, resolved at Phase 2), then explicit ins-

tantiation values, then the declared default for any parameter still unbound. Template-embedded sets are forbidden when the prompt is in a collection ( §5.9 ), so the two set sources cannot collide.

- For an `@@@include B(name=value)` call site, COMPUTE the binding values here, in the caller's frame ; a value that substitutes one of the caller's own parameters is resolved at this point. These bindings are NOT yet checked against B's interface — B is not resolved until Phase 8.

> ***`standard.md §5.9` — Parameter Set Syntax (excerpt) :***
>
> *A `@@@parameter-set <name> ... @@@end-parameter-set` block declares one-binding-per-line `<param_name>: <value>` entries ; values may be strings, numbers, or booleans. RESTRICTION : `parameter-set` blocks are ONLY allowed in templates that do NOT belong to a collection. If the prompt's `@@@template` header is collection-qualified, `@@@parameter-set` is FORBIDDEN — use collection-level parameter sets instead.*

**Errors raised** : `TYPE_MISMATCH` (constants), `UNDEFINED_REFERENCE` (unknown set), `UNDEFINED_REFERENCE` (a caller-frame symbol feeding an include binding value that names nothing).

## 30.6 — Phase 6 — Constant Resolution

**Inputs** : the (effective, post-inheritance) constant declarations, already resolved at the start of Phase 5.

**Outputs** : a constant table mapping each constant name to its bound value ; immutability enforced.

**Operations** ( `§6.1` PHASE 6) :

- Constant values were resolved and type-validated at the start of Phase 5 (above), since they are instantiation-independent and

may feed an include binding value. This phase confirms that resolution and validates immutability constraints.

- Detect attempts to re-bind a constant — typically a child prompt re-declaring a parent's constant with a different value. Raise IMMUTABILITY_VIOLATION.
- A reference to an undefined constant surfaces during substitution (Phase 9) as UNDEFINED_REFERENCE; some implementations also flag a missing constant declaration here as UNDEFINED_REFERENCE.

**Errors raised**: `IMMUTABILITY_VIOLATION`, `UNDEFINED_REFERENCE`.

## 30.7 — Phase 7 — Attachment Binding

**Inputs**: the (effective, post-selection) attachment declarations, the instantiation context (which attachments did the caller provide?), the storage backend.

**Outputs**: a slot-filled attachment map: each declared attachment is either bound to a content reference or known to be absent (optional and not provided).

**Operations** (`§6.1` PHASE 7):

- For each `fixed` attachment (source `file` / `blob` / `record`): validate that the path exists or the ref resolves through the configured store. Raise ATTACHMENT_NOT_FOUND on a missing path or unknown ref. (A `blob` data URI's well-formedness was already checked statically at Phase 1; nothing is fetched here.)
- For each `to-include` attachment (required, no `?` prefix on `attachment`): consume the instantiation's value. If no value was provided, raise ATTACHMENT_REQUIRED.
- For each `to-include` attachment (optional, `?` prefix on `attachment`): consume the instantiation's value if provided; otherwise, mark the slot as absent.

**Errors raised**: `ATTACHMENT_NOT_FOUND`, `ATTACHMENT_REQUIRED`.

## 30.8 — Phase 8 — Include Resolution

**Inputs**: the effective definition from Phase 3, the effective prompt set from Phase 2 (for L5 collection-axis `^parent` references), the binding values computed in Phase 5, the templates in the storage backend.

**Outputs**: the prompt body with every `@@@include` (line form) and `@@@{include ...}` (inflowing form) replaced by the rendered content of its target.

**Operations** (`§6.1` PHASE 8 — resolve, then validate interface, then apply bindings, then render recursively):

- For each include directive, RESOLVE the reference to a single target by walking the ordered lookup cascade (8.a qualified handle → 8.b the caller's collection inheritance chain → 8.c root). The first non-empty match wins silently; ambiguity at a single stage raises AMBIGUOUS_INCLUDE. Alias references are resolved here, during segmentation. Template-axis `parent`/`self` references (no `^`) resolve against the inheritance chain (Chapter XX); collection-axis `^parent.<template_id>` / `^self.<template_id>` references (with `^`) resolve against the effective prompt set from Phase 2. The two axes are disjoint — no precedence, no fallback.
- VALIDATE the interface of the call-site bindings against the resolved target's effective signature: a binding naming a parameter the target does not declare raises UNDECLARED_INCLUDE_PARAMETER.
- APPLY the binding values computed in Phase 5, then RENDER the target recursively in its own frame.
- Detect cycles (CIRCULAR_INCLUDE) by tracking the include stack during recursive resolution.
- The verbatim regions identified in Phase 1 are skipped: an include inside a verbatim block is left as literal text.

- The depth of include recursion is implementation-defined but finite. A reasonable default is 32 levels.
- An include whose resolved target is a `virtual` section (a section that carries no body) raises INCLUDE_OF_VIRTUAL_SECTION — the renderer has nothing to substitute. The error applies to both `parent.<section>` and `<prompt>.<section>` include forms.

**Errors raised**: `CIRCULAR_INCLUDE`, `AMBIGUOUS_INCLUDE`, `UNDECLARED_INCLUDE_PARAMETER`, `INCLUDE_OF_VIRTUAL_SECTION`, `UNDEFINED_REFERENCE`.

## 30.9 — Phase 9 — Parameter Substitution

**Inputs**: the body from Phase 8, the parameter-binding table from Phase 5, the constant table from Phase 6, the system- variable provider, the override list from any include call sites.

**Outputs**: the body with every `@@@=name` reference substituted by its bound value.

**Operations** (`§6.1` PHASE 9):

- For each `@@@=name` (and `@@@{= name }`) reference in the retained body:
- If `name` is a known system variable, substitute the processor's resolved value (Chapter X).
- If `name` is a known parameter, substitute the bound value. If the parameter has no binding and is required, raise UNDEFINED_PARAMETER. If the parameter is optional, use its (mandatory) default.
- If `name` is a known constant, substitute the constant's value.
- Otherwise, raise UNDEFINED_REFERENCE (the symbol is neither a system variable nor a declared parameter nor a declared constant).
- Compute the required-parameter set over the retained (non-inert) body and check it is bound.

- Validate every substituted value against its declared constraints. Raise CONSTRAINT_VIOLATION or SELECT_INVALID_OPTION.

**Errors raised**: `UNDEFINED_PARAMETER`, `UNDEFINED_REFERENCE`, `CONSTRAINT_VIOLATION`, `SELECT_INVALID_OPTION`.

## 30.10 — Phase 10 — Assertion Evaluation

**Inputs**: the fully rendered body from Phase 9, the scope and section structure from the parsed tree, the `@@@assert` directives.

**Outputs**: the same body, unchanged, when every assertion holds — assertions never alter the text; or a blocked instantiation when one fails.

**Operations** (`§6.1` PHASE 10):

- Validate scope nesting. Sections, `@@@scope` boundaries, includes, and blocks must form a properly nested tree. A `@@@scope` opened inside a section, where a new `@@@section` marker begins before the matching `@@@end-scope`, raises MALFORMED_SCOPE.
- For each `@@@assert`, evaluate its predicate (`contains`, `not contains`, `matches`, `minlen`, `maxlen`) against the ENTIRE rendered content of its nearest enclosing scope (`@@@scope`, section, include, or whole prompt), independent of the assert's own position within that scope.
- Multiple asserts sharing a scope combine as an implicit AND; every one must hold.
- The content an assertion reads carries NO structure heading: those are emitted in Phase 11, after this one. A predicate written against the author's own content is never decided by a processor-emitted title — `maxlen 10` on a ten-character section still holds.
- On any unsatisfied predicate, raise ASSERTION_FAILED and block instantiation — no output is emitted.

Evaluation is deterministic (identical bindings → identical rendered content → identical verdict) and applies to the rendered prompt only, never to the model's response (Chapter XIII).

> ***`standard.md §5.15.4` — Evaluation timing (excerpt):***
>
> *Assertions are evaluated LATE: AFTER substitution, includes and inheritance have fully resolved, and BEFORE emission. Evaluation is deterministic. Assertions do NOT alter the output text; on success the rendered content is emitted unchanged (PHASE 11), on failure instantiation is blocked (PHASE 10, `ASSERTION_FAILED`).*

**Errors raised**: `MALFORMED_SCOPE`, `ASSERTION_FAILED`.

## 30.11 — Phase 11 — Output Generation

**Inputs**: the substituted body from Phase 9, validated in Phase 10, the parsed-tree markers that have not yet been stripped, the rendering options.

**Outputs**: a `RenderedTemplate` — the final output string ready to send to the language model, accompanied by the ordered list of attachments bound in Phase 7.

**Operations** (`§6.1` PHASE 11 — every directive is stripped here):

- Strip the identity header and declarations (`@@@template ...`, `@@@name ...`, `@@@description ...`, `@@@template-body`). The id, the `inherits` clause, and the `virtual` / `final` modifiers all ride on the `@@@template` header, so they vanish with it.
- Strip lifecycle markers (`@@@alias ...`, `@@@deprecated ...`).
- Strip declaration directives that did not affect the output (`@@@param ...`, `@@@const ...`, `@@@attachment ...`, `@@@collection ...` / `@@@end-collection`, the `@@@parameter-set ...` block).

- Strip the `@@@scope` / `@@@end-scope` boundaries, which are transparent to output.
- Replace the definition of each retained section by its **structure heading** — `## <Title>` for a section, `### <n>. <Title>` for a sub-section, numbered among its retained siblings. The prompt itself contributes NO heading: `text` carries content, and a prompt's `@@@name` is a label while its id is an address — the host receives both on the returned value, as it receives attachments, and titles a document if a document wants a title. The `<Title>` is the section's NAME with its first letter uppercased; the `: <description>` is dialog prose and never reaches the output (§5 DESCRIPTION SYNTAX). Three levels, and three only: that is why *body text* may not open a level 1-3 heading (`§5.3`, RESERVED HEADING LEVELS). A header `@@@section` listing declares visibility only and is stripped like any other directive.
- Replace a full-line `@@@include` in place by its target's rendered text (resolved at Phase 8).
- Strip comments (`@@@-- ...`, `@@@(-- ... )`, `@@@comment ... @@@end-comment`) — they have been preserved through the prior phases only to be removed here.
- Strip verbatim *delimiters* (the inner content was preserved through every prior phase).
- Separate each heading from what surrounds it by exactly one blank line. Everything else is byte-exact: body text is emitted verbatim, character for character, and blank source lines inside it are preserved — two conforming processors produce the same `text`.

**Errors raised**: none (a render that reaches Phase 11 has already passed every error gate, including the assertions of Phase 10).

## 30.12 — A short reading of the phase numbering

The numbering is *normative*: a conformant processor SHALL execute the phases in this order (`§6.1`). The numbering is also *pedagogical*: an author reading the chapters in numerical order encounters the constructs in the same order the processor processes them, which is why the book's organization mirrors the spec's.

The phases are not always *visible* steps in an implementation. A well-tuned processor may fold Phase 6 (Constant resolution) into Phase 5 (Parameter binding) — the spec already hoists constant resolution to the start of Phase 5 — or combine Phase 8 (Includes) and Phase 9 (Parameter substitution) into a single tree-walk. The *observable* behaviour must match the spec; the internal implementation may consolidate.

The next chapter is the full error catalogue: every condition raised in the phases above, with its trigger and remediation.

# Chapter XXXI — Error Catalog

> ***`standard.md §6.2` — Error Conditions (the complete catalog):***
>
> *The catalog is the FULL set of **58 error codes** and **3 warning codes** — not a representative selection. The 58 errors: `INVALID_SYNTAX`, `INVALID_BOUNDED_KEYWORD`, `DUPLICATE_DECLARATION`, `SECTION_WITH_SUBSECTIONS_HAS_BODY`, `EXPORT_OF_UNKNOWN_SECTION`, `EXPORT_OF_UNKNOWN_MEMBER`, `INVALID_SECTION_PLACEMENT`, `INVALID_SECTION_ANCHOR`, `CIRCULAR_INHERITANCE`, `CIRCULAR_COLLECTION_INHERITANCE`, `CIRCULAR_LIBRARY_DEPENDENCY`, `CIRCULAR_INCLUDE`, `AMBIGUOUS_INCLUDE`, `AMBIGUOUS_COLLECTION`, `AMBIGUOUS_ENTITY`, `AMBIGUOUS_USES_TARGET`, `UNDEFINED_USES_TARGET`, `UNDEFINED_REFERENCE`, `UNKNOWN_COLLECTION`, `UNKNOWN_FRAME`, `UNKNOWN_PARAMETER_SET`, `UNKNOWN_SECTION_TOGGLE`, `UNIMPLEMENTED_VIRTUAL_TEMPLATE`, `UNIMPLEMENTED_VIRTUAL_SECTION`, `OVERRIDE_OF_FINAL_TEMPLATE`, `OVERRIDE_OF_FINAL_SECTION`, `OVERRIDE_CONTRACT_VIOLATION`, `VIRTUAL_TEMPLATE_HAS_BODY`, `VIRTUAL_SECTION_HAS_BODY`, `DELETED_SECTION_HAS_BODY`, `DELETE_OF_UNINHERITED_SECTION`, `EMPTY_SECTION`, `TOGGLABLE_SECTION_WITHOUT_DESCRIPTION`, `VIRTUAL_SIGNATURE_MISMATCH`, `SECTION_SHAPE_MISMATCH`, `INCLUDE_OF_VIRTUAL_SECTION`, `NOT_EXPORTED`, `UNDEFINED_PARAMETER`, `UNDECLARED_INCLUDE_PARAMETER`, `UNDECLARED_INSTANTIATION_PARAMETER`, `UNDECLARED_PARAMETER_SET_BINDING`, `UNDECLARED_ATTACHMENT_BINDING`, `ATTACHMENT_NOT_FOUND`, `ATTACHMENT_REQUIRED`, `IMMUTABILITY_VIOLATION`, `TYPE_MISMATCH`, `CONSTRAINT_VIOLATION`, `INVALID_CONSTRAINT`, `SELECT_INVALID_OPTION`, `UNENUMERABLE_INCLUDE_TARGET`, `TRANSPARENT_INCLUDE_ON_TEMPLATE`, `ALIAS_CONFLICT`, `USES_A-*

> *LIAS_CONFLICT, SYSTEM_VARIABLE_OVERRIDE, ASSERTION_FAILED, INVALID_ASSERTION, MALFORMED_SCOPE, RESOURCE_LIMIT_EXCEEDED. The 3 warnings: MISSING_DESCRIPTION, DEPRECATED_TEMPLATE, PARAMETER_NAME_IS_PREFIX.*

The specification enumerates the error and warning conditions that a conformant processor must handle. This chapter is the descriptive companion to that enumeration — the COMPLETE catalog of 58 errors and 3 warnings. For each condition it gives when it fires, where it is raised, and what the author or operator should do about it. The order roughly follows the phase in which each condition typically surfaces.

## 31.1 — Errors

**`INVALID_SYNTAX`**

**When** : Phase 1. **Trigger** : a malformed delimiter, an unknown keyword, a reserved keyword used as an identifier, a directive containing another `@@@=...` directive inside its argument body, or a declaration carrying both the `virtual` and `final` modifiers (`@@@section virtual final task`, `@@@template virtual final C.task`) — the two are mutually exclusive, and the grammar rejects the combination as malformed. Also a **body-text line opening a markdown heading of level 1, 2 or 3** (one, two or three `#` followed by a space, a tab or the end of line) **inside a declared section** — unstructured body content (a Level 0 source, a prompt with no declared section, the implicit section) has no section heading emitted for it and keeps its Markdown as written : those three levels are reserved to the structure the renderer emits (`§5.3`, Chapter VII). A `: description` is not body text and is never rendered — a marker written in it (`@@@section rules : ## Règle finale`) is inert, neither emitted nor refused. **Remediation** : fix the offending source. A linter or syntax- highlighting editor catches the majority of these before the render is attempted. For a reserved heading, the usual fix is to make the line a section of its own, whose name the

structure step emits as that very heading; otherwise use `####` or deeper — level 4 and below are ordinary body text and nest naturally under the emitted `###` — or wrap the line in `@@@verbatim` when the literal `#` is the point (Chapter IX).

#### `ALIAS_CONFLICT`

**When**: Phase 1. **Trigger**: a `@@@alias <id>` declaration whose alias collides with another prompt's `id` or with another prompt's alias in the same scope. **Remediation**: rename one of the colliding identifiers, or remove the alias if it is no longer needed.

#### `SYSTEM_VARIABLE_OVERRIDE`

**When**: Phase 1. **Trigger**: a `@@@param` or `@@@const` is declared with a name that collides with a reserved system variable (`DATE`, `USER`, `LANG`, …). The substitution `@@@=name` always resolves to the system value, so the declaration could never take effect. **Common cause**: an author named a parameter `USER` or `DATE` without realising those names are reserved (Chapter X). **Remediation**: rename the parameter or constant to a non-reserved identifier. The error is raised at validate / save time, not deferred to render.

#### `VIRTUAL_SECTION_HAS_BODY`

**When**: Phase 1. **Trigger**: a `virtual section` declaration is followed by non-empty content before the next section opening or the end of the prompt. Only whitespace and comments (`@@@-- ...`, `@@@(-- ... )`, `@@@comment ... @@@end-comment`) are tolerated; anything else fires. **Common cause**: an author wrote a « default » body inside a virtual section, confusing the section-level `virtual` (which declares an empty contract) with a parent prompt's overridable body. **Remediation**: either remove the body (keeping the section abstract for descendants to implement) or remove the `virtual` modifier (making the section a normal, overridable section with a default body).

### `CIRCULAR_COLLECTION_INHERITANCE`

**When**: Phase 2. **Trigger**: a collection chain that visits the same collection twice via the `inherits` clause on its `@@@collection` header — directly (A → A) or indirectly (A → B → A). **Remediation**: examine the collection chain (typically using the implementation's collection-inheritance viewer) and break the cycle. Often the right move is for one collection to inherit from a common ancestor instead of from each other.

### `UNKNOWN_COLLECTION`

**When**: Phase 2. **Trigger**: a reference names a collection that no `@@@collection` block declares — most often an `inherits` clause on a `@@@collection` header pointing at a name that is only a folder (a folder is pure storage and declares no collection), or a mistyped collection name. **Remediation**: declare the target with a `@@@collection` block (promote the folder to a collection if its semantics warrant it), fix the spelling, or remove the `inherits` reference and let the dependent collection be its own root.

### `UNIMPLEMENTED_VIRTUAL_TEMPLATE`

**When**: Phase 2. **Trigger**: rendering a virtual prompt — or a prompt that transitively depends on a virtual — without a non-virtual override in the leaf collection. **Remediation**: provide a concrete override in the leaf collection (a member prompt with the same `id` and matching parameters). Until the override exists, the virtual is not instantiable.

### `OVERRIDE_OF_FINAL_TEMPLATE`

**When**: Phase 2. **Trigger**: a derived collection declares a member prompt with the same id as a prompt whose `@@@template` header carries the `final` modifier upstream in the collection inheritance chain. The error fires regardless of whether the override changes the content — sealing is total. **Common cause**: a collection author specialised a base that explicitly sealed one of its members. The override is forbidden by design. **Remediation**: remove the over-

ride, accepting the sealed member's content. If the base's seal is wrong (the member *should* be overridable), revisit the base and remove the `final` modifier — a sealed member is a deliberate commitment by the base author and should not be lifted casually.

#### `CIRCULAR_INHERITANCE`

**When**: Phase 3. **Trigger**: a template-level `inherits` chain (the inline clause on the `@@@template` header) that visits the same prompt twice. **Remediation**: break the cycle in the inheritance chain. The fix is usually a refactoring: one of the involved templates should inherit from a common ancestor instead of from each other.

#### `OVERRIDE_OF_FINAL_SECTION`

**When**: Phase 3. **Trigger**: a descendant prompt redeclares a section whose ancestor declaration carried the `final` modifier. The error fires for *any* redeclaration — including one that changes only the presence toggle (`?` / `!`) or the description, with no body change. Each section declaration is read integrally. **Common cause**: an author wanted to tweak just the description or just the toggle of a final section, not realising that any redeclaration is forbidden. **Remediation**: remove the override. If the section needs to vary at the descendant level, the base author must remove the `final` modifier from the ancestor — sealing is a deliberate commitment.

#### `UNIMPLEMENTED_VIRTUAL_SECTION`

**When**: Phase 3. **Trigger**: a section declared `virtual` somewhere in the inheritance chain has no concrete (non-virtual) override at the chain leaf. The deepest-non-virtual rule cannot resolve a body for the section. **Common cause**: an intermediate descendant redeclared the section `virtual` (escalation) and the leaf descendant forgot to implement, or the leaf is the parent prompt itself and was used directly without a derived child. **Remediation**: provide a concrete override of the section (no `virtual` modifier, with body)

in a descendant prompt that is the actual render target. If the prompt should be abstract by design and never rendered directly, surface this in tooling so callers never select it.

#### `CIRCULAR_INCLUDE`

**When**: Phase 8. **Trigger**: an `@@@{include }` chain that, directly or transitively, includes the prompt that started the chain. **Remediation**: identify the cycle (the implementation's include-graph viewer shows it). Extract the shared content into a third prompt that both ends include.

#### `AMBIGUOUS_INCLUDE`

**When**: Phase 8 (Include Resolution). **Trigger**: an include handle resolves to more than one candidate prompt at a single stage of the lookup cascade (`§6.1`). Most often: two collections each expose a prompt with the same id, and the caller's containment / inheritance chain does not prefer one over the other. **Diagnostic**: the candidate list — each entry pairs the canonical prompt id with the collection path that hosts it. **Remediation**: qualify the handle. `@@@{include Marketing.welcome}` selects `welcome` in `Marketing` regardless of any homonymous prompt elsewhere.

```
 # Two templates named `welcome` in unrelated collections:
@@@{include welcome}            → AMBIGUOUS_INCLUDE
@@@{include Marketing.welcome}  → resolves
```

#### `INCLUDE_OF_VIRTUAL_SECTION`

**When**: Phase 8. **Trigger**: an include of the form `@@@{include parent.<section>}` or `@@@{include <prompt>.<section>}` resolves to a `virtual` section — a section with no body. The renderer has nothing to substitute. **Common cause**: a child prompt's section pulled `parent.<x>` without realising the parent's `<x>` is still abstract, or an include reached across to a sibling prompt whose target section is `virtual`. **Remediation**: include from a prompt that has

a concrete (non-virtual) declaration of the target section. Often this means walking one level further down the inheritance chain to a concrete implementation, or selecting a different include target.

#### `UNDEFINED_REFERENCE`

**When**: any of Phases 2, 3, 8, 9. **Trigger**: a reference (in an `inherits` clause, an `include`, collection membership, a `parameter-set` selection, or a `@@@=name` substitution naming a parameter or constant that no declaration introduces) that does not resolve to an existing entity in the current scope. **Remediation**: verify the spelling of the reference, check that the target exists at the expected scope, and confirm that the caller has read access if the implementation uses group-scoped access control.

#### `UNDECLARED_INCLUDE_PARAMETER`

**When**: Phase 8 (include resolution, once the target is resolved). **Trigger**: a parameter binding passed at an `@@@{include … (name=value)}` site — or on a `self.` / `parent.` reference — names a parameter that the *target* prompt does NOT declare. The construct parses fine; the binding simply has no matching parameter on the target (Chapter XVIII). **Common cause**: a typo in the binding name, or a binding left behind after the target's parameter was renamed or removed. **Remediation**: bind only parameters the target declares. Implementations SHOULD attach a Damerau-Levenshtein « did you mean » suggestion against the target's declared parameter names. Note: only parameters *referenced* in the target's rendered dependency closure are required — passing a declared-but-unused parameter is fine; passing an *undeclared* one is the error.

#### `UNDEFINED_PARAMETER`

**When**: Phase 9. **Trigger**: the rendered body references a parameter that neither the instantiation nor a parameter set bound, and the parameter is required with no default. **Remediation**: bind the parameter at instantiation, declare it as optional (`param?`), or give it a default value at the declaration site.

`ATTACHMENT_NOT_FOUND`

**When**: Phase 7. **Trigger**: a `fixed` attachment whose path does not exist in the storage backend, or whose data URI is malformed. **Remediation**: correct the path, or replace it with a data URI that contains the intended content.

`ATTACHMENT_REQUIRED`

**When**: Phase 7. **Trigger**: a required `to-include` attachment (no `?` prefix on `attachment`) was not provided at instantiation. **Remediation**: provide the attachment at instantiation, or mark the attachment optional if absence is sometimes acceptable.

`IMMUTABILITY_VIOLATION`

**When**: Phase 6. **Trigger**: an attempt to **bind a constant at instantiation** — the caller supplies a value for a `@@@const` name, which a constant may never accept. (A child prompt *re-declaring* a parent's constant is NOT this error — constant-on-constant override is allowed, and Phase 3 simply replaces the value.) **Remediation**: do not pass the constant's name at instantiation; if the value genuinely needs to vary per caller, refactor the constant into a parameter (Chapter XIV's anti-pattern).

`TYPE_MISMATCH`

**When**: Phase 5 (constants, resolved at the start of parameter binding) or Phase 9 (parameters). **Trigger**: a value does not match its declared type — a `number` parameter bound to a string, a `boolean` constant declared with a non-boolean value, etc. **Remediation**: correct the type at the binding site, or adjust the parameter declaration to accept the type the caller is sending.

`CONSTRAINT_VIOLATION`

**When**: Phase 5 (constants) or Phase 9 (parameters). **Trigger**: a value of the right type that nonetheless violates a declared constraint (`min`, `max`, `minlen`, `maxlen`, `pattern`). **Remediation**: change the bound value to satisfy the constraint, or — if the constraint itself was wrong — adjust the constraint declaration.

`SELECT_INVALID_OPTION`

**When**: Phase 5 (constants) or Phase 9 (parameters). **Trigger**: a value bound to a `select` parameter or constant is not one of the declared options. **Remediation**: bind one of the declared options; if the intended option is missing, add it to the parameter's option list.

`MALFORMED_SCOPE`

**When**: Phase 10. **Trigger**: a `@@@scope` partially overlaps a section — it is opened inside a section and a new `@@@section` marker begins before the matching `@@@end-scope`, so sections and scopes do not form a properly nested tree. (See Chapter XIII.) **Common cause**: an author opened a scope mid-section to wrap a clause but let the next `@@@section` start before closing it. **Remediation**: either close the scope before the next section opens (so the scope sits within one section), or open it before the first section so that it encloses one or more *whole* sections. A scope may never straddle a section boundary.

`ASSERTION_FAILED`

**When**: Phase 10. **Trigger**: a blocking `@@@assert` predicate (`contains`, `not contains`, `matches`, `minlen`, `maxlen`) is not satisfied by the rendered content of its nearest enclosing scope (Chapter XIII). Assertions are evaluated after substitution, includes, and inheritance have fully resolved, and before output. Instantiation is blocked — no output is emitted. **Common cause**: a required phrase is missing from the rendered text, a forbidden term slipped in through a parameter value, or the rendered content fell outside a

`minlen`/`maxlen` bound. **Remediation**: adjust the content (or the bindings that feed it) so the predicate holds; or, if the assertion itself is wrong, correct the predicate. The diagnostic SHOULD name the failing predicate and the scope's label, if any.

## 31.2 — The remaining error conditions

The following complete the 56-code catalog. Each is a typed `ErrorCode` of `§6.2`; the concise form below gives the trigger and phase.

**`DUPLICATE_DECLARATION`**

**When**: Phase 1. A single-valued directive appears twice, or two declarations share a canonical id (recall identifiers are case- and diacritic-insensitive, so `Task` and `task` collide). **Remediation**: remove the duplicate or rename.

**`SECTION_WITH_SUBSECTIONS_HAS_BODY`**

**When**: Phases 1 and 8. A section that has sub-sections also carries a body of its own — a container renders only its sub-sections. An `@@@include` is NOT a body: it inserts, and what it inserts is judged once inserted. So the rule has two moments under this one code — literal text written beside sub-sections at 1, on the document alone; rendered text an include BRINGS into such a section at 8, on the material. A container may therefore carry the includes that feed it. **Remediation**: move the body into a sub-section, or pull sub-sections rather than a leaf section.

**`DELETED_SECTION_HAS_BODY`**

**When**: Phase 1. A `@@@section delete <name>` is followed by rendered material. A deleted section renders nothing — heading included — so that material would be silently lost. **Remediation**: drop the body, or drop the `delete` and override the section with the content you want.

### `DELETE_OF_UNINHERITED_SECTION`

**When** : Phase 3. `@@@section delete <name>` names a section the prompt does not inherit. Removing one's OWN section is simply not writing it. **Remediation** : delete only what an ancestor declares.

### `EMPTY_SECTION`

**When** : Phases 1 and 3. A section definition carries no rendered material. Emptying an override was the de-facto way to suppress inherited content — silently — and `delete` now says it out loud, so the mute form has no reason to exist. FOUR exemptions, each already a rule of the language rather than a dispensation : a container (its content is its sub-sections), a `virtual` section (abstract), a `delete` section (renders nothing), and every section of a `virtual` prompt. The rule is judged on the EFFECTIVE shape, because containment can be inherited : at 1 for a prompt that inherits nothing, at 3 otherwise — inheriting is not an exemption, it only moves the moment of judgement. **Remediation** : write the content, or `@@@section delete` it.

### `TOGGLABLE_SECTION_WITHOUT_DESCRIPTION`

**When** : Phase 1. A `?` or `!` section is declared with no `: description`, or an empty one. The section is a choice offered at instantiation, and the description is all the dialog has to say what is being ticked — the rendered heading comes from the name, so the description has no other role, and it is mandatory here. Judged on the declaring line, for every prompt : an override REPLACES the inherited description in full, so inheritance never supplies one to a section the child redeclares. A `delete` section is exempt — it renders nothing and is no choice. **Remediation** : write the prose a user needs to decide, or make the section plain if it was never meant to be optional. Never « drop the `?` » on the author's behalf — they wanted a choice.

### `INVALID_BOUNDED_KEYWORD`

**When**: Phase 1. A bounded form (`@@@( … )` and its three sibling families) carries a keyword that does not exist there. Only four do: a substitution (`= name`), an `include`, a comment (`--`) and a `verbatim` body. A structural or declaration directive — `section`, `name`, `prompt`, `param`, … — exists in LINE form only. A processor MUST NOT fall back to emitting the construct as literal text: the markup would reach the rendered prompt, and an author who believed they were declaring something would get their own source back with nothing said. **Remediation**: use the line form; to write a literal sigil, escape it as `@@@@`.

### `TRANSPARENT_INCLUDE_ON_TEMPLATE`

**When**: Phase 8. An include marks its target transparent — `@@@include (<reference>)`, §5.3, the consumer-side half of `@@@section (<name>)` — but the reference resolves to a WHOLE TEMPLATE rather than to a section.

**Why**: an included prompt emits no heading of its own (§6.1 11.a), so the mark has nothing to withhold. Reading it as « make every section of that prompt transparent » would be a wider request than what was written, applied silently to sections the author never named.

**Fix**: mark the sections that should lose their heading in their own document (`@@@section (<name>)`), or target one section at the include (`@@@include (<prompt>.<section>)`).

### `UNENUMERABLE_INCLUDE_TARGET`

**When**: Phase 8. An indirect include target (`@@@include @@=name`) names a symbol whose domain is not declared and finite — a `text` parameter, a system variable, an undeclared name. Only a `select` parameter or a constant may name a target, so the reachable set stays enumerable from the source and the include graph analysable

before instantiation. The refusal is about the DOMAIN, not the value. **Remediation**: declare the target symbol as a `select` with its options, or as a constant.

**`SECTION_SHAPE_MISMATCH`**

**When**: Phase 3. An inheritance chain turns a leaf section into a container (or vice versa). **Remediation**: keep a section's shape (leaf vs container) consistent across the chain.

**`EXPORT_OF_UNKNOWN_SECTION` / `EXPORT_OF_UNKNOWN_MEMBER`**

**When**: Phases 1–2. A `@@@template-body` header lists a section that is not in the prompt's effective definition (`EXPORT_OF_UNKNOWN_SECTION`), or a `@@@collection` block lists a member that no prompt defines (`EXPORT_OF_UNKNOWN_MEMBER`). **Remediation**: list only sections/members that exist.

**`NOT_EXPORTED`**

**When**: Phase 8. A cross-template or cross-collection reference targets a PRIVATE section (not listed after `@@@template-body`) or a private member. **Remediation**: export the target by listing it in the header, or reference a public one.

**`INVALID_SECTION_PLACEMENT` / `INVALID_SECTION_ANCHOR`**

**When**: Phase 3. A `first`/`last`/`before`/`after` placement clause sits on an override or in a prompt with no `inherits` parent (`INVALID_SECTION_PLACEMENT`), or a `before`/`after` anchor is not an inherited section (`INVALID_SECTION_ANCHOR`). **Remediation**: place only ADDED sections, and anchor against an inherited section.

**`OVERRIDE_CONTRACT_VIOLATION`**

**When**: Phase 3.b (call contract). A public override breaks a call valid against the inherited element — tightening a constraint, turning `?` into `!`, making an omissible parameter required, and similar. **Remediation**: overrides may only RELAX the contract; keep the override substitutable.

`VIRTUAL_TEMPLATE_HAS_BODY`

**When** : Phase 1. A prompt whose `@@@template` header carries the `virtual` modifier also carries a rendered body. **Remediation** : a virtual prompt is abstract — remove the body, or drop `virtual`.

`VIRTUAL_SIGNATURE_MISMATCH`

**When** : Phase 2. A concrete override's parameter signature is incompatible with the virtual it implements (a required parameter dropped, a type changed). **Remediation** : match the abstract signature (optional additions are fine).

`UNKNOWN_FRAME` / `UNKNOWN_SECTION_TOGGLE` / `UNKNOWN_PARAMETER_SET`

**When** : Phases 2/4/5. A render selects a frame, a section toggle, or a parameter set that the effective definition does not declare. **Remediation** : select only declared frames, toggles, and sets (these replace any « implementation-defined » handling).

`UNDECLARED_INSTANTIATION_PARAMETER` / `UNDECLARED_PARAMETER_SET_BINDING` / `UNDECLARED_ATTACHMENT_BINDING`

**When** : Phases 5–7. An instantiation value, a parameter-set binding, or an attachment binding names something the target does not declare. **Remediation** : bind only declared parameters / attachments.

`INVALID_CONSTRAINT`

**When** : Phase 1. A constraint is malformed — a `pattern` that is not a valid ECMAScript regex, a constraint key on a type that does not accept it. **Remediation** : fix the constraint expression.

`INVALID_ASSERTION`

**When** : Phase 10. An `@@@assert` predicate is malformed — a non-integer/negative length, or a `matches` whose assembled regex is not valid ECMAScript. **Remediation** : correct the predicate.

`AMBIGUOUS_COLLECTION` / `AMBIGUOUS_ENTITY`

**When** : Phase 2/8. A collection reference (or a general entity reference) resolves to more than one candidate at a single lookup stage. **Remediation** : qualify the reference through its containment path.

`CIRCULAR_LIBRARY_DEPENDENCY`

**When** : Phase 1–2. Two libraries ( `@@@library` , §5.16) depend on each other in a cycle. **Remediation** : break the library dependency cycle.

`UNDEFINED_USES_TARGET` / `AMBIGUOUS_USES_TARGET` / `USES_ALIAS_CONFLICT`

**When** : Phases 1–2. A `@@@uses` import (§5.17) names a target that does not exist, is ambiguous, or whose local alias collides. **Remediation** : name an existing, unambiguous target and a free alias.

`RESOURCE_LIMIT_EXCEEDED`

**When** : any phase (commonly Phase 8). An IMPLEMENTATION limit was reached — include-nesting depth, inheritance-chain length, attachment size, member count, and similar ( `§10.1` ). The limit is implementation-defined but MUST be documented and reported as this code, naming the limit, never as a language-level error. **Remediation** : reduce the depth / size, or raise the documented limit.

## 31.3 — Warnings

POO²L defines exactly THREE warning conditions : `MISSING_DESCRIPTION` , `DEPRECATED_TEMPLATE` , and `PARAMETER_NAME_IS_PREFIX` . A warning NEVER blocks rendering in the baseline semantics, and *how* a processor handles one is implementation latitude : it MAY ignore the warning, surface it to the user, or (as a documented stricter mode) elevate it to a blocking error. Whi-

chever it chooses, the choice does NOT affect conformance — the specification only DEFINES the conditions (`§6.2`, `§8.2`). They are therefore never part of the conformance SHALL set.

**`MISSING_DESCRIPTION`**

**When**: Phase 1. **Trigger**: a structured (L1+) prompt declares `@@@template` and `@@@name` but omits the OPTIONAL `@@@description` (`§5.2`). The prompt is valid and renders normally; the condition merely flags the omission for catalog / UI / exchange use. **Remediation**: add a `@@@description`, or — if the omission is deliberate — leave it and accept the advisory. The warning never blocks the render.

**`DEPRECATED_TEMPLATE`**

**When**: Phase 1. **Trigger**: a deprecated prompt is instantiated, included, or inherited. The migration message accompanies the warning. **Remediation**: migrate the caller per the deprecation message. Until migration, the warning is informational; the render continues.

**`PARAMETER_NAME_IS_PREFIX`**

**When**: Phase 9 (substitution, where the longest-match binding applies). **Trigger**: a declared parameter/constant name is a strict prefix of another declared name; an implicit `@@@=name` substitution binds the longest match, which may not be the one the author intended. **Remediation**: use the braced form `@@@{= name }` for the shorter name to disambiguate, or rename so that no name is a prefix of another. The condition is advisory only.

## 31.4 — A summary table

The table below is a quick-reference for the most frequently hit codes; the complete 58-error / 3-warning catalog is the itemized set above.

| Code | Phase | Severity | Catch by |
|---|---|---|---|
| `INVALID_SYNTAX` | 1 | error | linter / editor |
| `ALIAS_CONFLICT` | 1 | error | save-time validation |
| `SYSTEM_VARIABLE_OVERRIDE` | 1 | error | save-time validation |
| `VIRTUAL_SECTION_HAS_BODY` | 1 | error | linter / editor |
| `DEPRECATED_TEMPLATE` | 1 | warning | tooling surface |
| `MISSING_DESCRIPTION` | 1 | warning | tooling surface |
| `CIRCULAR_COLLECTION_INHERITANCE` | 2 | error | collection viewer |
| `UNKNOWN_COLLECTION` | 2 | error | registry validator |
| `UNIMPLEMENTED_VIRTUAL_TEMPLATE` | 2 | error | conformance test |
| `OVERRIDE_OF_FINAL_TEMPLATE` | 2 | error | conformance test |
| `CIRCULAR_INHERITANCE` | 3 | error | inheritance viewer |
| `OVERRIDE_OF_FINAL_SECTION` | 3 | error | inheritance viewer |
| `UNIMPLEMENTED_VIRTUAL_SECTION` | 3 | error | conformance test |
| `TYPE_MISMATCH` | 5/9 | error | render |

| Code | Phase | Severity | Catch by |
|---|---|---|---|
| `CONSTRAINT_VIOLATION` | 5/9 | error | render |
| `SELECT_INVALID_OPTION` | 5/9 | error | render |
| `IMMUTABILITY_VIOLATION` | 6 | error | save-time validation |
| `ATTACHMENT_NOT_FOUND` | 7 | error | storage validator |
| `ATTACHMENT_REQUIRED` | 7 | error | instantiation dialog |
| `CIRCULAR_INCLUDE` | 8 | error | include graph viewer |
| `AMBIGUOUS_INCLUDE` | 8 | error | include graph viewer |
| `INCLUDE_OF_VIRTUAL_SECTION` | 8 | error | include graph viewer |
| `UNDECLARED_INCLUDE_PARAMETER` | 8 | error | dependency check |
| `UNDEFINED_REFERENCE` | 2/3/8/9 | error | dependency check |
| `UNDEFINED_PARAMETER` | 9 | error | parameter dialog |
| `PARAMETER_NAME_IS_PREFIX` | 9 | warning | linter / editor |
| `MALFORMED_SCOPE` | 10 | error | linter / render |
| `INVALID_ASSERTION` | 10 | error | linter / render |
| `ASSERTION_FAILED` | 10 | error | render |

| Code | Phase | Severity | Catch by |
|---|---|---|---|
| `RESOURCE_LIMIT_EXCEEDED` | any (8) | error | render (documented limit) |

## 31.5 — Style of error reporting

Error-reporting UX is *not* normative — `§10` covers only Resource Limits (§10.1) and Security (§10.2), and explicitly notes that performance, tooling, and error-reporting guidance is non-normative. Still, the qualities that make a diagnostic usable are worth stating ; good practice is for an implementation to provide :

- Clear error messages with context.
- Line and column information for syntax errors.
- Dependency-chain information for resolution errors.
- Suggestions for common mistakes.

The first three are non-negotiable for a tool a human will use. The fourth is the difference between a helpful tool and a frustrating one. The IziBuddy reference implementation surfaces suggestions for typical typos (a misspelled `inherits` target suggests the closest existing id by string distance) ; other implementations may take similar approaches.

A *batch* renderer — one that processes many templates in a single CI run, for example — should aggregate errors per prompt rather than failing at the first one. The audit value of a complete error report is much greater than that of « your library has a problem somewhere ».

The next chapter formalizes the *conformance levels* an implementation may claim, and what each implies.

# Chapter XXXII — Conformance Levels

> ***`standard.md §8.1` — Conformance Levels (excerpt):***
>
> *A conforming implementation declares a single conformance level N (1 ≤ N ≤ 5); declaring **POO²L Level N** means it implements all of levels 0 through N. Level 0 (unstructured text) is a MANDATORY BASELINE but is NOT a claimable level — the minimum claimable badge is `POO²L 1.0 Level 1`. LEVEL 1 (structured sections; identity in the source: `@@@template` + `@@@name` required, `@@@description` OPTIONAL — its absence draws the MISSING_DESCRIPTION warning); LEVEL 2 (sections + includes); LEVEL 3 (structured data and validation — parameter sets, parameters/constants/attachments, the `select` type and declarative constraints, `@@@deprecated` / `@@@alias`, and the `@@@assert` + `@@@scope` validation layer); LEVEL 4 (Object-Oriented — prompt inheritance, OO #1, with the full section-modifier toolbox: `@@@section virtual`, `@@@section final`, `?`/`!` presence toggles, `parent`/`self`, and the INVALID_SYNTAX / VIRTUAL_SECTION_HAS_BODY / OVERRIDE_OF_FINAL_SECTION / UNIMPLEMENTED_VIRTUAL_SECTION / INCLUDE_OF_VIRTUAL_SECTION errors); LEVEL 5 (full OO² — the second OO axis on collections: the `inherits` clause on a `@@@collection` header, the `virtual` and `final` modifiers on a member's `@@@template` header, member override, the collection-axis `^parent.<template_id>` / `^self.<template_id>` references, containment nesting, and the `CIRCULAR_COLLECTION_INHERITANCE` / `UNKNOWN_COLLECTION` / `AMBIGUOUS_COLLECTION` / `UNIMPLEMENTED_VIRTUAL_TEMPLATE` / `OVERRIDE_OF_FINAL_TEMPLATE` errors). Every conforming implementation MUST publish a badge of the form `POO²L <version> Level <N>`. Folders are cross-level and MUST be supported by every conformant implementation. The three WARNING condi-*

> *tions — MISSING_DESCRIPTION, DEPRECATED_TEMPLATE, PARAMETER_NAME_IS_PREFIX — are NOT part of the conformance SHALL set : handling them is implementation policy and does not affect conformance.*
>
> ***`standard.md §8.2` — Validation (excerpt) :***
>
> *Conforming processors SHALL : accept Level 0 templates ; parse all valid POO²L syntax for the claimed level ; generate correct output per the processing model ; detect and report all specified error conditions ; preserve semantic equivalence across transformations ; correctly implement both attachment modes ; correctly handle optional parameters and attachments ; validate `select` parameters against their option lists ; validate parameter values against declarative constraints ; resolve alias references to canonical ids. The WARNING conditions (DEPRECATED_TEMPLATE, MISSING_DESCRIPTION, PARAMETER_NAME_IS_PREFIX) are NOT covered by this SHALL : handling them is implementation policy and does not affect conformance.*
>
> ***`standard.md §8.3` — Conformance Testing (excerpt) :***
>
> *Implementations claiming POO²L conformance MUST pass the official conformance test suite, document the conformance level claimed, and document any implementation-specific extensions or behaviors. The pack EXISTS and ships with the standard (`POO²L/standard-conformance-tests/`, several hundred cases across L0–L5, exercised by the reference runner). Use of the « POO²L » trademark requires demonstrated conformance.*

An implementation that calls itself a POO²L processor claims a *conformance level* — the set of language features it implements correctly. The six levels mirror the six architectural levels from Chapter II : an implementation that handles L5 templates handles L0 templates too, by construction. This chapter walks the conformance claims, their requirements, and what a conformance test looks like.

## 32.1 — The conformance levels

`§8.1` :

- **Level 0 (baseline)** — accepts and processes *unstructured templates* (plain text). Required for every implementation that claims any POO²L conformance at all, but it is NOT a claimable level : a Level 0 badge is forbidden, since any passthrough trivially « supports » it. The minimum claimable badge is `POO²L 1.0 Level 1`. An implementation that refuses an L0 prompt is not conformant.

- **Level 1 conformance** — Level 0 plus *structured sections*. The implementation handles `@@@section <name>` declarations (and the optional and dotted sub-section variants) and renders sections as part of the output. Identity is now WRITTEN IN THE SOURCE : the `@@@template` header (carrying the id) and `@@@name` are MANDATORY ; `@@@description` is OPTIONAL, and its absence draws only the advisory `MISSING_DESCRIPTION` warning ( `§5.2` ), never a conformance failure.

- **Level 2 conformance** — Level 1 plus *includes*. The implementation resolves `@@@{include <ref>}` references including the section-selection form ( `<prompt>.<section>` ), the parameter-override form ( `<ref>(p=v, ...)` ), and the cycle-detection requirement.

- **Level 3 conformance** — Level 2 plus *structured data and validation*. The implementation supports the data layer and the declarative-validation layer :

- Parameter sets (definition and binding).
- Parameters, constants, and attachments in the two modes ( `fixed` , `to-include` ), including the optional `attachment?` flavour for `to-include` .
- The `select` parameter type with inline options.
- Declarative constraints ( `min` , `max` , `minlen` , `maxlen` , `pattern` ).

- The `deprecated` marker and the DEPRECATED_TEMPLATE warning.
- The `alias` mechanism and the ALIAS_CONFLICT error.
- The `@@@assert` / `@@@scope` validation layer, with the ASSERTION_FAILED and MALFORMED_SCOPE error checks.
- **Level 4 conformance** — Level 3 plus *prompt inheritance* (OO #1). The implementation processes the inline `inherits <parent>` clause on the `@@@template` header, performs section merging and override semantics during Phase 3, and supports the `parent` / `self` reference family for inheritance-aware navigation. L4 also brings the full inheritance toolbox at section scope :
- **`@@@section virtual`** — abstract slot ; concrete descendant MUST implement. UNIMPLEMENTED_VIRTUAL_SECTION fires when no descendant overrides at the leaf.
- **`@@@section final`** — sealed slot ; descendants MUST NOT redeclare. OVERRIDE_OF_FINAL_SECTION fires on a redeclaration.
- **Presence toggles `?` / `!`** — user choice at instantiation, default OFF / default ON respectively ; description is the dialog text, never rendered, and mandatory (TOGGLABLE_SECTION_WITHOUT_DESCRIPTION).
- The INVALID_SYNTAX, VIRTUAL_SECTION_HAS_BODY, and INCLUDE_OF_VIRTUAL_SECTION error checks.
- **Level 5 conformance** — full OO². Every L4 capability plus the second OO axis on collections :
- **Collection inheritance** — entities declared by a `@@@collection` block, the `inherits` clause on that header, the effective-template-set computation, member override.
- **Virtual templates** — the `virtual` modifier on a member's `@@@template` header, the UNIMPLEMENTED_VIRTUAL_TEMPLATE check.

- **Final templates** — the `final` modifier on a member's `@@@template` header, symmetric to virtual, sealing the prompt against override by derived collections. OVERRIDE_OF_FINAL_TEMPLATE fires on a redeclaration.
- The collection-axis `^parent.<template_id>` and `^parent.<template_id>.<section>` references (and `^self.<template_id>`) from a prompt rendered inside a derived collection, plus containment nesting to unbounded depth.
- The CIRCULAR_COLLECTION_INHERITANCE, UNKNOWN_COLLECTION and AMBIGUOUS_COLLECTION error conditions.

The spec calls out two important non-features :

- **Folders** (a storage location with no `@@@collection` block) are *cross-level*. Every conforming implementation, regardless of claimed level, must support folders. Folders are not a Level 5 requirement.

- **Implementations conforming below Level 5** MUST NOT silently give a `@@@collection` block a DEGRADED meaning. Faced with an `inherits` clause (or another OO² construct) it does not implement, an implementation MUST either reject the input or preserve it unrendered and refuse to render that collection — never quietly treat the members as a flat folder namespace. Silent degradation would make the same source yield different results across engines ; that is forbidden.

## 32.2 — What a conformance test looks like

`§8.2` lists what a conformant processor SHALL do ; `§8.3` lists the supporting expectations. Together they constitute the checklist a conformance suite implements. The full checklist :

- Accept Level 0 templates (unstructured text).
- Parse all valid POO²L syntax for the claimed level.
- Generate correct output per the processing model.
- Detect and report all specified error conditions.

- Preserve semantic equivalence across transformations (whitespace normalization, comment removal, alias resolution).
- Correctly implement both attachment modes (`fixed`, `to-include`).
- Correctly handle optional parameters and attachments.
- Validate select-type parameters against the declared options list.
- Validate parameter values against declarative constraints when present.
- Resolve alias references to canonical prompt identifiers.

The three WARNING conditions — `DEPRECATED_TEMPLATE`, `MISSING_DESCRIPTION`, and `PARAMETER_NAME_IS_PREFIX` — are deliberately ABSENT from this SHALL list. A processor MAY ignore, surface, or block on a warning; whichever it does, the choice never gates conformance (`§8.2`, `§6.2`).

A conformance test suite (`§8.3`) is a body of templates and expected outputs that an implementation either passes or fails. The official pack **exists** and ships with the standard (`POO²L/standard-conformance-tests/`): several hundred cases spanning L0–L5, each with a source and an expected output or expected error, driven by the reference runner. `§8.3` says a conformant implementation MUST pass it. The IziBuddy reference implementation runs the pack in CI (and mirrors the L5 collection fixtures through its Lab `collections_oo` batch).

## 32.3 — What « claiming » a level means

An implementation *claims* a level by passing the corresponding test suite. The claim is a contract: every L-conformant implementation must accept every L-conforming prompt and produce the canonical output (up to whitespace normalization).

Two implementations at the same level produce the same output for the same input. This is the *interoperability* promise. A prompt rendered by one L5 implementation today must, ignoring the model's own non-determinism, render to the same text from any other L5 implementation tomorrow.

Use of the « POO²L » trademark requires demonstrated conformance ( `§8.3` ). Implementations that have not (yet) passed the suite may describe themselves as « POO²L-inspired » or « POO²L- compatible » but should not use the trademark itself.

## 32.4 — The reference implementation

`§8.1` , *Note (reference implementation status)* : « IziBuddy is the POO²L reference implementation, LEVEL 5 conformant (OO #1 on templates and OO #2 on collections). » OO #1 on templates has been part of IziBuddy since the inheritance feature shipped ; OO #2 on collections shipped together with this specification revision. The conformance pack exists and the engine passes it (the reference runner drives the full L0–L5 pack green in CI), so the L5 claim is demonstrated, not provisional.

The reference implementation's role is to serve as the *disambiguator* when the spec's prose leaves room for interpretation. Where a clause says « implementations MAY … », the reference implementation's choice is descriptive, not normative : other implementations are free to make different choices in those spaces. Where a clause says « implementations SHALL … », the reference implementation's behaviour is the canonical reading.

The reference implementation also hosts the conformance test fixtures. The IziBuddy Lab's `collections_oo` batch is the canonical L5 collection-feature test suite ; other implementations may use it to validate their own OO² behaviour.

## 32.5 — Tactical advice for implementers

A new implementation typically targets L0 → L5 in stages. The recommended order, validated by experience with the reference implementation :

1. **L0 + L1.** Parse text, recognize sections, strip directives at Phase 11. Two hundred lines of code, no external dependencies.

2. **L2.** Add include resolution and cycle detection. The storage abstraction now matters : includes resolve against « the library ».

3. **Tooling.** Linters, syntax highlighters, dependency viewers. Conformance is technically achievable without these, but a library above ten templates is unusable without them.

4. **L3 data and validation machinery.** Parameters, constants, attachment modes, `select`, declarative constraints, parameter sets, `deprecated` / `alias`, and the `@@@assert` / `@@@scope` validation layer. The error catalog (Chapter XXXI) starts to require systematic handling.

5. **L4 prompt inheritance.** Add `inherits` resolution, section merging, `parent` / `self`, `virtual` / `final` sections. The phase model becomes worth implementing explicitly ; many implementations refactor at this point.

6. **L5 collection axis.** The `@@@collection` block, the effective prompt set, virtual templates, the second `parent` form. This is the largest single jump and the one most likely to reveal bugs in the earlier phases.

An implementation that stops at L4 is a useful, conformant, production-grade POO²L processor. L5 is for libraries that have outgrown L4’s reach.

## 32.6 — Related features and why

**Six levels** (Chapter II). Conformance levels and architectural levels are the same six steps viewed from two sides — the language side (what a prompt may contain) and the implementation side (what a processor must support).

**Closed-extension policy** (Chapter XXXIII). Conformance is *meaningful* because the language does not extend. An implementation that today claims L5 conformance is forever allowed to claim it ; v1.0 templates will always render correctly.

**Error catalog** (Chapter XXXI). A conformance test suite is, in practice, a test of the error catalog : every error condition must fire when it should, and the catalogue of valid templates must render correctly.

The next chapter is the closed-extension policy itself — the reason this chapter’s promises hold.

# Chapter XXXIII — The Closed-Extension Policy

> ***`standard.md §1` — Scope (excerpt):***
>
> *This specification is COMPLETE and CLOSED. There are no planned extensions, variants, or alternative syntaxes. Specifically PROHIBITED: control flow (if/else, loops, while), dynamic code generation, macro systems beyond defined mechanisms, alternative syntaxes with different semantics. This ensures POO²L remains deterministic, testable, and maintainable.*
>
> ***`standard.md §9.1` — No Extensions Permitted (excerpt):***
>
> *This specification is COMPLETE and CLOSED. The following are explicitly PROHIBITED: control flow statements (`if`, `else`, `switch`, `case`); looping constructs (`for`, `while`, `loop`); dynamic code generation or evaluation; macro systems beyond includes; alternative syntaxes with different semantics; Turing-complete computation. Rationale: adding any of those would introduce non-determinism, complicate testing, reduce traceability, and create security risks.*
>
> ***`standard.md §9.2` — Extension Proposals (excerpt):***
>
> *Proposals for future versions MAY be submitted to `standard@poo2l.com` with detailed rationale and use cases, backward-compatibility analysis, and a reference implementation if possible. The standard's steward evaluates against alignment with POO²L design principles, ecosystem impact, implementation complexity, and backward compatibility. Submission does not guarantee inclusion — proposals that conflict with POO²L's core principles MAY be rejected.*

> ***`standard.md §9.3` — Implementation Extensions (excerpt) :***
>
> *POO²L distinguishes LANGUAGE-LEVEL (CLOSED : implementations MUST NOT add keywords, alter semantics, create alternative syntaxes or incompatible dialects, or use « POO²L » for non-conforming implementations) from APPLICATION-LEVEL (OPEN : tooling, performance optimizations, alternative storage backends, integrations, UI/UX enhancements, additional error messages, and application metadata stored OUTSIDE prompt text — user-specific parameter sets, dates, tags, flags, access control, version history, usage statistics).*

POO²L v1.0 is closed. The specification says so in three places — the preamble, `§1` Scope, and `§9.1` — and means it. This chapter unpacks what « closed » excludes, what it does *not* exclude, the process by which a future revision could happen, and the practical guidance the spec gives implementers who want to extend their *tool* without extending the *language*.

## 33.1 — What is closed

`§9.1` makes the boundary explicit. The following are PROHIBITED :

- **Control flow.** No `if`, `else`, `switch`, `case`.
- **Looping constructs.** No `for`, `while`, `loop`.
- **Dynamic code generation or evaluation.** No runtime expression evaluation.
- **Macro systems beyond defined mechanisms.** Includes are defined ; arbitrary macro expansion is not.
- **Alternative syntaxes with different semantics.** No JSON, YAML, or XML « dialect » of POO²L.
- **Turing-complete computation.** The language must remain decidable.

The rationale (`§9.1`, *Rationale*) : adding any of those features would compromise the four properties the closed specification guarantees — determinism, testability, traceability, and the absence of security risks rooted in dynamic evaluation.

The spec's framing closes the door on *extensions*, not on *restrictions*. An implementation may refuse to support a specific construct (a renderer aimed at L3 templates may decline L4 input) ; but it may not *add* to the language.

## 33.2 — What is not closed

`§9.3` draws the boundary on the other side. The closed- extension policy applies to **language-level** features — syntax, semantics, the meaning of constructs. **Application- level** features are open and welcomed :

- Additional tooling (editors, validators, debuggers, formatters).
- Performance optimizations (caching, lazy evaluation, parallel rendering).
- Alternative storage backends (filesystems, databases, cloud blob stores).
- Integration with external systems (Git, CI, model APIs).
- UI/UX enhancements (dashboards, diff viewers, search).
- Additional error messages or warnings (beyond the spec's required catalog).
- Application metadata stored *outside* prompt text : user-specific parameter sets, modification dates, tags beyond the spec's, flags, access control, version history, usage statistics.

The split between « in prompt text » and « in sidecar metadata » is the line. Anything in the prompt's body must be valid POO²L v1.0 source ; anything in the storage layer around the prompt is the implementation's choice.

The rationale (`§9.3`, *Rationale*) : a user with read-only access to a shared prompt may want to save their personal parameter sets. Those CANNOT live in the read-only prompt text ; implementations MAY store them externally. This is an application-level feature, not a language extension.

## 33.3 — Why the closure is the point

A language that allows arbitrary extension fragments quickly. JavaScript, SQL, regular expressions — every widely-used language with an « extensible » syntax accumulates dialects that nominally share a name but in practice cannot run each other's code. The author of a SQL query that worked in MySQL 5.7 discovers, three years later, that the same query no longer runs in MySQL 8 or PostgreSQL because some clause is a « vendor extension ».

POO²L's closure is the discipline that prevents this. A prompt written today against v1.0 will compile against v1.0 forever, on every implementation that claims v1.0 conformance. A new implementation that wants to add an `if` keyword cannot call itself a POO²L implementation ; it must invent its own name and accept that templates written for it will not be portable.

The closure is also the foundation of *trust*. A team adopting POO²L is implicitly making a long-term bet : their library will remain renderable. The bet is good because the language commits, in writing, not to change.

## 33.4 — Future versions

`§9.2` describes the process by which a future POO²L could happen :

- **Submission** : proposals are sent to `standard@poo2l.com`.
- **Required content** : detailed rationale and use cases ; backward-compatibility analysis ; a reference implementation if possible.

- **Evaluation criteria** : alignment with POO²L's design principles, ecosystem impact, implementation complexity, backward compatibility.

The spec is explicit that submission does not guarantee inclusion : the steward reserves the right to reject proposals that conflict with POO²L's core principles.

A single steward raises the obvious question of what happens if that steward disappears, and the spec answers it rather than leaving it open. Under CC BY-ND the text would stay redistributable but frozen for good — nobody could correct or extend it. Section 7.5 removes that lock in advance : on dissolution without a successor rights-holder, or twenty-four months of silence, the specification relicenses to CC BY 4.0 and derivatives become permitted, provided they do not present themselves as the canonical 1.0.

If a v2 ever happens, it will be a *new specification* with its own version number. The relationship to v1.0 is that of a parallel standard, not a successor that obsoletes v1.0. v1.0 templates will continue to compile under v1.0 implementations indefinitely ; v2 implementations may also accept v1.0 input, but that compatibility is a v2 design question, not a v1.0 commitment.

This is the same model used by mature standards bodies for languages where stability matters more than fashion. ISO C adopts new features through new revisions ; C89, C99, C11, C17, C23 are all distinct languages that conformant compilers may target separately.

## 33.5 — Implementer's perspective

The closed-extension policy is liberating, not constraining. An implementation team building a POO²L processor knows the full feature set on day one. The spec does not change under them ; their conformance test suite, once green, remains green through any number of internal refactors.

The same team can innovate freely on the application side. A new diff viewer, a smarter linter, a faster renderer, a richer storage backend — none of these are language extensions ; all are improvements the trademark welcomes.

The two layers — language and application — are also where *competition* happens. Implementations compete on application- level value : speed, ergonomics, integrations, user experience. They do not compete on what the language means. A prompt written for one implementation works on every other.

## 33.6 — Author's perspective

For an author of POO²L templates, the closed-extension policy is the answer to a question they may not have known to ask : *will this prompt I am writing today still work in five years ?*

The answer is yes. The language does not change. An implementation may go out of business ; another will replace it. The prompt source — every character of it — is portable.

A second consequence, less obvious : the *patterns* a team develops on top of POO²L are also forever. The convention that every reviewer collection declares `summarize`, `score`, and `suggest` virtuals is a *pattern*, not a *feature* ; but because the underlying mechanisms (collections, virtual templates, virtual signatures) are stable, the pattern is too.

The next chapter — the last of the reference part — collects the practical guidance for implementers that the spec scatters through `§10`.

# Chapter XXXIV — Implementer Notes

> ***`standard.md §10.1` — Resource Limits (excerpt):***
>
> *Implementations MAY impose limits — e.g. include-nesting depth, inheritance-chain length, comment/verbatim nesting depth, attachment size, member count per collection. Such limits are implementation-defined but MUST be documented, and exceeding one MUST be reported as `RESOURCE_LIMIT_EXCEEDED` (§6.2), naming the limit — never as a language-level error.*
>
> ***`standard.md §10.2` — Security Considerations (excerpt):***
>
> *Implementations MUST: validate all include references; prevent path-traversal attacks in includes and attachment paths; sanitize attachment content; enforce every configured access-control policy on collections; prevent resource exhaustion (include depth, attachment size limits, continuation-line counts, comment/verbatim nesting depth); validate attachment paths at instantiation time; implement secure defaults for all operations. These are MUST-level.*

The specification's `§10` is short: two subsections — Resource Limits (§10.1) and Security (§10.2). It explicitly notes that performance, tooling, and error-reporting guidance is NON-NORMATIVE and not part of the specification. This chapter first covers the two normative subsections, then offers the non-normative implementer guidance (performance, tooling, error reporting) that experience with the reference implementation has surfaced — clearly marked as guidance, not spec.

## 34.1 — Resource limits (§10.1, normative)

`§10.1` is the first of the two normative subsections. An implementation MAY impose limits — include-nesting depth, inheritance-chain length, comment/verbatim nesting depth, attachment size, member count per collection, continuation-line counts, and similar. Two rules bind :

- **Documented.** Every limit an implementation enforces MUST be documented, so a portable prompt author knows the floor they can rely on.
- **Reported as `RESOURCE_LIMIT_EXCEEDED`.** Exceeding a limit MUST be reported as the typed error `RESOURCE_LIMIT_EXCEEDED` (§6.2), naming the limit — NEVER disguised as a language-level error such as `CIRCULAR_INCLUDE`. A prompt that hits a depth cap did not necessarily contain a cycle ; the two conditions are distinct, and conflating them misleads the author.

The conformance pack exercises this : an include chain deeper than a configured floor must degrade gracefully to `RESOURCE_LIMIT_EXCEEDED` rather than crash or misreport.

## 34.2 — Performance considerations (non-normative guidance)

Performance is *not* part of `§10` — the spec calls this guidance non-normative. Still, four hot paths reward optimization :

- **Include caching.** Resolving an `@@@{include }` is a lookup in the storage backend. A library where many templates include the same fragment — a glossary, a safety preamble — is paying that lookup cost N times per render unless the resolver memoizes. The reference implementation caches resolved includes per render-context ; the cache is invalidated when the source changes.

- **Inheritance-chain pre-computation.** A child prompt's effective section list depends on its entire ancestor chain. Computing the chain on every render is wasteful when the chain is stable. The reference implementation pre-computes and caches the chain per prompt, invalidating it when any ancestor's source changes.

- **Parameter substitution efficiency.** Substitution in Phase 9 is a linear scan of the body, replacing `@@@=name` references with bound values. A naive string concatenation approach can be quadratic ; a buffer-based approach (collect fragments, join once) is the standard fix. The reference implementation uses a single pass with a `StringBuilder` - like buffer.

- **Attachment lazy loading.** Attachments — especially binary ones referenced by `to-include` slots — should not be loaded unless the render actually uses them. The reference implementation defers attachment loading to the model- invocation step ; if the model never asks for the attachment, the bytes never leave the storage backend.

Two further opportunities not in the spec :

- **Phase 1 result caching.** Parsing a prompt is deterministic ; the result depends only on the source. Caching parse trees per source content is a quick win for any implementation that renders the same prompt repeatedly.

- **Effective-template-set caching.** The Phase 2 effective set for a collection chain is stable as long as no member of the chain changes. Caching the effective set per leaf collection turns Phase 2 from « walk the chain every render » into « consult a hash table ».

## 34.3 — Tooling support (non-normative guidance)

Tooling is *not* part of `§10` either ; the spec marks it non-normative. A healthy POO²L ecosystem nonetheless tends to provide :

- **Syntax validators.** A standalone linter that reads a prompt source and reports every error from Chapter XXXI before the prompt is rendered. The validator is the first line of defence ; an editor that runs it on every save catches most authoring mistakes immediately.
- **Dependency analyzers.** A tool that reports the dependency graph (Chapter XXVIII) : what each prompt depends on, and what depends on each prompt. The reverse direction is the harder one and the more valuable one.
- **Visual editors with syntax masking.** A text editor that understands POO²L delimiters and renders the directives in a muted style, with autocomplete on prompt ids, parameter names, and section names. The reference implementation's smart editor is one example ; standalone POO²L plug-ins for other editors are welcome additions to the ecosystem.
- **Testing frameworks.** A framework for asserting « this prompt with this binding produces this output ». The reference implementation's Lab feature is one example ; a CI-friendly test runner is a natural complement.
- **Version control integration.** Tooling that translates between the implementation's internal version representation and Git history. The reference implementation maintains a Git mirror of every prompt's source ; other implementations may take similar approaches.
- **Attachment management tools.** Browsers, validators, and size monitors for attached files. A library with many attachments needs the same tooling that a library with many binary assets does.

- **Conformance test runners.** A runner that takes the official test suite (when released) and reports pass/fail per fixture. Implementations claiming conformance run this in their CI.

## 34.4 — Security considerations (§10.2, normative)

`§10.2` is the second normative subsection and lists seven security requirements (these are MUST-level, not SHOULD):

- **Validate all include references.** Untrusted prompt sources can attempt to include from unexpected paths; implementations MUST sanitize.
- **Prevent path traversal attacks.** Both in include references and in `fixed` attachment paths. `..` and absolute paths that escape the prompt scope must be rejected.
- **Sanitize attachment content.** Attachment payloads passed to downstream model APIs must be sanitized for context-injection attacks where applicable.
- **Enforce access control on collections.** The collection- level read/write/inheritance/include restrictions of `§7.3` must be enforced; bypasses through alias references or collection-scope `parent` must close the same checks.

> ***`standard.md §7.3` — Access Control (excerpt):***
>
> *Collections MAY enforce: read/write permissions; inheritance restrictions; include scope limitations; attachment access restrictions. The MAY framing means an implementation without*

> *group-scoped access control is conformant — but where the policy exists, the four boundary types are what the language allows it to attach to.*

- **Prevent resource exhaustion.** Include depth, inheritance depth, attachment size, continuation-line counts, and comment/verbatim nesting depth must have documented upper bounds; a breach is reported as `RESOURCE_LIMIT_EXCEEDED` (§10.1). A maliciously deep inheritance chain that consumes hours of CPU is a denial of service.

- **Validate attachment paths at instantiation time.** The path the caller supplies in a `to-include` slot must be validated for traversal and for access permissions before the storage backend reads the file.

- **Implement secure defaults.** Whenever the spec leaves a policy choice to the implementation, the default should err toward restriction: tighter access control by default, smaller resource limits by default, more verbose audit logging by default.

A few further notes that experience with the reference implementation has highlighted:

- **Verbatim is not a security boundary.** A verbatim block preserves its content verbatim; it does not sanitize it. An author who pastes user input into a verbatim block has put user input directly into the rendered prompt. Sanitize at the parameter layer if user input is involved.

- **Constraints are not security.** A `pattern` constraint validates *shape*, not *intent*. A `text` value that matches the regex but contains injection content is still injection content. Constraints are documentation and minimum bounds, not a security layer.

- **Aliases respect permissions.** A caller cannot use an alias to reach a prompt they would not be allowed to access by canonical id. The alias resolution in Phase 1 must run *before* the per-

mission check, not after — otherwise the permission check sees the original (aliased) id and may not catch the policy violation against the canonical target.

## 34.5 — Error reporting (non-normative guidance)

Error-reporting UX is *not* part of `§10` — the spec marks it non-normative. Good practice nonetheless favours four properties for every error the implementation reports :

- **Clear messages with context.** A bare « PARSE ERROR » is useless ; « INVALID_SYNTAX at line 14, column 8 : unknown keyword `iherits` (did you mean `inherits` ?) » is actionable.
- **Line and column information for syntax errors.** Editors need this to position cursors and underlines.
- **Dependency chain information for resolution errors.** A CIRCULAR_INHERITANCE error should print the chain ( `A → B → C → A` ) so the author can find the cycle. The reference implementation does so.
- **Suggestions for common mistakes.** Spell-check the identifier against existing templates in scope and offer the closest match. Recognize the typical pattern (a comment used where a deprecation marker was needed ; a constant re-declared in a child) and point to the correct construct.

An implementation that gives good error messages is, in practice, the difference between a tool authors enjoy using and a tool they tolerate. The investment in error reporting pays off on every render that fails.

## 34.6 — Operational closing notes

The spec ends `§10` here. Two practical points the implementation team will want to keep in mind :

- **The render budget is shared with the model.** A POO²L processor that takes 50 ms to render a 4000-token prompt is trivial overhead next to a 5-second model response. A processor that takes 5 seconds to render the same prompt doubles the user-perceived latency. Stay fast.
- **The audit trail is non-negotiable.** Every render's inputs (prompt source, parameter bindings, parameter set selection, instantiation user, render timestamp) and every render's warnings should be persisted somewhere inspectable. POO²L's determinism is only useful when the inputs are also traceable.

This closes Part VI. The appendices follow : the anti-patterns catalog, the cookbook of idiomatic recipes, and the crosswalk to the specification's section numbers. The normative EBNF is not reproduced — its single source of truth is the specification's own Appendix B ( `standard-syntax.md` ).

# Appendices

# Appendix C — Appendix C — Anti-Patterns and Refactorings

Each construct chapter (Chapters 6 through 28) closes with an anti-pattern : a piece of POO²L that *parses* but conveys an intent the language could have expressed more clearly. This appendix collects every such anti-pattern in one place, with the refactoring move that fixes each one. The audience is the author auditing a library and looking for shapes to improve.

## C.1 — Identity

**Duplicating intent across `name` and `description`.**

```
@@@name Code Reviewer (review code for best practices)
@@@description Code Reviewer — review code for best practices
```

→ The `name` is a *handle* ; the `description` is a *sentence*. Keep them distinct :

```
@@@name Code Reviewer
@@@description Reviews code for correctness, performance, and
   adherence to language-specific best practices.
```

## C.2 — Sections

**Using a section to carry a comment.**

```
@@@section _comment
TODO: rewrite this section in a friendlier tone.
```

→ A section renders into the output. Comments do not :

```
@@@-- TODO: rewrite the task section in a friendlier tone.
```

## C.3 — Comments

**Using a comment for a deprecation notice.**

```
@@@-- DEPRECATED: please migrate to code_review_v3.
```

→ Comments are invisible to tooling. Use the `deprecated` marker (Chapter XXVI) :

```
@@@deprecated "Use code_review_v3 instead"
```

## C.4 — Verbatim

**Wrapping a real parameter declaration in verbatim by mistake.**

```
@@@verbatim
@@@param recipient : text, "Recipient name"
@@@end-verbatim
```

→ The verbatim wrapper shields the declaration from Phase 1. Remove it :

```
@@@param recipient : text, "Recipient name"
```

## C.5 — System variables

**Using `DATE` as a project deadline.**

```
The contract expires on @@@=DATE.
```

→ `DATE` is today's render-time date, not a fixed value. Use a constant:

```
@@@const EXPIRY : date = "2027-01-01", "Contract expiry"

The contract expires on @@@=EXPIRY.
```

## C.6 — Parameters

**Making a value a parameter when it never varies.**

```
@@@param company_name : text = "Acme Corp", "Company name"
```

→ The value never changes across callers. Make it a constant:

```
@@@const COMPANY_NAME : text = "Acme Corp", "Company name"
```

## C.7 — Constraints

**Cross-parameter validation in disguise.**

```
@@@param min_words : number min=0 max=10000 = 100, "Minimum"
@@@param max_words : number min=0 max=10000 = 500, "Maximum"
```

→ The implicit `max_words >= min_words` cannot be expressed in POO²L (constraints are single-parameter and non-computational — Chapter XXXIII). Either validate in the calling code or restructure to a single `word_target` parameter that the model treats as approximate.

## C.8 — Constants

**A constant that should be a parameter.**

```
@@@const CUSTOMER_TIER : select("free", "pro", "enterprise") = "pro"
```

→ If two operators disagree about the value, it is a parameter:

```
@@@param customer_tier : select("free", "pro", "enterprise") = "pro", \
  "Customer tier"
```

## C.9 — Parameter sets

**A parameter set inside a collection-member prompt.**

```
@@@template reviewers.python_reviewer
@@@name Python Reviewer

@@@parameter-set PythonPerf
language: "Python"
focus: "performance"
@@@end-parameter-set
```

→ Forbidden by `§5.9` *Restriction*. Declare the set in the collection's `@@@collection` block instead; the set then inherits along the collection chain and applies to every member with matching parameters.

## C.10 — Attachments

**Using an attachment for a small text value.**

```
@@@attachment tagline : txt fixed blob \
  "data:text/plain;base64,VGhlIGZ1dHVyZSwgcmVuZGVyZWQu"
```

→ The decoded payload is one line of prose. Use a constant:

```
@@@const TAGLINE : text = "The future, rendered.", "Brand tagline"
```

## C.11 — Includes

**Using include where inheritance was needed.**

```
@@@include code_reviewer.role
@@@include code_reviewer.format

@@@section task
Review the attached Python code...
```

→ The author wanted « inherit everything ; override one section » :

```
@@@template python_code_reviewer inherits code_reviewer

@@@section task
Review the attached Python code...
```

## C.12 — Includes with overrides

**Overriding a parameter the target does not declare.**

```
@@@include templates/greeting(undefined_param="x")
```

→ Phase 8 (include resolution) raises UNDECLARED_INCLUDE_PARAMETER once the target is known. Read the target’s declarations and override only the parameters that exist.

## C.13 — Inheritance

**Using inheritance for content reuse.**

```
@@@template customer_reply inherits email_opener
```

```
@@@section body
...
```

→ The opener is not a *kind of* customer reply; it is a fragment. Use include:

```
@@@include email_opener

@@@section body
...
```

## C.14 — `parent` and `self`

**Using `parent` outside an inheritance chain.**

```
@@@section role
@@@include parent.role
```

→ No parent exists to resolve against. Either declare an `inherits`, or replace `parent.role` with a plain include of the desired prompt:

```
@@@section role
@@@include shared/standard_role
```

## C.15 — Folders

**Using a folder for a contract.**

```
Folder "reviewers":
  a storage location with no @@@collection block

(every member is expected to declare three specific sections,
 by convention)
```

→ Conventions decay. Promote to a collection — declare a `@@@collection` block (and qualify the members' ids) — and capture the contract as virtual templates (Chapter XXIII):

```
@@@collection reviewers
@@@template base
@@@end-collection

Virtual member with declared signature, declared in the collection.
```

## C.16 — Collections

**Using a collection where a folder was needed.**

```
@@@collection scratch
@@@end-collection

(members are unrelated half-finished templates)
```

→ Demote to a folder; the prompt sources are unchanged.

## C.17 — Collection inheritance and virtuals

**A collection inheriting from a folder.**

```
@@@collection marketing inherits drafts
@@@end-collection
  (where "drafts" is a plain folder, not a declared collection)
```

→ UNKNOWN_COLLECTION. Either promote `drafts` to a collection, or remove the `inherits`.

**A circular collection chain.**

```
Collection A inherits B
Collection B inherits A
```

→ CIRCULAR_COLLECTION_INHERITANCE. One direction must be the true ancestor ; the other should not inherit at all.

## C.18 — Section composition

**Accidental override.**

```
@@@section format
Use a friendly tone.
```

→ The author intended to extend the parent's `format`. The override discarded it. Use extend :

```
@@@section format
@@@include parent.format
Use a friendly tone.
```

**Empty override as a delete.**

```
@@@section format
```

→ The empty override hides the parent's content from the render but still leaves a section visible to other references. The right fix is upstream : if the section should not exist for this derived prompt, the chain should not have declared it.

## C.19 — Aliases

**Aliases as a synonym dictionary.**

```
@@@alias respond_to_customer
@@@alias answer_client
@@@alias reply_message
```

→ Aliases solve real renames. Synonyms belong to documentation. Remove the extra aliases.

## C.20 — Deprecation

**Deprecation as a comment.**

```
@@@-- DEPRECATED: migrate to code_review_v3
```

→ Use the marker:

```
@@@deprecated "Use code_review_v3 instead"
```

## C.21 — Versioning

**Modifying a published version in place.**

→ Procedural anti-pattern, not a syntactic one. Every edit to a published version produces a new version. Implementations SHOULD enforce immutable versions; the procedural discipline is the author's.

## C.22 — Access control

**Assuming the language enforces what the implementation does not.**

```
@@@deprecated "Internal only — do not share externally"
```

→ The marker is informational; it has no permission effect. Use the implementation's permission system (groups, scopes, ACLs).

## C.23 — A general pattern

Most of the anti-patterns above share a single shape: *the author reached for a construct that almost-but-not-quite fit the intent*. The fix is the construct that fits exactly. The language is rich enough that the right construct usually exists; the discipline is to pause long enough to find it.

The next appendix is a cookbook of *idiomatic* patterns — the ones that hold up over time.

# Appendix D — Appendix D — Cookbook

Ten short recipes for idiomatic POO²L patterns. Each recipe states the *problem*, the *shape*, and the *why*; the recipes are self-contained but cross-reference the chapter where each construct is treated.

The recipes target L2 through L4. A library at L0 or L1 has no use for the patterns in this appendix; the move to L2 is itself the first recipe.

---

## D.1 — Recipe 1 — A parameterized email template

**Problem**: write one email prompt that adapts to several recipients and tones without duplicating the body across files.

**Shape**:

```
@@@template email_drafter
@@@name Email Drafter
@@@description Drafts an email in a chosen tone

@@@param recipient : text, "Recipient name"
@@@param tone : select("formal", "casual", "warm") = "formal", "Tone"
@@@param topic : text, "Email subject in one sentence"

@@@section greeting
Dear @@@=recipient,

@@@section body
Write the email in a @@@=tone tone, about: @@@=topic.

@@@section sign_off
Best regards.
```

**Why**: three parameters (recipient, tone, topic) cover every useful variation. A `select` on tone restricts the value space to three options — the model and the renderer both benefit.

→ Chapter XI (parameters), Chapter XII (constraint via `select`), Chapter VII (sections).

---

## D.2 — Recipe 2 — A polymorphic collection of role templates

**Problem**: maintain a family of `reviewer` templates where every member implements the same three abstract operations.

**Shape**: a base collection with a virtual `task` prompt and two derived collections that each provide a concrete `task`.

```
Collection "reviewers" (declared by a @@@collection block):

  Virtual member task:
    @@@template virtual reviewers.task
    @@@name Review Task
    @@@description Abstract task to be implemented by derived
                   reviewers
    @@@param topic : text, "What to review"

Collection "code_reviewers" (a @@@collection block, inherits reviewers):

  Concrete member task:
    @@@template code_reviewers.task
    @@@param topic : text, "What to review"
    Review the code @@@=topic for correctness and style.

Collection "prose_reviewers" (a @@@collection block, inherits reviewers)

  Concrete member task:
    @@@template prose_reviewers.task
```

```
    @@@param topic : text, "What to review"
    Read @@@=topic for clarity and pacing.
```

**Why**: virtual templates express the contract; derived collections fulfill it. A caller of `code_reviewers.task` and a caller of `prose_reviewers.task` get the same signature, two different bodies. → Chapter XXIII.

---

## D.3 — Recipe 3 — A deprecation migration with alias

**Problem**: rename `code_review_v2` to `code_review_v3` without breaking existing references.

**Shape**:

```
New prompt (the successor):
  @@@template code_review_v3
  @@@name Code Review v3
  @@@alias code_review_v2
  @@@description Latest reviewer; supersedes v2
  ...

Old prompt (the deprecated stub, kept for the transition window):
  @@@template code_review_v2_stub
  @@@name Code Review v2 (deprecated)
  @@@deprecated "Use code_review_v3 instead"
  ...
```

After the migration window — once tooling reports zero references to the v2 name — the stub can be deleted and the alias on the v3 prompt can be removed.

→ Chapter XXV (aliases), Chapter XXVI (deprecation).

---

## D.4 — Recipe 4 — Splitting a giant prompt into includes

**Problem**: a single prompt has grown past two hundred lines and is hard to navigate.

**Shape**: identify the natural sub-sections (role, glossary, constraints, format) and extract each into its own prompt:

```
Original (200 lines):
  @@@section role
  ...50 lines...
  @@@section glossary
  ...80 lines...
  @@@section task
  ...20 lines...
  @@@section format
  ...50 lines...

After:
  @@@template reviewer_composed
  @@@name Reviewer (composed)
  @@@description Composed reviewer assembled from four blocks

  @@@include shared/reviewer_role
  @@@include shared/reviewer_glossary

  @@@section task
  ...20 lines (local to this reviewer)...

  @@@include shared/reviewer_format
```

**Why**: the body becomes a *table of contents*; each block lives in its own file and can evolve independently. The local `task` section is the only piece truly unique to this reviewer. → Chapter XVII.

---

## D.5 — Recipe 5 — Promoting a folder to a collection

**Problem**: a `reviewers` folder has grown to ten members; the team realizes the members share a contract worth enforcing.

**Shape**: declare a `@@@collection` block for the entity (this is what promotes it), qualify the members' ids, and add a base virtual prompt:

```
Before:
  "reviewers" is a folder — a storage location with no @@@collection
  block; its members carry bare @@@template ids.

After:
  Declare the collection in-language (and qualify member ids with
  the "reviewers." prefix):

    @@@collection reviewers
    @@@template base
    @@@end-collection

  Add a virtual member to capture the contract:
    @@@template virtual reviewers.base
    @@@name Reviewer Base
    @@@description Contract for every reviewer
    @@@param topic : text, "Subject of the review"
    @@@param tone : select("formal", "casual") = "formal", "Tone"
```

The abstract member `base` is now a contract: any *derived* collection must supply a non-virtual `base` (a member with the same canonical id and a matching signature), or rendering that member raises UNIMPLEMENTED_VIRTUAL_TEMPLATE — exactly the enforcement the team wants. → Chapter XXIII.

---

## D.6 — Recipe 6 — A collection with one virtual and three concretes

**Problem**: a family of three writing templates that share a contract; the base contract may grow over time.

**Shape**: a `writers` collection with a virtual `compose` member and three concrete overrides.

```
Collection "writers":
  Virtual member compose (signature: topic, audience, length).

Collection "blog_writers" inherits writers:
  Concrete compose: produces a blog post.

Collection "email_writers" inherits writers:
  Concrete compose: produces an email.

Collection "memo_writers" inherits writers:
  Concrete compose: produces an internal memo.
```

A caller asks for `compose` on any of the three child collections; the polymorphic resolution returns the right concrete. Adding `announcement_writers` later is a single new derived collection; existing concretes are unaffected.

→ Chapter XXIII.

---

## D.7 — Recipe 7 — Localizing a base prompt through `LANG` and

parameter sets

**Problem**: a base reviewer prompt should produce output in the locale of the caller.

**Shape** : use `@@@=LANG` to surface the locale to the model, *and* use a collection-scope parameter set per locale to toggle locale-specific text :

```
Base prompt (in collection "reviewers"):
  @@@section task
  Reply in @@@=LANG. Use the appropriate professional register
  for that language.

Collection "reviewers" (@@@collection block, collection-level
parameter sets — each a @@@parameter-set block inside the collection)

  @@@collection reviewers
  @@@parameter-set en
  tone: "concise"
  @@@end-parameter-set
  @@@parameter-set fr
  tone: "formel"
  @@@end-parameter-set
  @@@parameter-set de
  tone: "sachlich"
  @@@end-parameter-set
  @@@end-collection
```

The caller selects the set matching the desired locale ; the `LANG` system variable also reaches the model directly.

→ Chapter X, Chapter XV.

---

## D.8 — Recipe 8 — Replacing a long parameter list with a parameter

set

**Problem** : a prompt has eight parameters ; the caller has to remember which combination produces which outcome.

**Shape** : declare parameter sets that capture the typical combinations :

```
@@@param tier : select("free", "pro", "enterprise") = "pro", "Tier"
@@@param budget : number min=0 max=1000000 = 5000, "Budget"
@@@param tone : select("warm", "consultative") = "warm", "Tone"
@@@param ...   (five more)

@@@parameter-set HighValueOutreach
tier: "enterprise"
budget: 250000
tone: "consultative"
...
@@@end-parameter-set

@@@parameter-set EarlyStage
tier: "free"
budget: 0
tone: "warm"
...
@@@end-parameter-set
```

The caller picks `HighValueOutreach` or `EarlyStage` and gets eight values bound at once. → Chapter XV.

---

## D.9 — Recipe 9 — Using `parent.<section>` to add an appendix

**Problem** : a child reviewer wants to add a closing « Improvements » appendix to the parent's `task` section.

**Shape** : extend, not override :

```
@@@template detailed_code_reviewer inherits code_reviewer

@@@section task
```

```
@@@include parent.task
After the review, list the top three improvements in order of
impact.
```

The parent's `task` content appears first; the child's appendix follows. → Chapter XX, Chapter XXIV (*extend* pattern).

---

## D.10 — Recipe 10 — Verbatim-quoting POO²L in a teaching prompt

**Problem**: a prompt teaches POO²L to a model and must quote POO²L syntax in its body without triggering substitution.

**Shape**: wrap every quoted POO²L sample in a verbatim block:

```
A required parameter with a label looks like:

@@@verbatim
@@@param recipient : text, "Recipient name"
@@@end-verbatim

You may declare a parameter optional by putting `?` after the
keyword:

@@@verbatim
@@@param? notes : textarea = "", "Additional notes"
@@@end-verbatim
```

The verbatim blocks survive every phase; the model sees the literal POO²L source. → Chapter IX.

---

## D.11 — A short reading of the cookbook

These ten recipes are not exhaustive — a real library will develop its own patterns over time. What they share is that each is *small*: a single problem, a single shape, a single construct as the centerpiece. Combining two or three recipes in the same prompt is the usual move.

The final appendix is the crosswalk between this book's chapters and the specification's section numbers.

# Appendix E — Appendix E — Crosswalk to `standard.md`

A two-column map between every numbered section of the POO²L v1.0 specification and the chapter (or chapters) of this book that cover it. The audience is the reader who has a `§X.Y` reference in hand — perhaps from another POO²L document, a linter's error message, or a discussion — and wants to find the pedagogical treatment in this volume.

The reverse direction — book chapter to spec section — is implicit at the top of every chapter of this book, where each chapter cites the spec section(s) it is based on.

---

## E.1 — Front matter and `§1`

| Spec section | This book |
|---|---|
| Preamble | Chapter I (Why POO²L Exists) |
| §1 Scope | Chapter I, Chapter XXXIII (closed extensions) |

## E.2 — `§2` Normative references

| Spec section | This book |
|---|---|
| §2 Normative References | Chapter V (mentioned) |

## E.3 — §3 Terms and definitions

| Spec section | This book |
|---|---|
| §3 (header) | Chapter IV |
| §3.1 Template | Chapter IV |
| §3.2 Section | Chapter IV, Chapter VII |
| §3.3 Include | Chapter IV, Chapter XVII |
| §3.4 Inheritance | Chapter IV, Chapter XIX |
| §3.5 Parameter | Chapter IV, Chapter XI |
| §3.6 Constant | Chapter IV, Chapter XIV |
| §3.7 Attachment | Chapter IV, Chapter XVI |
| §3.8 Parameter Set | Chapter IV, Chapter XV |
| §3.9 Template Collection | Chapter IV, Chapter XXII |
| §3.10 OO² | Chapter IV, Chapter XXIII |
| §3.11 Constraint | Chapter IV, Chapter XII |
| §3.12 Deprecated | Chapter IV, Chapter XXVI |
| §3.13 Alias | Chapter IV, Chapter XXV |

## E.4 — §4 Architectural levels

| Spec section | This book |
|---|---|
| §4 (header) | Chapter II |
| §4.1 Level 0 : Unstructured | Chapter II |
| §4.2 Level 1 : Structured | Chapter II, Chapter VII |

| Spec section | This book |
|---|---|
| §4.3 Level 2 : Procedural | Chapter II, Chapter XVII |
| §4.4 Level 3 : Structured Data and Validation | Chapter II + Chapters XI–XVI, XXV, XXVI |
| §4.4.1 Template Identity | Chapter VI |
| §4.4.2 Parameters | Chapter XI, Chapter XII |
| §4.4.3 Constants | Chapter XIV |
| §4.4.4 Attachments | Chapter XVI |
| §4.4.5 Parameter Sets | Chapter XV |
| §4.4.6 Deprecation | Chapter XXVI |
| §4.4.7 Alias | Chapter XXV |
| §4.4.8 Composition Semantics (Data Layer) | Chapter XXIV |
| §4.5 Level 4 : Object-Oriented | Chapter II, Chapter XIX |
| §4.6 Level 5 : OO² | Chapter II + Chapters XXII–XXIX |
| §4.6.1 Template Collections | Chapter XXII, Chapter XXIII, Chapter XX (collection-axis `^parent`) |
| §4.6.2 Composition Semantics (OO² axes) | Chapter XXIV, Chapter XIX |
| §4.7 Folders (cross-level) | Chapter XXI |

## E.5 — §5 Syntax specification

| Spec section | This book |
|---|---|
| §5.1 General Syntax Rules | Chapter IV, Chapter V |

| Spec section | This book |
|---|---|
| §5.1.2 Sigil escaping ( `@@@@` ) | Chapter IX |
| §5.1.3 Bounded (delimited) form | Chapter VIII, Chapter XVII |
| §5.1.4 Line continuation ( `\` ) | Chapter VIII |
| §5.2 Identity Syntax | Chapter VI |
| §5.3 Section Syntax (incl. sub-sections, placement) | Chapter VII |
| §5.4 Include Syntax | Chapter XVII, Chapter XVIII, |
| | Chapter XX (self/parent) |
| §5.5 Inheritance Syntax | Chapter XIX |
| §5.6 Parameter Syntax | Chapter XI, Chapter XII |
| §5.7 Constant Syntax | Chapter XIV |
| §5.8 Attachment Syntax | Chapter XVI |
| §5.9 Parameter Set Syntax | Chapter XV |
| §5.10 Collection Syntax | Chapter XXI, Chapter XXII |
| §5.11 Comment Syntax (header) | Chapter VIII |
| §5.11.1 Line, Bounded & Tail Comment | Chapter VIII |
| §5.11.2 Block Comment | Chapter VIII |
| §5.12 Verbatim Syntax | Chapter IX |
| §5.13 Deprecation Syntax | Chapter XXVI |

| Spec section | This book |
|---|---|
| §5.14 Alias Syntax | Chapter XXV |
| §5.15 Assertions & Scopes ( `@@@assert` / `@@@scope` ) | Chapter XIII |
| §5.16 Libraries ( `@@@library` ) | Chapter XXII (mentioned) |
| §5.17 Uses ( `@@@uses` ) | Chapter XVII (mentioned) |
| §5.7.1 System variables | Chapter X |

## E.6 — §6 Processing model

| Spec section | This book |
|---|---|
| §6.1 Resolution Phases | Chapter III (overview), Chapter XXX (depth) |
| §6.2 Error Conditions | Chapter XXXI |

## E.7 — §7 Governance and versioning

| Spec section | This book |
|---|---|
| §7.1 Template Versioning | Chapter XXVII |
| §7.2 Dependency Management | Chapter XXVIII |
| §7.3 Access Control | Chapter XXIX |

## E.8 — §8 Conformance

| Spec section | This book |
|---|---|
| §8.1 Conformance Levels | Chapter XXXII |

| Spec section | This book |
|---|---|
| §8.2 Validation | Chapter XXXII |
| §8.3 Conformance Testing | Chapter XXXII |

## E.9 — `§9` Extensibility and restrictions

| Spec section | This book |
|---|---|
| §9.1 No Extensions Permitted | Chapter XXXIII |
| §9.2 Extension Proposals | Chapter XXXIII |
| §9.3 Implementation Extensions | Chapter XXXIII |

## E.10 — `§10` Implementation notes

| Spec section | This book |
|---|---|
| §10.1 Resource Limits ( `RESOURCE_LIMIT_EXCEEDED` ) | Chapter XXXIV, Chapter XXXI |
| §10.2 Security Considerations | Chapter XXXIV |

## E.11 — Appendices of `standard.md`

| Spec appendix | This book |
|---|---|
| Appendix A — Complete Examples | Not reproduced — single source of truth is the spec ( `standard-syntax.md` , Appendix A) |
| Appendix B — Grammar (EBNF) | Not reproduced — single source of truth is the spec ( `standard-syntax.md` , Appendix B) |

---

## E.12 — A note on chapter-to-section mapping

A few chapters of this book span more than one spec section because the pedagogical unit and the normative unit do not align exactly. The places where this happens :

- **Chapter IV** (Terms cheatsheet) touches every entry in §3 but in a different order — by topic group rather than by §3 numbering.
- **Chapter XX** (parent and self) consolidates the inheritance-aware references from §5.4 with the collection-axis parent forms from §4.6.1. The spec discusses them in two places because the two axes are introduced separately ; the book treats them together because that is how an author thinks about navigation.
- **Chapter XXIV** (Section composition semantics) is the book's own consolidation of override/extend/merge patterns. The spec mentions the patterns implicitly in §4.5 and §5.4 (parent forms) but does not name them. The names (override/extend/merge) are this book's contribution.
- **Chapter XXX** (Resolution phases in depth) elaborates the same eleven phases (plus the 3.b call-contract sub-phase) that §6.1 lists, with explicit inputs and outputs per phase. The spec defines what the phases do ; the book describes how to implement them.

The crosswalk aims to cover every numbered section of the spec at least once. A handful of newer constructs — libraries (§5.16), uses (§5.17), and the finer §5.1 sub-rules — are cross-referenced here to the chapters that touch them, even where a construct is only mentioned rather than given its own chapter. If a reader finds a section they cannot locate, that is a bug in this book — please report it through the errata channel listed in the front matter.

---

This concludes the *POO²L Language Reference*. Continued discussion happens in the body of the spec (`POO²L/standard.md`) and in the implementation literature of specific POO²L processors. For questions about the specification itself, see the contact addresses in the spec's preamble.

# Lexique

**Alias** — Alternative identifier for a prompt, enabling backward compatibility when a prompt's primary `id` changes. Declared as `@@@alias <old_id>`. Resolved transparently to the canonical `id` in Phase 1 of the processing model. (Spec §3.13, §5.14.) → Ch. XXV, Ch. IV

**Assertion** — A declarative, blocking constraint on RENDERED content — the text-level analogue of a parameter constraint. Declared as `@@@assert <predicate>` (a line directive). Predicates : `contains`, `not contains`, `matches` (ECMAScript regex), `minlen`, `maxlen` ; multiple assertions combine as an implicit AND (no boolean combinators). Validates the entire content of its nearest enclosing scope, evaluated in Phase 10 (after substitution/includes/inheritance, before output generation in Phase 11). Always blocking — an unsatisfied predicate raises ASSERTION_FAILED (a malformed predicate is INVALID_ASSERTION). Applies to the rendered prompt only, never to the LLM's response. (Spec §5.15.) → Ch. XIII, Ch. XXX

**Attachment** — File or binary content associated with a prompt, described by two orthogonal axes. WHEN it is bound : `fixed` (bound at definition) or `to-include` (provided at instantiation ; required by default, optional with the `?` prefix on `attachment`). SOURCE (only for `fixed` ; where the content lives) : `file` (filesystem path), `blob` (inline data URI), or `record` (external store ref). A `to-include` attachment carries no source. (Spec §3.7, §5.8.) → Ch. XVI, Ch. IV

**Block comment** — Multi-line comment delimited by `@@@comment ... @@@end-comment`. Removed at instantiation. May be nested. (Spec §5.11.2.) → Ch. VIII

**Collection** — OO² (horizontal-axis) entity whose *methods* are the templates that belong to it. Supports inheritance, member override (by canonical id), virtual members, and collection-axis `^pa-`

`rent.<template_id>` / `^self.<template_id>` resolution from templates rendered inside derived collections. Collections NEST by containment (a collection may contain sub-collections, to unbounded depth); qualified member ids carry one segment per containment level. There is no `kind` flag and no sidecar: an entity is a collection precisely because a `@@@collection <name> ... @@@end-collection` block declares it; a folder has no such block. Required for Level 5 conformance. (Spec §3.9, §4.6.1, §5.10.) → Ch. XXII, Ch. XXIII, Ch. XX

**Composition** — POO²L's three composition axes. *Vertical*: prompt inheritance hierarchies (OO #1). *Horizontal*: collection inheritance hierarchies (OO #2) plus parameter-set binding. *Orthogonal*: a derived prompt living inside a derived collection participates in both chains. (Spec §4.6.2.) → Ch. XXIV, Ch. XIX, Ch. XXIII

**Conformance level** — A claimed implementation profile. Six levels: L0 (unstructured), L1 (sections), L2 (sections + includes), L3 (structured data and validation — parameter sets, parameters/constants/attachments, select options, constraints, deprecation, aliases, `@@@assert` / `@@@scope`), L4 (Object-Oriented — prompt inheritance, OO #1), L5 (full OO² — collection inheritance, virtual/final templates, member override, the second OO axis). L0 support is mandatory for all conforming implementations but is NOT a claimable badge (minimum claimable: Level 1). (Spec §8.1.) → Ch. XXXII, Ch. II

**Constant** — Named immutable value bound at definition time. Same property structure as a parameter (name, type, label) but its value is fixed and cannot be re-bound at instantiation. Declared as `@@@const <NAME> : <type> = <value>`. (Spec §3.6, §4.4.3, §5.7.) → Ch. XIV, Ch. IV

**Constraint** — Declarative validation rule attached to a parameter, defining bounds or patterns that values MUST satisfy. Available keys: `min`, `max` (number), `minlen`, `maxlen` (text, textarea), `pattern` (text, regex). Violation raises CONSTRAINT_VIOLATION. (Spec §3.11, §4.4.2 *Declarative constraints*.) → Ch. XII, Ch. XI

**Declaration order** — The recommended *writing* order for declarations within a prompt: Identity → Lifecycle (alias, deprecated) → Parameters → Constants → [Content] → Attachments. Processors MUST accept declarations in any order; the order has no semantic significance for processing. (Spec §4.4 *Standard declaration order*.) → Ch. VI

**Deprecated** — Lifecycle marker on a prompt signaling that it is scheduled for removal and should be replaced by an indicated alternative. Declared as `@@@deprecated "<migration message>"`. Purely informational — does NOT prevent instantiation; implementations SHOULD emit a warning. (Spec §3.12, §4.4.6, §5.13.) → Ch. XXVI

**Description** — Brief human-readable explanation of what a prompt is for. OPTIONAL: declared as `@@@description <short description>`; omitting it raises the advisory MISSING_DESCRIPTION warning (§6.2), never blocking. Spaces allowed; no length cap in the language. (Spec §5.2, §6.2.) → Ch. VI

**Final prompt** — A prompt that carries the `final` modifier on its `@@@template` header (`@@@template final <collection>.<id>`), declaring itself **sealed** as a collection member: no derived collection MAY override the prompt by id (OVERRIDE_OF_FINAL_TEMPLATE at PHASE 2). Symmetric to the template-level virtuality modifier `virtual`. Mutually exclusive with `virtual` on the same prompt (INVALID_SYNTAX). The header is stripped from the rendered output, like `@@@name` and `@@@description`. (Spec §4.6.1 *Final templates*.) → Ch. XXIII, Ch. XXX

**Final section** — A section declared with the `final` inheritance modifier on its `@@@section` line — `@@@section final <name>` — that is sealed against any descendant redeclaration. Any override attempt, even one that changes only the toggle or the description, raises OVERRIDE_OF_FINAL_SECTION at PHASE 3. Mutually

exclusive with `virtual` (INVALID_SYNTAX). May combine freely with the `?` / `!` presence toggle. (Spec §5.3.) → Ch. VII, Ch. XIX, Ch. XXX

**Folder** — Pure organizational grouping of templates, analogous to a file-system directory. Carries no compositional semantics (no inheritance, no virtual members, no `parent` resolution). Available at all levels (L0–L5). Addressed with `/` (a collection is addressed with `.`). There is no `kind` flag : an entity is a folder because it is a storage location with no `@@@collection` block declaring it ; declaring such a block promotes it to a collection. (Spec §4.7, §3.21.) → Ch. XXI

**id** — Unique identifier of a prompt within its scope. NO spaces ; **case- and diacritic-insensitive** (canonical form UPPERCASE with diacritics stripped — `code_review`, `Code_Review` and `café` / `cafe` collide ; a redeclaration is DUPLICATE_DECLARATION) ; a Unicode letter then letters, digits, and underscore only (NO hyphen, which is reserved for the language's own multi-word keywords). It is NOT a separate `@@@id` directive : the `id` rides on the `@@@template <id>` header that opens the entity. Used for includes, inheritance, and collection references ; the header MAY be qualified (`<collection>.<id>`, deeper when collections nest) to declare collection membership. (Spec §5.2, Appendix B.) → Ch. VI, Ch. XVII

**Identity** — A prompt's stable `id`, human-readable `name`, and optional `description`. The `id` rides on the `@@@template <id>` header (MUST be the entity's first line) and the `@@@name` is mandatory ; `@@@description` is OPTIONAL — omitting it raises the advisory MISSING_DESCRIPTION warning (§6.2), never blocking. Identity ensures templates are not anonymous when exported and is what later constructs (includes, inheritance, collection membership) refer back to. (Spec §4.4.1, §5.2, §6.2.) → Ch. VI

**Include** — Reference mechanism that incorporates external prompt content by reference. Declared as `@@@{include <reference_path>}`, with optional parameter overrides via `(name=va-`

`lue, ...)`. Supports self/parent references and section selectors (`<prompt>.<section>`). (Spec §3.3, §4.3, §5.4.) → Ch. XVII, Ch. XVIII, Ch. XX

**Inheritance** — Mechanism whereby a child prompt derives structure and content from a parent prompt via the inline `inherits <parent>` clause on its `@@@template` header (`@@@template child inherits parent`) — not a separate directive. Single inheritance; cycles forbidden (CIRCULAR_INHERITANCE); an undefined parent raises UNDEFINED_REFERENCE. Resolved at PHASE 3 of the processing model. The child inherits all sections, may override them by name, and may add new sections. (Spec §3.4, §4.5, §5.5, §6.1.) → Ch. XIX

**Inline comment** — Short comment: line `@@@-- ...` (to end of line) or bounded `@@@(-- ... )`. Removed at instantiation. Non-nesting; raise the bracket-run count to admit a literal closer in the body. (Spec §5.11.1.) → Ch. VIII

**Maturity level** — POO²L's six progressive levels of template-engineering maturity: L0 (unstructured plain text), L1 (structured — sections), L2 (procedural — includes), L3 (structured data and validation — typed parameters, constraints, parameter sets, attachments, assertions, lifecycle metadata), L4 (Object-Oriented — prompt inheritance, OO #1), L5 (OO² — collections, virtual/final templates, the second OO axis). (Spec §4.) → Ch. II, Ch. XXXII

**name** — Human-readable handle for a prompt, declared with `@@@name`. Spaces and special characters allowed. Mandatory for a structured (L1+) prompt, alongside the `@@@template <id>` header. The `name` is for people; the `id` is for machines. (Spec §5.2.) → Ch. VI

**OO² / Object-Oriented Squared** — Two parallel Object-Oriented mechanisms operating on different membership relations. OO #1 (templates): a prompt's methods are its sections; supports `inherits`, virtual sections, and the template-axis references `parent.<section>` / `self.<section>` (no `^`). OO #2 (collections): a

collection's methods are its templates ; supports collection inheritance, virtual templates, member override, and the collection-axis references `^parent.<template_id>` / `^self.<template_id>` (with the unary `^` operator) from a prompt rendered inside a derived collection. TWO DISJOINT AXES — the `^` names the axis, no precedence and no fallback. The two axes compose orthogonally. (Spec §3.10, §4.6.) → Ch. XXIII, Ch. XXIV, Ch. II

**Override** — Replacement of an inherited element by a same-named declaration in a descendant. *Section override* (OO #1) : a child prompt re-declares a section with the parent's name ; the child's content replaces the parent's for that section. *Member override* (OO #2) : a derived collection declares a prompt with the same `id` as one inherited from the parent collection ; the derived prompt wins. → Ch. XIX, Ch. XXIII, Ch. XXIV

**Parameter** — Named variable within a prompt that can be bound to a value at instantiation time. Required by default ; optional with the `?` suffix on the keyword (`@@@param? <name> ...`). Standard types : text, textarea, number, date, boolean, select. May carry declarative constraints. (Spec §3.5, §4.4.2, §5.6.) → Ch. XI, Ch. XII

**Parameter set** — Named collection of parameter bindings used to instantiate a generic prompt with specific values. Declared inline as `@@@parameter-set <name> ... @@@end-parameter-set`. Forbidden inside templates that belong to a collection (use collection-level parameter sets instead). (Spec §3.8, §4.4.5, §5.9.) → Ch. XV

**Processing model** — The deterministic eleven-phase pipeline POO²L processors execute : (1) Parsing, (2) Collection Context Application, (3) Template Inheritance Resolution, (4) Section Selection, (5) Parameter Binding, (6) Constant Resolution, (7) Attachment Binding, (8) Include Resolution, (9) Parameter Substitution (with constraint validation), (10) Assertion Evaluation, (11) Output Generation. Identity and other directives are stripped at PHASE 11. (Spec §6.1.) → Ch. III, Ch. XXX

**Reserved keyword** — A word POO²L reserves; it MUST NOT be used as a user identifier (parameter, constant, attachment, section, prompt id, or collection). §5.1 defines two classes. LINE-DIRECTIVE keywords (written `@@@keyword`): prompt, name, description, section, template-body, param, const, attachment, parameter-set, end-parameter-set, collection, end-collection, library, end-library, requires, deprecated, alias, uses, verbatim, end-verbatim, comment, end-comment, assert, scope, end-scope, include, plus the OPTIONAL terminators end-template and end-section. CONTEXTUAL keywords (reserved, but never a bare directive — meaningful only at a fixed position inside a host directive, and rejected as INVALID_SYNTAX if written alone): inherits, as, first, last, before, after, fixed, file, blob, record, virtual, final. Block closers follow the uniform `@@@end-<keyword>` pattern. (Spec §5.1.) → Ch. VI

**Resolution phase** — One of the eleven ordered steps of the processing model (see *Processing model*). Each phase has a single responsibility (parsing, collection context, prompt inheritance, section selection, parameter binding, constant resolution, attachment binding, includes, substitution, assertion evaluation, output). (Spec §6.1.) → Ch. XXX, Ch. III

**Scope** — The namespace within which an `id` (and its aliases) must be unique. Scope rules are enforced by implementations at save time. Two templates may share an `id` if they live in different scopes; the resolution rule for cross-scope references belongs to the referencing site. → Ch. VI, Ch. XVII

**Section** — Named, delimited component of a structured prompt, built on independent axes: an optional **inheritance modifier** (`virtual` / `final`), a **presence toggle** glued to the keyword (`?` / `!` / none), and a name that MAY be a dotted `<parent>.<child>` **sub-section** (one level deep). There is NO section numbering `<N>` — ordered items are sub-sections. A plain `@@@section <name>` is always rendered; `?` opens a user include/exclude dialog defaulting to OFF (opt-in); `!` opens the same dialog defaulting to ON (opt-out). Description is preceded by `:` and is mandatory for `?` and `!` (it is the dialog text). A container section

(one with sub-sections) carries no body of its own (else SECTION_WITH_SUBSECTIONS_HAS_BODY). Names are case- and diacritic-insensitive. Sections are OPTIONAL at the prompt level — a prompt may declare zero, one, or many. The unit of override under inheritance. (Spec §3.2, §4.2, §5.3.) → Ch. VII, Ch. XIX

**self** — Inheritance-aware include selector that returns the *most specialized* version of a section available in the current prompt's inheritance chain. Template-axis forms (no `^`): `self.<section>`, `self.<prompt>.<section>`. The collection-axis counterpart carries the `^` prefix: `^self.<template_id>` is the current collection's effective member. (Spec §5.4 *Inheritance-aware references*.) → Ch. XX

**Sidecar metadata** — Application-level storage that lives *outside* prompt text — `tags`, catalog `description`, permissions, version history, usage statistics, and similar non-compositional data. None of POO²L's compositional model is sidecar: an entity is a collection because a `@@@collection` block declares it; the collection's `inherits` parent rides on that header; a prompt declares its *membership* in-text by QUALIFYING its `@@@template` header id (`@@@template <collection>.<id>`); and `virtual` / `final` are modifiers on a member's `@@@template` header. Sidecar carries only application data, never the OO² composition. (Spec §9.3.) → Ch. XXII, Ch. XXI

**Sigil** — Every POO²L construct begins with the single sigil `@@@`: a line directive (`@@@keyword ...`), a substitution (`@@@=name` / `@@@{= name }`), an inline include (`@@@{include ...}`), or a comment (`@@@-- ...` line, `@@@(-- ... )` bounded). To output a literal `@@@`, double it: `@@@@`. Whitespace around a directive's keyword and arguments is flexible. (Spec §5.1.) → Ch. VII, Ch. VIII

**Super** — Inheritance-aware include selector that returns content from the *parent* (OO #1, TEMPLATE axis) or *parent collection* (OO #2, COLLECTION axis), skipping the current prompt's or current collection's override. Template axis (no `^`): `parent`, `parent.<section>`, `parent.<prompt>.<section>`. Collection axis (with the unary `^`): `^parent.<template_id>`, `^parent.<template_id>.<section>`. TWO DISJOINT AXES — the `^`

fixes the axis syntactically; there is NO precedence and NO fallback (a `parent.<name>` matching no ancestor section is UNDEFINED_REFERENCE, never a collection lookup). The header form `@@@template P inherits ^parent.P` extends OO #1 across the collection axis. (Spec §5.4, §4.6.1.) → Ch. XX, Ch. XXIII

**System variable** — Read-only runtime value substituted in PHASE 9 (Parameter Substitution) without parameter plumbing, via `@@@=NAME` (never a `{{NAME}}` form). The twelve names: `DATE`, `TIME`, `DATETIME` (ISO 8601, `YYYY-MM-DDTHH:MM:SSZ`), `YEAR`, `MONTH`, `DAY`, `WEEKDAY`, `USER` (display name), `USERNAME` (login — host-provided; the engine reserves eleven names natively, the host supplies the login), `LANG`, `GROUPS`, `GROUPS_ADMIN`. Declaring a parameter/constant with a reserved name raises SYSTEM_VARIABLE_OVERRIDE. (Spec §5.7.1.) → Ch. X

**Template** — Structured or unstructured text input provided to an LLM to elicit a desired response or behaviour. The unit of identity and composition in POO²L. (Spec §3.1.) → Ch. IV, Ch. VI

**to-include** — Attachment binding time where the file is provided at instantiation, not at definition. Required by default; optional with `?` prefix on the `attachment` keyword. Carries no source. Contrast with the `fixed` binding time (sources `file` / `blob` / `record`) where the content is bound at definition. (Spec §3.7, §5.8.) Replaces the legacy `ask` / `to_include` mode. → Ch. XVI

**Validation boundary** — A pure validation boundary delimited by `@@@scope … @@@end-scope` that bounds which rendered text an assertion governs. Transparent to output (its content renders normally — it transforms nothing, unlike verbatim/comment) and structurally inert (not a section, not addressable, not a target of include or inheritance). May carry an optional label — `@@@scope "<label>"` — for readable diagnostics. Sits within one section or encloses whole sections; partial overlap with a section raises MALFORMED_SCOPE. Distinct from the identifier-uniqueness sense of *Scope* above. (Spec §5.15.3.) → Ch. XIII

**Verbatim block** — Block delimited by `@@@verbatim ... @@@end-verbatim` whose content is preserved literally in the output : POO²L syntax inside is NOT interpreted. Used for meta-templates, documentation of POO²L itself, and escaping. May be nested. (Spec §5.12.) → Ch. IX

**Virtual prompt** — An abstract prompt member of a collection, carrying the `virtual` modifier on its `@@@template` header (`@@@template virtual <collection>.<id>`). Its parameter list constitutes the signature that descendant collections' concrete overrides must implement. Rendering a virtual without an override raises UNIMPLEMENTED_VIRTUAL_TEMPLATE at PHASE 2. Mutually exclusive with `final` on the same prompt (INVALID_SYNTAX). (Spec §4.6.1 *Virtual templates*.) → Ch. XXIII

**Virtual section** — A section declared with the `virtual` inheritance modifier on its `@@@section` line — `@@@section virtual <name>` — that carries no body (only whitespace and inline comments are tolerated ; non-empty content raises VIRTUAL_SECTION_HAS_BODY). The section declares an abstract slot the descendant MUST implement ; failure to do so raises UNIMPLEMENTED_VIRTUAL_SECTION at PHASE 3. Mutually exclusive with `final` (INVALID_SYNTAX). May combine freely with the `?` / `!` presence toggle. (Spec §5.3.) → Ch. VII, Ch. XIX, Ch. XXX

---

# Index thématique

## Concepts

Anti-patterns → Ch. VII, Ch. XI, Ch. XIX
Closed specification → Ch. XXXIII, Ch. I
Composition (vertical, horizontal, orthogonal) → Ch. XXIV, Ch. XXIII, Ch. XIX
Conformance → Ch. XXXII
Cross-axis composition → Ch. XXIII, Ch. XXIV
Declaration order (recommended writing order) → Ch. VI
Determinism → Ch. I, Ch. III, Ch. XXXIII
Effective prompt set → Ch. XXIII, Ch. XXX
Final prompt (collection marker) → Ch. XXIII
Final section → Ch. VII, Ch. XIX
Identity → Ch. VI
Identity vs membership → Ch. VI, Ch. XXII
Member override → Ch. XXIII
Optionality (` ?` suffix) → Ch. VII, Ch. XI, Ch. XVI
Reserved keywords → Ch. VI
Scope (`id` uniqueness) → Ch. VI, Ch. XVII
Section override → Ch. XIX, Ch. XXIV
Section presence toggle (` ?` / ` !`) → Ch. VII
Sidecar metadata → Ch. XXI, Ch. XXII
Versioning → Ch. XXVII, Ch. XXV, Ch. XXVI
Virtual prompt (collection marker) → Ch. XXIII
Virtual section → Ch. VII, Ch. XIX

## Constraints

max → Ch. XII
maxlen → Ch. XII
min → Ch. XII

minlen → Ch. XII
pattern → Ch. XII

## Divers

Folder vs collection → voir Folder
Glossary → voir Lexicon

## Error conditions

ALIAS_CONFLICT → Ch. XXXI, Ch. XXV
AMBIGUOUS_COLLECTION → Ch. XXXI, Ch. XXIII
AMBIGUOUS_ENTITY → Ch. XXXI, Ch. XVII, Ch. XXX
AMBIGUOUS_INCLUDE → Ch. XXXI, Ch. XVII
AMBIGUOUS_USES_TARGET → Ch. XXXI, Ch. XXVIII
ASSERTION_FAILED → Ch. XXXI, Ch. XIII
ATTACHMENT_NOT_FOUND → Ch. XXXI, Ch. XVI
ATTACHMENT_REQUIRED → Ch. XXXI, Ch. XVI
CIRCULAR_COLLECTION_INHERITANCE → Ch. XXXI, Ch. XXIII
CIRCULAR_INCLUDE → Ch. XXXI, Ch. XVII
CIRCULAR_INHERITANCE → Ch. XXXI, Ch. XIX
CIRCULAR_LIBRARY_DEPENDENCY → Ch. XXXI, Ch. XXVIII
CONSTRAINT_VIOLATION → Ch. XXXI, Ch. XII
DEPRECATED_TEMPLATE (warning) → Ch. XXXI, Ch. XXVI
DUPLICATE_DECLARATION → Ch. XXXI, Ch. VI
EXPORT_OF_UNKNOWN_MEMBER → Ch. XXXI, Ch. XXII
EXPORT_OF_UNKNOWN_SECTION → Ch. XXXI, Ch. VII
IMMUTABILITY_VIOLATION → Ch. XXXI, Ch. XIV
INCLUDE_OF_VIRTUAL_SECTION → Ch. XXXI, Ch. VII, Ch. XVII
INVALID_ASSERTION → Ch. XXXI, Ch. XIII
INVALID_CONSTRAINT → Ch. XXXI, Ch. XII
INVALID_SECTION_ANCHOR → Ch. XXXI, Ch. VII
INVALID_SECTION_PLACEMENT → Ch. XXXI, Ch. VII
INVALID_SYNTAX → Ch. XXXI, Ch. VI
INVALID_SYNTAX → Ch. XXXI, Ch. VII
MALFORMED_SCOPE → Ch. XXXI, Ch. XIII

MISSING_DESCRIPTION (warning) → Ch. XXXI, Ch. VI
NOT_EXPORTED → Ch. XXXI, Ch. VII, Ch. XXII
OVERRIDE_CONTRACT_VIOLATION → Ch. XXXI, Ch. VII, Ch. XII
OVERRIDE_OF_FINAL_SECTION → Ch. XXXI, Ch. VII, Ch. XIX
OVERRIDE_OF_FINAL_TEMPLATE → Ch. XXXI, Ch. XXIII
PARAMETER_NAME_IS_PREFIX (warning) → Ch. XXXI, Ch. XI
RESOURCE_LIMIT_EXCEEDED → Ch. XXXI, Ch. XXXIV
SECTION_SHAPE_MISMATCH → Ch. XXXI, Ch. VII
SECTION_WITH_SUBSECTIONS_HAS_BODY → Ch. XXXI, Ch. VII
SELECT_INVALID_OPTION → Ch. XXXI, Ch. XI
SYSTEM_VARIABLE_OVERRIDE → Ch. XXXI, Ch. X
TYPE_MISMATCH → Ch. XXXI, Ch. XI
UNDECLARED_ATTACHMENT_BINDING → Ch. XXXI, Ch. XVI
UNDECLARED_INCLUDE_PARAMETER → Ch. XXXI, Ch. XVIII
UNDECLARED_INSTANTIATION_PARAMETER → Ch. XXXI, Ch. XI
UNDECLARED_PARAMETER_SET_BINDING → Ch. XXXI, Ch. XV
UNDEFINED_PARAMETER → Ch. XXXI, Ch. XI
UNDEFINED_REFERENCE → Ch. XXXI, Ch. XIV
UNDEFINED_REFERENCE → Ch. XXXI, Ch. XVII, Ch. VI
UNDEFINED_USES_TARGET → Ch. XXXI, Ch. XXVIII
UNIMPLEMENTED_VIRTUAL_SECTION → Ch. XXXI, Ch. VII, Ch. XIX
UNIMPLEMENTED_VIRTUAL_TEMPLATE → Ch. XXXI, Ch. XXIII
UNKNOWN_COLLECTION → Ch. XXXI, Ch. XXIII
UNKNOWN_FRAME → Ch. XXXI, Ch. XXX
UNKNOWN_PARAMETER_SET → Ch. XXXI, Ch. XV
UNKNOWN_SECTION_TOGGLE → Ch. XXXI, Ch. VII
USES_ALIAS_CONFLICT → Ch. XXXI, Ch. XXV, Ch. XXVIII
VIRTUAL_SECTION_HAS_BODY → Ch. XXXI, Ch. VII
VIRTUAL_SIGNATURE_MISMATCH → Ch. XXXI, Ch. XXIII
VIRTUAL_TEMPLATE_HAS_BODY → Ch. XXXI, Ch. XXIII

## Keywords

alias → Ch. XXV, Ch. VI
assert → Ch. XIII
attachment → Ch. XVI
collection (keyword) → Ch. XXI, Ch. XXII, Ch. XXIII
comment (block) → Ch. VIII
comment (line `@@@-- …`, bounded `@@@(-- … )`) → Ch. VIII

const → Ch. XIV
deprecated → Ch. XXVI
description → Ch. VI
id → Ch. VI, Ch. XVII
include → Ch. XVII, Ch. XVIII, Ch. XX
inherits → Ch. XIX, Ch. XXIII
name → Ch. VI
param → Ch. XI, Ch. XII
parameter-set → Ch. XV
scope (`@@@scope` / `@@@end-scope`) → Ch. XIII
section → Ch. VII, Ch. XXIV
verbatim → Ch. IX
virtual → Ch. XXIII

## Maturity levels

L0 — Unstructured → Ch. II, Ch. XXXII
L1 — Structured → Ch. II, Ch. VII
L2 — Procedural → Ch. II, Ch. XVII
L3 — Structured Data and Validation → Ch. II, Ch. XI, Ch. XII, Ch. XV, Ch. XVI
L4 — Object-Oriented → Ch. II, Ch. XIX
L5 — OO² (Object-Oriented Squared) → Ch. II, Ch. XXIII

## Modes

blob (attachment source) → Ch. XVI
file (attachment source) → Ch. XVI
fixed (attachment binding time) → Ch. XVI
record (attachment source) → Ch. XVI
to-include (attachment binding time) → Ch. XVI

## Processing phases

Phase 1 — Parsing → Ch. III, Ch. XXX
Phase 10 — Assertion Evaluation → Ch. XXX, Ch. XIII
Phase 11 — Output Generation → Ch. XXX, Ch. III
Phase 2 — Collection Context Application → Ch. XXX, Ch. XXIII
Phase 3 — Template Inheritance Resolution → Ch. XXX, Ch. XIX
Phase 3.b — Call-Contract Validation → Ch. XXX, Ch. XII
Phase 4 — Section Selection → Ch. XXX, Ch. VII
Phase 5 — Parameter Binding → Ch. XXX, Ch. XI, Ch. XV
Phase 6 — Constant Resolution → Ch. XXX, Ch. XIV
Phase 7 — Attachment Binding → Ch. XXX, Ch. XVI
Phase 8 — Include Resolution → Ch. XXX, Ch. XVII
Phase 9 — Parameter Substitution → Ch. XXX, Ch. XI

## References

^parent. (collection axis, OO #2) → Ch. XX, Ch. XXIII
^self. (collection axis, OO #2) → Ch. XX, Ch. XXIII
parent (alone) → Ch. XX, Ch. XIX
parent..
→ Ch. XX
parent.
→ Ch. XX
self..
→ Ch. XX
self.
→ Ch. XX

## System variables

DATE → Ch. X
DATETIME → Ch. X
DAY → Ch. X
GROUPS → Ch. X
GROUPS_ADMIN → Ch. X
LANG → Ch. X
MONTH → Ch. X

TIME → Ch. X
USER → Ch. X
USERNAME → Ch. X
WEEKDAY → Ch. X
YEAR → Ch. X

## Types

boolean (type) → Ch. XI
date (type) → Ch. XI
number (type) → Ch. XI, Ch. XII
select (type) → Ch. XI
text (type) → Ch. XI, Ch. XII
textarea (type) → Ch. XI, Ch. XII

# Index alphabétique


## ^

^parent. (collection axis, OO #2) → Ch. XX, Ch. XXIII
^self. (collection axis, OO #2) → Ch. XX, Ch. XXIII

## A

alias → Ch. XXV, Ch. VI
ALIAS_CONFLICT → Ch. XXXI, Ch. XXV
AMBIGUOUS_COLLECTION → Ch. XXXI, Ch. XXIII
AMBIGUOUS_ENTITY → Ch. XXXI, Ch. XVII, Ch. XXX
AMBIGUOUS_INCLUDE → Ch. XXXI, Ch. XVII
AMBIGUOUS_USES_TARGET → Ch. XXXI, Ch. XXVIII
Anti-patterns → Ch. VII, Ch. XI, Ch. XIX
assert → Ch. XIII
ASSERTION_FAILED → Ch. XXXI, Ch. XIII
attachment → Ch. XVI
ATTACHMENT_NOT_FOUND → Ch. XXXI, Ch. XVI
ATTACHMENT_REQUIRED → Ch. XXXI, Ch. XVI

## B

blob (attachment source) → Ch. XVI
boolean (type) → Ch. XI

## C

CIRCULAR_COLLECTION_INHERITANCE → Ch. XXXI, Ch. XXIII
CIRCULAR_INCLUDE → Ch. XXXI, Ch. XVII
CIRCULAR_INHERITANCE → Ch. XXXI, Ch. XIX
CIRCULAR_LIBRARY_DEPENDENCY → Ch. XXXI, Ch. XXVIII
Closed specification → Ch. XXXIII, Ch. I
collection (keyword) → Ch. XXI, Ch. XXII, Ch. XXIII
comment (block) → Ch. VIII
comment (line `@@@-- …`, bounded `@@@(-- … )`) → Ch. VIII
Composition (vertical, horizontal, orthogonal) → Ch. XXIV, Ch. XXIII,

Ch. XIX
Conformance → Ch. XXXII
const → Ch. XIV
CONSTRAINT_VIOLATION → Ch. XXXI, Ch. XII
Cross-axis composition → Ch. XXIII, Ch. XXIV

## D

DATE → Ch. X
date (type) → Ch. XI
DATETIME → Ch. X
DAY → Ch. X
Declaration order (recommended writing order) → Ch. VI
deprecated → Ch. XXVI
DEPRECATED_TEMPLATE (warning) → Ch. XXXI, Ch. XXVI
description → Ch. VI
Determinism → Ch. I, Ch. III, Ch. XXXIII
DUPLICATE_DECLARATION → Ch. XXXI, Ch. VI

## E

Effective prompt set → Ch. XXIII, Ch. XXX
EXPORT_OF_UNKNOWN_MEMBER → Ch. XXXI, Ch. XXII
EXPORT_OF_UNKNOWN_SECTION → Ch. XXXI, Ch. VII

## F

file (attachment source) → Ch. XVI
Final prompt (collection marker) → Ch. XXIII
Final section → Ch. VII, Ch. XIX
fixed (attachment binding time) → Ch. XVI
Folder vs collection → voir Folder

## G

Glossary → voir Lexicon
GROUPS → Ch. X
GROUPS_ADMIN → Ch. X

## I

id → Ch. VI, Ch. XVII
Identity → Ch. VI
Identity vs membership → Ch. VI, Ch. XXII
IMMUTABILITY_VIOLATION → Ch. XXXI, Ch. XIV
include → Ch. XVII, Ch. XVIII, Ch. XX
INCLUDE_OF_VIRTUAL_SECTION → Ch. XXXI, Ch. VII, Ch. XVII
inherits → Ch. XIX, Ch. XXIII
INVALID_ASSERTION → Ch. XXXI, Ch. XIII
INVALID_CONSTRAINT → Ch. XXXI, Ch. XII
INVALID_SECTION_ANCHOR → Ch. XXXI, Ch. VII
INVALID_SECTION_PLACEMENT → Ch. XXXI, Ch. VII
INVALID_SYNTAX → Ch. XXXI, Ch. VI
INVALID_SYNTAX → Ch. XXXI, Ch. VII

## L

L0 — Unstructured → Ch. II, Ch. XXXII
L1 — Structured → Ch. II, Ch. VII
L2 — Procedural → Ch. II, Ch. XVII
L3 — Structured Data and Validation → Ch. II, Ch. XI, Ch. XII, Ch. XV, Ch. XVI
L4 — Object-Oriented → Ch. II, Ch. XIX
L5 — OO² (Object-Oriented Squared) → Ch. II, Ch. XXIII
LANG → Ch. X

## M

MALFORMED_SCOPE → Ch. XXXI, Ch. XIII
max → Ch. XII
maxlen → Ch. XII
Member override → Ch. XXIII
min → Ch. XII
minlen → Ch. XII
MISSING_DESCRIPTION (warning) → Ch. XXXI, Ch. VI
MONTH → Ch. X

## N

name → Ch. VI
NOT_EXPORTED → Ch. XXXI, Ch. VII, Ch. XXII
number (type) → Ch. XI, Ch. XII

## O

Optionality (` ?` suffix) → Ch. VII, Ch. XI, Ch. XVI
OVERRIDE_CONTRACT_VIOLATION → Ch. XXXI, Ch. VII, Ch. XII
OVERRIDE_OF_FINAL_SECTION → Ch. XXXI, Ch. VII, Ch. XIX
OVERRIDE_OF_FINAL_TEMPLATE → Ch. XXXI, Ch. XXIII

## P

param → Ch. XI, Ch. XII
parameter-set → Ch. XV
PARAMETER_NAME_IS_PREFIX (warning) → Ch. XXXI, Ch. XI
parent (alone) → Ch. XX, Ch. XIX
parent..
→ Ch. XX
parent.
→ Ch. XX
pattern → Ch. XII
Phase 1 — Parsing → Ch. III, Ch. XXX
Phase 10 — Assertion Evaluation → Ch. XXX, Ch. XIII
Phase 11 — Output Generation → Ch. XXX, Ch. III
Phase 2 — Collection Context Application → Ch. XXX, Ch. XXIII
Phase 3 — Template Inheritance Resolution → Ch. XXX, Ch. XIX
Phase 3.b — Call-Contract Validation → Ch. XXX, Ch. XII
Phase 4 — Section Selection → Ch. XXX, Ch. VII
Phase 5 — Parameter Binding → Ch. XXX, Ch. XI, Ch. XV
Phase 6 — Constant Resolution → Ch. XXX, Ch. XIV
Phase 7 — Attachment Binding → Ch. XXX, Ch. XVI
Phase 8 — Include Resolution → Ch. XXX, Ch. XVII
Phase 9 — Parameter Substitution → Ch. XXX, Ch. XI

## R

record (attachment source) → Ch. XVI
Reserved keywords → Ch. VI

RESOURCE_LIMIT_EXCEEDED → Ch. XXXI, Ch. XXXIV

## S

scope (`@@@scope` / `@@@end-scope`) → Ch. XIII
Scope (`id` uniqueness) → Ch. VI, Ch. XVII
section → Ch. VII, Ch. XXIV
Section override → Ch. XIX, Ch. XXIV
Section presence toggle (` ?` / ` !`) → Ch. VII
SECTION_SHAPE_MISMATCH → Ch. XXXI, Ch. VII
SECTION_WITH_SUBSECTIONS_HAS_BODY → Ch. XXXI, Ch. VII
select (type) → Ch. XI
SELECT_INVALID_OPTION → Ch. XXXI, Ch. XI
self..
→ Ch. XX
self.
→ Ch. XX
Sidecar metadata → Ch. XXI, Ch. XXII
SYSTEM_VARIABLE_OVERRIDE → Ch. XXXI, Ch. X

## T

text (type) → Ch. XI, Ch. XII
textarea (type) → Ch. XI, Ch. XII
TIME → Ch. X
to-include (attachment binding time) → Ch. XVI
TYPE_MISMATCH → Ch. XXXI, Ch. XI

## U

UNDECLARED_ATTACHMENT_BINDING → Ch. XXXI, Ch. XVI
UNDECLARED_INCLUDE_PARAMETER → Ch. XXXI, Ch. XVIII
UNDECLARED_INSTANTIATION_PARAMETER → Ch. XXXI, Ch. XI
UNDECLARED_PARAMETER_SET_BINDING → Ch. XXXI, Ch. XV
UNDEFINED_PARAMETER → Ch. XXXI, Ch. XI
UNDEFINED_REFERENCE → Ch. XXXI, Ch. XIV
UNDEFINED_REFERENCE → Ch. XXXI, Ch. XVII, Ch. VI
UNDEFINED_USES_TARGET → Ch. XXXI, Ch. XXVIII
UNIMPLEMENTED_VIRTUAL_SECTION → Ch. XXXI, Ch. VII, Ch. XIX
UNIMPLEMENTED_VIRTUAL_TEMPLATE → Ch. XXXI, Ch. XXIII
UNKNOWN_COLLECTION → Ch. XXXI, Ch. XXIII

UNKNOWN_FRAME → Ch. XXXI, Ch. XXX
UNKNOWN_PARAMETER_SET → Ch. XXXI, Ch. XV
UNKNOWN_SECTION_TOGGLE → Ch. XXXI, Ch. VII
USER → Ch. X
USERNAME → Ch. X
USES_ALIAS_CONFLICT → Ch. XXXI, Ch. XXV, Ch. XXVIII

## V

verbatim → Ch. IX
Versioning → Ch. XXVII, Ch. XXV, Ch. XXVI
virtual → Ch. XXIII
Virtual prompt (collection marker) → Ch. XXIII
Virtual section → Ch. VII, Ch. XIX
VIRTUAL_SECTION_HAS_BODY → Ch. XXXI, Ch. VII
VIRTUAL_SIGNATURE_MISMATCH → Ch. XXXI, Ch. XXIII
VIRTUAL_TEMPLATE_HAS_BODY → Ch. XXXI, Ch. XXIII

## W

WEEKDAY → Ch. X

## Y

YEAR → Ch. X

---