

Chapter 3

Design Concepts of the J% Programming Language

Revised edition (v2). This is a revision of Chapter 3 of the author’s dissertation (*Integrating Domain-Specific Languages with General-Purpose Languages*, AUEB, 2015), prepared in 2026 alongside the j-mod reimplementation of the J% compiler. The revision corrects errors in the 2015 text, replaces informal claims with a stated module contract and a safety theorem, makes the language rules precise enough to test against, and adds an analysis of environmental oracles that the original walked past. All changes are itemized in §3.8. Notes marked [**Retrospective**] reference systems built after 2015 and would be omitted from a version reconstructing the historical chapter.

*Two languages implementing the same idea must, on pain of death,
use different terms.*

— Feldman’s Law of Programming Terminology

In the previous chapter, DSL embedding methods and state-of-the-art research were analysed and categorised according to classifications from the literature (Spinellis 2001; Mernik et al. 2005; Renggli et al. 2009). All those approaches address the same problem:

How can Domain-Specific Languages be embedded in General-purpose Programming Languages?

The 2015 text qualified this question with “efficiently” and conceded that efficiency is difficult to define — and then never defined it. This revision decomposes the question into three sub-questions that the rest of the thesis can answer with evidence:

- Q1 (Validation).** Can DSL programs embedded in a GPL be validated at compile time, and *which classes of faults* move from run time to compile time as a result?
- Q2 (Boundary safety).** Can the data flow between the GPL and each DSL be made type-safe, and *relative to what assumptions* does that safety hold?
- Q3 (Extensibility).** Can support for a new DSL be added *without modifying the host compiler*, and at what cost to the programmer who merely uses it?

“Efficiency” is thereby operationalized in three measurable dimensions, evaluated in Chapter 5 and Appendix B: **E1** — embedding effort (lines of code and number of new concepts versus a plain-library baseline); **E2** — run-time overhead versus hand-written code (the FIRE/J benchmark, Appendix B); **E3** — error-class shift (the inventory of fault classes detected at compile time that the baseline detects at run time or not at all).

3.1 Design Strategy

This section sets the design goals of a generic system that embeds DSLs in GPLs, as functional and non-functional requirements, and then isolates the key elements a solution must provide. Unlike the 2015 text, each requirement is stated so that Chapter 5 can check it, and §3.1.3 traces each requirement to the design element that realizes it and the place where the guarantee is established.

3.1.1 Requirements

The functional requirements derive from the case studies and metaprogramming frameworks surveyed in Chapter 2.

- F1 — DSL syntax validation.** Every embedded DSL program is checked at compile time. Most systems surveyed achieve this for one hard-wired DSL; the requirement here is that validation is provided *uniformly, for every embedded DSL*. What “checked” means is necessarily relative to a *validation oracle* — the grammar or engine that defines well-formedness — and §3.5 shows this relativity is not pedantry but a design dimension.
- F2 — Boundary type safety.** Values exchanged between the GPL and a DSL are governed by a declared type mapping, checked at compile time; incompatibilities are compile-time errors. (The 2015 text also promised that “the actual values are proper,” a value-level property no static check can promise in general; the claim is withdrawn and replaced by the precise guarantee of Theorem 3.1.)
- F3 — Extensibility.** Support for a new DSL is added as a *module* — a compiler plug-in — without modifying the host compiler or the host grammar. The 2015 text criticized prior systems as “tightly coupled with one specific host language”; that criticism applies verbatim to J%, which is coupled to Java. The claim is therefore sharpened: J%’s differentiator is not host-independence but **module-pluggability at low buy-in cost** — one keyword and one grammar production buy embedding for an open set of DSLs, where comparable systems demand a new toolchain (Metaborg) or a new front-end per DSL.
- F4 — Diagnostic attribution** (new in v2). Every error is reported against coordinates in the source the programmer wrote, and every run-time failure at a language boundary identifies its responsible party (module, environment, or user code). A validation architecture that produces errors pointing into generated code the user has never seen fails its usability purpose; the 2015 requirements list omitted this entirely, and §3.4 shows it is where the current design is weakest.

The non-functional requirements:

- N1 — Semantic parsimony.** The host language extension is minimal and follows the Principle of Least Astonishment (Raymond 2003): a programmer who knows Java should predict J%’s behaviour from Java’s. Concretely: one new keyword, one new type declaration form, one reference token — and *no construct that looks like Java but behaves differently* (a rule the 2015 configuration-type design quietly violated; §3.2.6 repairs the claim).
- N2 — Domain-model maintenance.** Each DSL keeps its own concrete syntax; embedded programs are written as they would be in the DSL’s native tools, so domain knowledge and existing corpora carry over. One scoped exception is admitted openly (the 2015

text claimed none): the external-reference token `#[...]` is injected *into* DSL bodies. The design accepts this single, lexically distinctive extension per DSL as the price of Q2, rather than pretending the DSL is untouched.

- N3 — Re-usability.** Embedded DSL programs are nominal, importable types: written once, referenced anywhere, shareable as libraries. The design’s granularity — one external type per source file (§3.2.3) — serves this goal for substantial DSL programs and works against it for one-liners; the trade-off is acknowledged in §3.2.3 rather than left implicit.
- N4 — Domain-specific optimisation opportunity.** Because the compiler sees each DSL program whole at compile time, modules can perform domain-specific optimisations: precompiling patterns, rewriting queries, specialising generated code.
- N5 — Reproducibility** (new in v2). Compilation is a function of the source and of *declared* inputs. Where a module consults anything beyond the source — a schema file, a live database — that dependency is explicit, named, and versioned (§3.5). An accepted program must not become unbuildable because the world changed silently.

The 2015 text correlated the requirements with the ISO/IEC 9126 quality model in a figure; the figure asserted coverage without argument, and 9126 had already been superseded by ISO/IEC 25010 (2011). The figure is replaced by the traceability table below (Table 3.1), which carries actual information.

3.1.2 Key Elements for Efficient Language Mixing

From the research and systems described in Chapter 2, five semantic and technical elements are isolated that a GPL–DSL integration system must provide.

DSL language code blocks. The DSL program is isolated from host code in a delimited block, keeping the DSL’s native syntax without perturbing the host grammar. Per-DSL operators (Perl’s regex operators being the canonical example) integrate one language beautifully and generalise to none. Code blocks are the trade-off that scales: native DSL syntax inside, one host production outside. The delimiting rule itself must be *precise* — a block whose boundary is ambiguous makes every guarantee downstream vacuous, because the validator may check different text than the programmer wrote (§3.2.4).

Special types. User-declarable types mark the presence of DSL code and trigger module-specific validation and code generation at compile time — compile-time hooks carried by the type system rather than by new expression syntax. (Comparable mechanisms: Anorm’s typed SQL, Felix’s embedded domains; [**Retrospective**] F# type providers (Syme et al. 2012) are this element in its purest form.)

Configuration hooks. Compile-time parameters that direct the module — expressed in host-language syntax and resolved before code generation (§3.2.6).

Type mapping. A declared correspondence between DSL types and host types, so the compiler can check every crossing of the language boundary and reject incompatible ones (§3.3.2). Precedents: the JDBC and ODBC specifications, which map SQL types to Java and C types respectively.

Type references. Named, typed placeholders inside the DSL body through which values pass between the two language domains — the single point where each DSL’s syntax is extended, and the syntactic anchor for boundary type checking (§3.2.7).

These elements fit statically compiled languages; in dynamic languages both the special types and the compile-time mapping lose their anchor. This thesis realizes the five elements in J%, an extension of Java, following the *extensible compiler/interpreter* pattern of Mernik et al. (2005).

3.1.3 Requirements Traceability

Req.	Realized by	Guarantee established at	Status
F1 syntax validation	module validator (§3.4, C2)	compile time, relative to the module’s oracle (§3.5)	achieved, oracle-relative
F2 boundary type safety	external references (§3.2.7) + type-mapping axioms (§3.3.2) + contract C3	compile time + final javac pass	achieved, conditional
F3 extensibility	module registry and contract (§3.4)	per module, host compiler fixed	achieved for the Java host
F4 diagnostics	source coordinates on parse errors and references; blame Corollary 3.2	partially	open: javac errors in generated code not yet mapped back
N1 parsimony	one keyword, one production, one reference token	by construction	achieved
N2 domain model	verbatim DSL bodies; single scoped injection #[...]	by construction	achieved, stated exception
N3 re-usability	externals are nominal, importable types	by construction	achieved; file granularity limits small fragments
N4 DSO opportunity	modules see whole DSL programs at compile time	per module	demonstrated (regex precompilation, SQL IN-list expansion)
N5 reproducibility	oracle discipline of §3.5; hermetic mode as default	per configuration	achieved in hermetic mode; live probe opt-in

Table 3.1: Requirements traceability: requirement, realizing design element, locus of the guarantee, and status.

3.2 The J% Language

J% adopts Java syntax with a minimal extension set. This section defines the terminology, the grammar, and — new in this revision — the precise lexical rules that the 2015 text left ambiguous. Where the rules are normative they are stated as such; where the current implementation (j-mod) restricts or extends the 2015 design, the divergence is noted explicitly.

3.2.1 Basic Terminology

External module. A compiler component adding support for one DSL. A module declares one or more *external base types*, one *configuration type*, a *type mapping* (§3.3.2), a

compile-time *validator*, and a *code generator* producing Java-compatible source (or JVM bytecode). Modules are the unit of extensibility (F3) and the party bound by the contract of §3.4.

External type. A user-defined reference type, marked by the keyword `external`, that embodies one DSL program. External types are final subtypes of their module’s external base type.

Configuration type. A Java class, passed as the type argument of the external base type, whose fields parameterize the module at compile time and, for a module-documented subset, at run time.

External reference. The typed placeholder `#[name]<Type>` inside a DSL body: the bridge through which values cross between Java and the DSL, and the only extension J% makes to any DSL’s syntax.

J% term	Key element (§3.1.2)
External type	DSL language code blocks
External base type	Special types
Configuration type	Configuration hooks
External reference	Type mapping, type references
External module	Extensibility carrier: packages special types, mapping, validator, generator

Table 3.2: J% terms related to the key concepts of §3.1.2. (The 2015 table mapped “External module” to “Special types,” a row error: the module is the unit that supplies all five elements for one DSL.)

3.2.2 Grammar

As an extension of Java, J% is described with the conventions of the Java Language Specification (Gosling et al. 2014); Appendix E contains the full grammar. The BNF conventions: $[x]$ — zero or one; $\{x\}$ — zero or more; $x \mid y$ — alternatives; literal characters in double quotes.

```

CompilationUnit:
    [ "package" QualifiedIdentifier ";" ]
    { ImportDeclaration }
    ExternalDeclaration

ImportDeclaration:
    "import" [ "static" ] QualifiedIdentifier [ "." "*" ] ";"

ExternalDeclaration:
    { Modifier } "external" Identifier "extends" QualifiedIdentifier
    [ "<" QualifiedIdentifier ">" ] ExternalBody

Modifier:
    "public" | "static" | "final"

ExternalBody:
    "{" BalancedText "}"

```

Differences from the 2015 grammar, all normative in v2: the configuration type argument is *optional* (historical `.jmod` sources omit it; the module’s default configuration then applies);

annotations are not permitted anywhere in a `.jmod` compilation unit; the modifier set is closed (§3.2.3); and `ExternalBody` is defined by the precise rule of §3.2.4 rather than the 2015 rule `"{" { AnyCharacter } "}"`, which was formally ambiguous — if *any* character may appear, the production does not determine where the body ends.

3.2.3 External Types and Their Modifiers

An external type is a user-defined reference type embodying one DSL program. It instructs the compiler that DSL code is present and selects, through its base type, the module that will validate and translate it.

Declaration rules (consolidating the 2015 §3.2.3 and §3.2.6, which contradicted each other — the former allowed only `public`, the latter also `static`):

- **One external type per compilation unit.** Rationale: it mirrors Java’s one-public-class-per-file convention, makes classification trivial (a source file is external if and only if it declares `external`), and gives each DSL program a stable, importable name. Cost, stated honestly: small DSL fragments pay a whole file each, and external types cannot be composed or inherited, so a regex cannot be built from named parts. This is accepted as a deliberate scope restriction; finer granularity (nested or expression-level externals) is future work, not a claimed feature.
- **Modifiers.** Exactly three are accepted: `public` (otherwise package access), `final` (permitted, redundant — every external type is implicitly final), and `static` (accepted for compatibility with historical sources; it has no effect on a top-level type). The modifiers `private`, `protected`, `abstract`, `native`, `synchronized`, `transient`, `volatile`, and `strictfp` are compile-time errors, as are annotations.
- External types are implicitly `final` and cannot be extended. They are subtypes of their module’s external base type (`org.jmod.dsl.module.ExternalBaseType`), instantiate with `new` like any reference type, and may carry Javadoc comments.

```
public external NumberRegex extends Regex<RegexConfiguration> {
    [0-9]+
}
```

```
NumberRegex nr = new NumberRegex();
```

Module-determined interface — an acknowledged POLA cost. The methods of an external type beyond those of its base type are produced by the module’s code generator; they cannot be enumerated by reading the `.jmod` source alone. The design accepts this (it is the same property F# type providers have) but no longer leaves it unstated: each module **MUST** document the generated API of its externals, **SHOULD** emit Javadoc on generated members, and the absence of IDE support for browsing that API is recorded as an open tooling problem under F4.

3.2.4 The External Body and Its Delimiting Rule

The external body carries the DSL program verbatim. The 2015 production `"{" { AnyCharacter } "}"` did not define where the body ends; v2 states the actual rule, normatively:

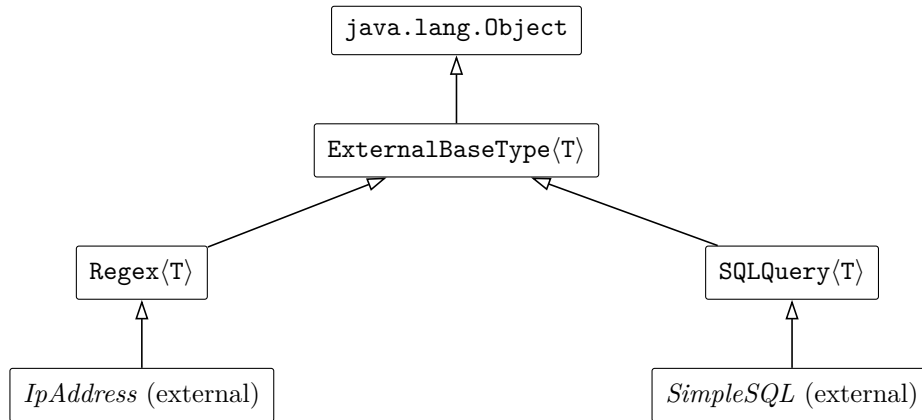


Figure 3.1: A typical external type hierarchy. Open arrowheads denote the subtype relation; user-declared external types are leaves.

Delimiting rule. After the opening `{`, the body extends to the matching `}` under *balanced-brace counting*: a scan tracks brace depth, incrementing at `{` and decrementing at `}`; the body is the longest prefix in which depth never reaches zero, and the first depth-zero `}` closes the declaration. The scan is purely lexical: it does not interpret string literals, comments, or escape sequences of the embedded DSL. End of input before the closing brace is a compile-time error (“unclosed external body”). Leading line breaks and trailing whitespace of the body are trimmed; interior text is preserved byte-for-byte.

Consequence, stated as a limitation. A DSL program whose text contains an *unbalanced* brace — a regex character class `[]`, a SQL string literal `'` — cannot be expressed in an external body. Balanced braces (as in `([0-9]{1,3}\.){3}[0-9]{1,3}`) are unproblematic. This is the classic delimiting problem of every language-box design; scannerless parsing (Metaborg) and typed literal blocks with declared delimiters (Omar et al. 2014) solve it at substantial machinery cost, and J% instead documents the restriction. An escape mechanism or per-module delimiter choice is future work; until then the restriction is part of the language definition, not an implementation accident — because the module contract (§3.4, K1) obliges the compiler to hand the module *exactly* the delimited text, the boundary must be decidable without consulting the module.

3.2.5 Packages and Imports

External types belong to a package like any Java type (the default package if none is declared) and use `import` statements, including static imports, to bring types into scope — in particular the module’s base type and the configuration type:

```
import org.jmod.dsl.regex.Regex;
import org.jmod.dsl.regex.RegexConfiguration;
```

3.2.6 Configuration Types

Configuration types parameterize a module’s compile-time behaviour and part of its run-time behaviour. Syntactically they are Java classes, used as the type argument of the external base type; every configuration type extends the module’s base configuration (ultimately `org.jmod.dsl.module.ExternalConfiguration`).

```

public class SimpleSQLConfiguration extends SQLConfiguration {
    public boolean SQLMOD_NS_AWARE = true;
    public String  SQLMOD_NS_URI   = "file:///examples/simplesql/schema.sql";
}

```

The 2015 text claimed that “since these types are classes, no special rules apply,” and that “standard Java semantics permit override in the initialisation of a field’s value.” Both claims were wrong and are withdrawn. Java has no field overriding — subclass fields *hide* superclass fields — and the compiler does not execute configuration classes; it reads them declaratively. The actual rules, now normative:

CT1 (Literal initializers). Compile-time configuration fields are public fields whose initializers are literals (string, boolean, character, or numeric). A computed initializer is not evaluated at compile time; it **MUST** be diagnosed as a compile-time error rather than silently ignored. (*Implementation status: j-mod currently extracts fields textually and ignores non-literal initializers; the diagnosis is specified here and pending.*)

CT2 (Overlay order). The effective configuration is a map built by overlaying, in order: the module’s defaults, then a configuration class found on the compiler classpath (read reflectively), then the user’s configuration source, walking its **extends** chain from base to derived so that derived declarations win. “Override” in J% configuration is this overlay, not a Java-semantics notion.

CT3 (Relative resource URIs). A `file:///...` value resolves relative to the directory of the configuration source that declares it.

CT4 (Compile/run-time split). Each module documents which keys are compile-time only and which are visible to generated code at run time. The split is part of the module’s public contract, not an emergent property.

The design trade-off, stated honestly. Under CT1–CT4, configuration classes are a *declarative subset of Java* — a small configuration language wearing Java’s syntax. The 2015 text rejected annotations as configuration hooks on the ground that annotations are “another DSL”; by that standard, so is this. The choice is therefore re-argued on its actual merits: configuration classes give typed fields, ordinary **extends**-based composition and sharing of configurations across externals, and plain-Java readability — at the cost of a restricted-evaluation rule the programmer must know (N1 is honoured by *diagnosing* the restriction, per CT1). Annotations would give idiomatic metadata syntax but no inheritance and awkward composition. Either choice is defensible; this one is now defended rather than asserted.

3.2.7 External References

Each DSL keeps its own syntax (N2). When a DSL program must exchange values with Java, the programmer declares an **external reference** — a contract between the two type systems and a placeholder that the module replaces during code generation.

```

ExternalRef:
    "#[" ParameterName "]" [ "<" FieldType ">" ]

ParameterName:
    Identifier                                     // [A-Za-z_][A-Za-z0-9_]*

FieldType:
    Identifier { "." Identifier } [ "[]" ]

```

External references are recognized *lexically* anywhere inside the external body. The rules, consolidated and made precise (the 2015 text stated none of R2–R5):

- R1 (Typing).** The `FieldType` names a Java type, written either fully qualified or by the conventional short names, which are canonicalized (`Integer` $\rightarrow$ `java.lang.Integer`, `String` $\rightarrow$ `java.lang.String`, and the JDBC-relevant families under `java.sql` and `java.time`). If the type argument is omitted, the reference defaults to `java.lang.String`.
- R2 (Repetition).** The same parameter name may appear more than once in a body. All occurrences must agree on the canonicalized type; conflicting declarations are a compile-time error reported at the offending occurrence’s source coordinates.
- R3 (Constructor derivation).** The declared parameters of the external type’s generated constructor are the body’s *unique* parameter names in order of first textual occurrence, each with its declared Java type. Arity is bounded by the JVM’s 255-parameter-slot limit on method descriptors.
- R4 (Array parameters).** A reference of array type `T[]` (other than `byte[]`, which remains a JDBC BLOB parameter) denotes a *collection* parameter; the SQL module expands it into an IN-list of placeholders at run time.
- R5 (Reserved token).** The token `#[` introduces a reference wherever it appears in a body; a DSL program that needs a literal `#[` cannot currently express it. This is the same class of limitation as the delimiting rule (§3.2.4) and is documented as such.

Fragility of positional construction, acknowledged. R3 makes the constructor signature a function of *textual order* in the DSL body: reordering the clauses of a query can silently permute the constructor’s parameters, and Java call sites are positional. Mitigations: the generated constructor **MUST** carry Javadoc naming each parameter and its source line; and modules **MAY** additionally emit a named builder/factory (`SimpleSQL.withCId(int)`) — specified here as a design extension, not yet implemented. The fragility is inherent to positional binding and is the strongest argument for the builder form becoming the default in a future revision.

Example — a SQL query taking an `int`:

```
public external SimpleSQL extends SQLQuery<SQLConfiguration> {
    select * from customer where customerId = #[c_id]<int>
}
```

generates the constructor `public SimpleSQL(int c_id)`. Further examples:

References in body	Generated constructor
<code>#[char_v]<char></code>	<code>TypeName(char char_v)</code>
<code>#[p1]<int>, #[p2]<java.lang.Integer></code>	<code>TypeName(int p1, Integer p2)</code>
<i>(no references)</i>	<code>TypeName()</code>
<code>#[sh]<short[]>, #[i]<int></code>	<code>TypeName(short[] sh, int i)</code>
<code>#[q]</code> <i>(type omitted)</i>	<code>TypeName(String q)</code>

Table 3.3: External references and the constructors they derive.

3.2.8 Example

A regular expression matching an IP address, in plain Java:

```
import java.util.regex.*;

public class IpAddress {
    private Pattern ipPattern;

    public IpAddress() {
        this.ipPattern = Pattern.compile("([0-9]{1,3}\\.){3}[0-9]{1,3}");
    }
}
```

Here the DSL travels as a `String`; the pattern’s validity is discovered at run time, as an exception. The same functionality in J%:

```
import org.jmod.dsl.regex.Regex;
import org.jmod.dsl.regex.RegexConfiguration;

public external IpAddress extends Regex<RegexConfiguration> {
    ([0-9]{1,3}\\.){3}[0-9]{1,3}
}
```

The pattern is validated when the program is compiled (F1), and the generated class exposes the module’s matching API. One convention is visible in the example: backslashes in a `Regex` or `SQL` body follow Java string-literal conventions (`\\.` for a literal dot), because those modules’ generators embed the body in a string literal without re-escaping backslashes — and the validator checks that literal’s *runtime value*, the exact pattern the generated class will compile (§3.4.2). The `JSON` module fully escapes its bodies, which are therefore raw text. Each module’s embedding convention is part of its public contract; for `Regex` and `SQL` it is a scoped, documented concession against N2, not a hidden one.

3.3 The J% Type System

This section states what J% adds to the Java type system and formalizes the type mapping mechanism. The 2015 edition prefaced this material with a survey of type theory (ramified types through pure type systems) and two pages of JVM internal type signatures; both are removed. The survey did no work in the design, and the signature tables described the obsolete pre-JNI native-method format — array descriptors carrying lengths (`[5I]`), malformed method descriptors (`(V)` for a no-argument void method, rather than `()V`) — which the modern JVM specification does not contain and, more to the point, which J% never uses: the mapping of §3.3.2 operates on source-level type names, not on descriptors.

What remains is the material the design actually stands on. Following Pierce (2002):

“A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.”

A language is *safe* when it protects its own abstractions: Java’s checked array access makes it statically checked and safe; C and C++ are statically checked but unsafe; Lisp, Scheme, and Perl are dynamically checked and safe (Pierce 2002 — whose quadrant of dynamically checked *unsafe* languages is essentially unpopulated; the 2015 table placed Bash and awk there without support from the cited source, and the claim is withdrawn). Static

checking buys early error detection, machine-checked documentation, and an optimisation substrate — the three properties J% extends across the language boundary: F1 moves DSL faults to compile time, the external reference types document the boundary, and N4 exploits compile-time knowledge of the DSL program.

One correction of record: the 2015 text illustrated “soft typing” with Java auto-boxing, translating `Integer i = 3;` to `new Integer(3)`. Auto-boxing is an implicit *coercion* in a fully static type system, compiled as `Integer.valueOf(3)` (JLS §5.1.7) — observably different from `new Integer(3)`, since `valueOf` caches small values — and is unrelated to soft typing (Cartwright & Fagan 1991), in which a checker inserts run-time checks into a *dynamically* typed program. The corrected relevance to J% is the coercion, not the soft typing: §3.3.2 uses exactly such standard Java coercions as the glue of its mapping mechanism.

3.3.1 The J% Extension to the Java Type Hierarchy

Java’s types divide into eight primitives and the reference types under `java.lang.Object` — classes, interfaces, enums, and arrays (Gosling et al. 2014). J% is statically typed exactly as Java is: every declaration is checked at compile time (the 2015 phrase “every variable’s type is inferred” overstated — Java of this design’s era infers only generic type arguments; types are *declared and checked*).

J% adds precisely one reference kind: the **external type**. Like an enum, it is a restricted class form with special compile-time treatment; unlike any Java type, its body is not Java. After compilation it behaves as a final class extending its module’s base type. This is the entire footprint of J% in the type system — the grammar adds one production, the hierarchy one kind (N1).

3.3.2 Type Mapping, Formalized

The type mapping mechanism defines which values may cross the boundary between Java and each DSL, in both directions: parameters flowing in through external references, and results flowing back through the generated API. Precedents include the JDBC specification’s SQL–Java correspondence and the JSON-library convention mapping JSON’s types onto `String`, `Number`, `Boolean`, `List`, and `Map`.

The 2015 text introduced *compatible types* and *dominant types* by example and stopped. The revision states the structure and its obligations. For a module M :

- Θ_M — the set of DSL types M exposes at the boundary;
- $C_M \subseteq T_{\text{Java}} \times \Theta_M$ — the *compatibility relation*: $(T, \tau) \in C_M$ means Java values of type T may be passed where the DSL expects τ , and results of DSL type τ may be received as T ;
- $\text{dom}_M : \Theta_M \rightarrow T_{\text{Java}}$ — the *dominant type* function, selecting for each DSL type the canonical Java type the compiler prefers when resolution is ambiguous;
- $\delta_M : \Theta_M \rightarrow \text{Values}$ — a *default value* for each mapped DSL type.

Four axioms make the mechanism well-defined; each closes a gap the 2015 text left open:

A1 (Dominance is compatible). $(\text{dom}_M(\tau), \tau) \in C_M$ for every $\tau \in \Theta_M$. The canonical choice is itself a legal choice.

A2 (Dominance is a function). Every mapped DSL type has exactly one dominant Java type. Ambiguity resolution is deterministic.

- A3 (Coherence by factoring).** For each $(T, \tau) \in C_M$ the module supplies a conversion $\text{conv}_{T, \tau}$. The obligation: every conversion factors through the dominant type — $\text{conv}_{T, \tau} = \text{conv}_{\text{dom}(\tau), \tau} \circ \text{coerce}_{T \rightarrow \text{dom}(\tau)}$, where *coerce* is a standard Java coercion (widening, boxing/unboxing). Consequently, if a value can reach τ by two compatible routes, the routes agree: coherence — the classical requirement on implicit conversion systems (Reynolds 1980) — holds *by construction* rather than by per-module luck.
- A4 (Totality of defaults).** δ_M is total on Θ_M . Defaults are what make whole-program compile-time evaluation of DSL bodies possible when a placeholder’s actual value is unknown — notably the SQL module’s probe, which substitutes δ -literals for references (§3.5.3).

In the implementation, the mapping surface is the `TypeMapping` interface, whose operations realize the structure directly: `isCompatible` realizes C_M , `dslToJava` realizes dom_M , and `javaToDSL` gives the DSL image of a Java type; type names are first normalized by the canonicalization of §3.2.7, R1. A worked example: a DSL exposing *integer* and *character* maps them to dominant types `int` and `char`; `java.lang.Integer` is additionally compatible with *integer*, its conversion factoring through the unboxing coercion to `int`. (The 2015 running example switched mid-stream from a *long* DSL type to a *character* one; corrected here.) If a program passes a Java type T where the body declares τ and $(T, \tau) \notin C_M$, the compiler reports the incompatibility at the reference’s source coordinates (F2, F4).

3.4 The Module Contract

The preceding sections describe mechanisms. This section — new in v2, and the theoretical center of the chapter — states what the mechanisms *guarantee*, and under which obligations. The need arises because J% is an **open** compiler: validation, type mapping, and code generation are delegated to modules that the core does not control. Any safety property of J% is therefore conditional on module behaviour, and a foundation must say exactly what is assumed of whom. The same move is standard wherever checking is pluggable: pluggable type systems state per-checker obligations (Bracha 2004; Papi et al. 2008), foreign-function interfaces state what the foreign side must preserve, and gradual typing states its guarantee as a blame theorem (Wadler & Findler 2009). J% admits — and, with the contract below, receives — a guarantee of the same shape.

Setting. A declaration `external X extends B<K> { body }` carries references $\#[p_1] \langle T_1 \rangle \dots \#[p_n] \langle T_n \rangle$ (unique names, first-occurrence order, per §3.2.7). The module M owning base type B supplies a validator, the mapping $(\Theta_M, C_M, \text{dom}_M, \delta_M)$ of §3.3.2, and a code generator. In the implementation these are fused in `Module.evaluate(unit, config)`; the contract separates them conceptually: `evaluate = validate`; `generate`.

3.4.1 Core Obligations

The J% core compiler — the trusted, module-independent part — must guarantee:

- K1 (Faithful extraction).** The text handed to M ’s validator and generator is exactly the delimited body of §3.2.4, unmodified. (This is why the delimiting rule must be decidable without the module: if extraction mis-slices, the module validates text the programmer did not write, and every downstream guarantee is about the wrong program.)
- K2 (Faithful references).** The reference list handed to M is exactly the occurrences in the body, with the source coordinates, canonicalized types, and ordering of §3.2.7, R1–R3.

K3 (Final arbiter). The complete generated source tree is compiled by `javac` as the pipeline’s last stage; nothing reaches bytecode unverified by the host language’s own type checker.

3.4.2 The Contract

Every module M must satisfy, for every accepted external:

C1 (Generation well-typedness). If the validator accepts $(body, K)$, the generated compilation unit compiles under `javac` against the module’s runtime, exposing exactly the constructor $X(T_1 p_1, \dots, T_n p_n)$ derived by R3.

C2 (Validation soundness, relative to a named oracle). If the validator accepts $body$, then the engine that will execute $body$ at run time accepts it too — *for the properties the validator claims to check, relative to the module’s declared oracle* (§3.5). The clause forces each module to name its oracle, and the built-in modules already span the interesting range: the **regex** module validates, with `java.util.regex.Pattern.compile`, the runtime value of the very literal its generator emits — the exact pattern the generated class will compile — so C2 holds by construction. (It did not always: an earlier build validated the raw body instead, and precisely this clause exposed the divergence — patterns containing backslashes were validated in one form and executed in another. The implementation was aligned to the contract in `jmodv3 0.2.0`.) The **SQL** module validates with `JSqlParser`, an *approximation* of any particular DBMS’s grammar, so C2 holds only up to the declared oracle, with false accepts and rejects possible in both directions; the **JSON** module validates against a schema whose draft version is the oracle’s version. Same mechanism, three different strengths of guarantee — a distinction the 2015 framework could not even express.

C3 (Mapping adequacy). The conversions of A3 are actually implemented by the generated code and honoured by the DSL runtime: every declared compatibility is *witnessed* at run time. (The JDBC precedent cuts both ways: the specification’s table is C_M ; a driver that mis-implements a getter violates C3 with no compile-time symptom — which is precisely why the witness obligation must be named.)

C4 (Hygiene and attributable failure). Generated code introduces no identifiers that capture or shadow user identifiers, leaves user-written compilation units untouched, and surfaces DSL-domain run-time failures as documented exception types of the module’s runtime.

3.4.3 Safety by Translation

Theorem 3.1 (Safety by translation). *Let P be a J% program accepted by a compiler whose core satisfies K1–K3, with every registered module satisfying C1–C4. Then: (i) the generated program $\llbracket P \rrbracket$ is a well-typed Java program and enjoys Java’s safety properties; (ii) every value crossing a $\text{Java} \leftrightarrow \text{DSL}$ boundary at run time is converted according to the responsible module’s declared mapping; (iii) every run-time failure originating in a DSL domain is attributable to a violation of C2–C4 by a module or to drift of an environmental oracle (§3.5) — never to the code the J% core generated.*

Proof sketch. (i): by C1 each generated unit is well-typed; by K3 `javac` verifies the whole tree, so acceptance of P implies `javac` accepted $\llbracket P \rrbracket$; Java’s safety then applies to $\llbracket P \rrbracket$. (ii): immediate from C3, with coherence of the conversion choice from A3. (iii): by (i) the glue cannot go wrong in Java’s sense; by C4 it neither perturbs user code nor fails undocumented;

hence a boundary failure implies a body the validator accepted and the engine rejected ($\neg C2$, including the oracle-drift case) or an unwitnessed conversion ($\neg C3$). $\square$

The theorem is deliberately cheap, and that is the design’s virtue made explicit: **because the final pipeline stage is the host compiler (K3), host-level type safety is inherited rather than re-proved.** What the contract adds is the cross-language part — the part javac cannot see.

Corollary 3.2 (Blame). *Under Theorem 3.1, extended with oracle pinning (§3.5.4), every run-time boundary failure carries blame: it identifies a module (a contract violation) or an oracle (drift), and never the user’s Java code. This is the analogue, for embedded DSLs, of “well-typed programs can’t be blamed” (Wadler & Findler 2009).*

3.4.4 The Trusted Computing Base

The contract clauses fail differently, and the difference locates the trust:

Violated	Detected	How it manifests
C1	at build time, by javac (fail-stop)	error in generated code — safety holds, diagnosis is misattributed (F4 gap)
C2	possibly never before run time	run-time rejection of a “validated” body
C3	possibly never before run time	wrong values silently crossing the boundary
C4	possibly never before run time	perturbed user code or unattributable failure
K1, K2	not detectable downstream	every guarantee holds about the wrong text

Table 3.4: Failure modes of the contract clauses.

The **trusted computing base** of a J% program is therefore: the core’s extraction (K1, K2), each module’s validator (C2), its marshallng (C3), and its hygiene (C4). Module *code generation* is untrusted-but-verified — javac re-checks it. The 2015 chapter never identified this boundary; identifying it is what a foundations section is for.

Contract as test obligations. Each clause induces a concrete test class, and the j-mod suite already instantiates part of the mapping: C1 is checked by the generated-runtime tests (compile an external, then load and execute the generated class); C2 invites differential tests of validator against engine; C3 invites round-trip conversion tests per compatible pair; K1/K2 are parser unit-test territory. The contract is thus not decoration: it is the module author’s checklist, and it was implicitly present in the test taxonomy all along.

3.5 Environmental Oracles and Snapshot Soundness

(*New in v2.*) Clause C2 made every validation guarantee relative to an **oracle** — the authority defining well-formedness for a DSL. This section examines what happens when the oracle is not a fixed mathematical object but something in the world, and turns what the 2015 text treated as two SQL-module configuration switches into a design dimension with named, analysable points.

3.5.1 Oracles, Classified

Definition. A module’s validation oracle O_M is the triple (identity, version, state) of the authority its validator consults; the validation judgment is written $\Gamma \vdash_{M,O} \text{body ok}$. Three

classes occur in J%:

Intrinsic oracles. The engine ships inside the compiler/runtime. The regex module’s oracle is the running JDK’s regex dialect. Note that even this oracle has a *version*: a pattern accepted when compiling on one JDK may target an older runtime whose dialect differs. The relativity never fully disappears; it is merely smallest here.

Document oracles. A file designated by configuration: the SQL module’s DDL schema (SQLMOD_NS_URI), the JSON module’s schema with its declared draft (drafts 4 through 2020-12). The oracle is versioned by the document’s content; compilation is a pure function of source plus document (N5 holds).

Stateful oracles. A live system: the SQL module’s JDBC probe against a running database. The oracle’s state is mutable *by parties outside the compilation*.

3.5.2 The Indexed Judgment and Its Consequences

With a stateful oracle, the judgment acquires an index: not “ P is well-typed” but “ P is well-typed **against database D at time t** ” — $\Gamma = \Gamma_{\text{source}} \cup \text{schema}(D, t)$. Two consequences follow.

Reproducibility fails (N5). The same source and compiler yield different verdicts on different days. A hermetic build system correctly classifies such a compile as an impure action; CI requires a database fixture; build caches must key on schema state they cannot see.

Soundness degrades to a snapshot guarantee. The honest statement is:

Snapshot soundness. If $\Gamma \vdash_{M, O_t} \text{body}$ ok, and at run time the oracle $O_{t'}$ agrees with O_t on the *footprint* of *body* — the fragment of the oracle’s state the body actually references — then executing *body* raises no well-formedness errors with respect to the checked properties.

For SQL the footprint is computable: JSqlParser yields exactly the tables and columns a query mentions. Between t and t' lies a time-of-check/time-of-use gap: `ALTER TABLE customer DROP COLUMN customerId` after compilation invalidates the guarantee without touching the program. What the compiler issues is therefore not a proof but a **certificate conditional on an environmental assumption** — kin to a lockfile, or to proof-carrying code whose proof obligation includes the environment (Necula 1997). The principle, stated once and generally:

When a static checker consults the environment, its verdict must be reified as a datum — a snapshot of the consulted state — that travels with the artifact and is re-validated where the artifact is used. Checking against the world yields a lease, not a proof.

3.5.3 The SQL Module, Concretely

J% implements two oracle modes today. **Hermetic** (document oracle): the query is parsed and checked against the DDL schema file — identifiers, arity, and the type-mapping compatibility of every `#[...]` binding site. **Live probe** (stateful oracle, opt-in via `SQLMOD_LIVE_TEST`): the compiler substitutes each reference with its default literal $\delta_M(\tau)$ (axiom A4 earning its keep) and *executes* the resulting statement over JDBC. Two properties of the probe must be documented normatively rather than discovered: it is **effectful** — the statement genuinely runs, so the probe target must be a disposable development database, never production (a non-SELECT statement mutates it); and it is **environment-coupled** — its verdict carries the full weight of §3.5.2.

3.5.4 The Design Space

Mode	Oracle	Repro. (N5)	Drift detected	Notes
Hermetic (schema file)	document	yes	at run time, as SQL errors	drift = schema file vs. real DB; managed by migration discipline
Live probe	stateful	no	at compile time only	effectful; CI needs a fixture; verdict expires silently
Snapshot-and-pin (<i>specified, not yet implemented</i>)	stateful at compile, document thereafter	yes, given the committed snapshot	at initialization, with blame	compiler records the footprint digest; generated runtime re-validates it and fails fast with a <i>drift</i> error naming the changed relation

Table 3.5: The oracle design space for the SQL module.

Snapshot-and-pin is the synthesis: at compile time the module digests the query’s footprint (those tables and columns, their types); the digest is embedded in the generated class; at class initialization or first connection the runtime re-checks the digest against the live catalog and fails with an error that *names the drift* (“column customer.customerId no longer exists — compiled against schema of 2026-08-26”). This converts silent run-time SQL failures into attributable staleness errors, which is exactly what Corollary 3.2 requires to extend blame over environmental change. The recommended posture for J%: hermetic by default, the live probe as an opt-in *lint* outside the typing judgment, and pinning as the planned mechanism that lets the stateful oracle participate in the judgment honestly.

[**Retrospective**]. Industry replayed this analysis a decade later. Rust’s `sqlx` began at “live probe” — compile-time query checking against `DATABASE_URL` — and, once that broke CI and reproducible builds at scale, grew an “offline mode” that caches per-query metadata in the repository: snapshot-and-pin, reinvented under another name. Go’s `sqlc` and Java’s `jOOQ` generator sit at “hermetic”. That the ecosystem converged on the same three points, from the same pressures, is the strongest available evidence that the taxonomy carves the space correctly — and that the 2015 chapter, which had implemented two of the three modes without naming the dimension, was nearer to a general result than it knew.

3.6 The Compilation Process

The pipeline, aligned with the implementation and annotated with where each obligation of §3.4 attaches (Figure 3.2):

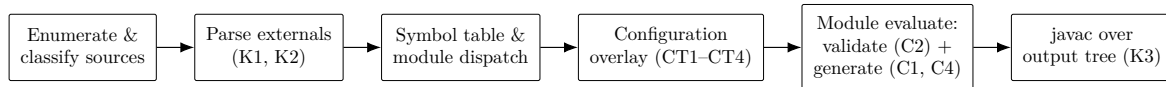


Figure 3.2: Pipeline of the compilation process, annotated with the obligations of §3.4 that attach to each stage.

1. **Enumerate and classify.** Input trees are walked; `.jmod` files are external compilation units, and plain `.java` files pass through to the output tree verbatim (C4’s “user units untouched” begins here).

2. **Parse.** Each external unit is parsed per §3.2; the body is extracted by the delimiting rule (K1) and its references by R1–R3 (K2), with source coordinates retained for F4.
3. **Analyse and dispatch.** Symbol tables are populated; each external’s base type is resolved through its imports to the owning module.
4. **Configure.** The effective configuration map is built by the overlay of CT2 and validated by the module’s declarative configuration validators before any DSL evaluation runs.
5. **Evaluate.** The module validates the body against its oracle (C2, §3.5) and checks every reference against the type mapping (F2, A1–A3); failures are reported at source coordinates. On success it generates the external’s Java source (C1) into the package-derived output path.
6. **Compile.** `javac` runs in-process over the entire output tree (K3) — the final arbiter from which Theorem 3.1(i) inherits Java’s safety.

The 2015 pipeline listed a separate pretty-printing stage; in the current design generated sources go to `javac` directly, and formatting of generated code is a module code-generation concern. The known F4 gap sits between stages 5 and 6: a C1 violation surfaces as a `javac` diagnostic pointing into generated code; mapping such diagnostics back to `.jmod` coordinates is specified as required and remains open.

3.7 Discussion

Chapter 2’s comparison table scored embedding approaches on syntax checking, typing, and complexity. J%’s position can now be stated more exactly than in 2015. Against extensible-syntax systems (Metaborg; SugarJ, Erdweg et al. 2011): J% trades their generality — grammar-level composition, host-independence — for a one-production extension whose entire safety story reduces to the host compiler plus a per-module contract; buy-in cost, not expressive power, is the differentiator (F3 as sharpened). Against typed literal approaches (Wyvern’s type-specific languages, Omar et al. 2014): Wyvern obtains formal composability guarantees inside the language’s own type theory, where J% obtains a translation-level guarantee (Theorem 3.1) conditional on module obligations; Wyvern’s is the stronger theorem, J%’s the cheaper machinery retrofit onto an industrial host. Against compile-time component generators ([**Retrospective**] F# type providers): the closest relatives in spirit — special types plus configuration hooks, with the same module-determined-API property and, for stateful sources, the same oracle problem §3.5 analyses.

Standing limitations, all stated above at their sites rather than discovered by the reader: the balanced-brace and `#[` lexical restrictions (§3.2.4, R5); file-level granularity and non-composability of externals (§3.2.3); positional constructor fragility (§3.2.7); configuration’s restricted-evaluation rule (CT1); the F4 diagnostic-mapping gap (§3.6); and the not-yet-implemented status of snapshot pinning (§3.5.4). None is fatal to the thesis question; each is a place where the mechanism is simpler than the problem, and says so.

3.8 Changes from the 2015 Edition

Corrections of fact.

- Removed the JVM internal type-signature figure and tables, which reproduced the obsolete pre-JNI format (array lengths in descriptors, (V) for ()V) and were unused by the design.

- Auto-boxing corrected to `Integer.valueOf` per JLS §5.1.7 and disentangled from soft typing.
- Pierce’s safety/checking classification restored to its source; the unsupported Bash/awk placement withdrawn.
- “Types are inferred” corrected to “declared and checked”.
- Resolved the §3.2.3/§3.2.6 contradiction on modifiers into one closed rule (`public` | `static` | `final`, with stated semantics).
- Terminology table row for External Module corrected; the mixed *integer/long/character* mapping example made consistent; the `#[sh]` constructor example fixed; typographical errors (*Dynamic*, *singed*, *Installability*) corrected.
- ISO/IEC 9126 correlation figure removed (superseded standard, decorative claim); replaced by the traceability table (§3.1.3).

Made precise.

- The research question decomposed (Q1–Q3) and “efficiency” operationalized (E1–E3) with pointers to Chapter 5 and Appendix B.
- The external-body delimiting rule stated (balanced-brace counting) with its limitation documented (§3.2.4).
- External references given full rules R1–R5: optional type with `String` default, canonicalization, repetition policy, first-occurrence constructor ordering, array parameters, reserved-token limitation.
- Configuration types given normative semantics CT1–CT4, replacing the incorrect “no special rules apply”; the annotations-versus-classes choice argued on merits.
- Type mapping formalized as $(\Theta, C, \text{dom}, \delta)$ with axioms A1–A4; coherence obtained by factoring through dominant types.

Added.

- Requirements F4 (diagnostic attribution) and N5 (reproducibility).
- §3.4: core obligations K1–K3, module contract C1–C4, Theorem 3.1 (safety by translation), Corollary 3.2 (blame), the TCB analysis, and the contract-to-test mapping.
- §3.5: oracle classification, the indexed judgment, snapshot soundness, the lease-not-proof principle, the SQL module’s modes analysed, snapshot-and-pin specified.
- §3.7: sharpened positioning against SugarJ, Wyvern TSLs, and type providers.

Honesty edits. The extensibility claim scoped to module-pluggability within a fixed host; the module-generated API acknowledged as a POLA cost; the `#[...]` injection acknowledged as a scoped exception to N2; file granularity and positional constructors acknowledged as trade-offs; the live probe documented as effectful.

Erratum (this printing). An earlier printing of this revision showed the §3.2.8 pattern with single backslashes and claimed the body is not read as a Java string literal; the embedding convention is the opposite for the Regex and SQL modules, and is now stated at the example. The C2 discussion (§3.4.2) was updated to name the validated text precisely — the divergence it originally glossed over was real, was caught by the contract itself, and was fixed in jmodv3 0.2.0.

References

Works cited in this chapter; entries marked [†] are new relative to the 2015 bibliography.

- Bracha, G. (2004). *Pluggable Type Systems*. OOPSLA Workshop on Revival of Dynamic Languages.[†]
- Cartwright, R., & Fagan, M. (1991). *Soft Typing*. PLDI.
- Erdweg, S., Rendel, T., Kästner, C., & Ostermann, K. (2011). *SugarJ: Library-based Syntactic Language Extensibility*. OOPSLA.[†]
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2014). *The Java Language Specification*, Java SE 8 Edition.
- Hudak, P. (1996). *Building Domain-Specific Embedded Languages*. ACM Computing Surveys 28(4es).
- ISO/IEC 25010:2011. *Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*.[†]
- Mernik, M., Heering, J., & Sloane, A. M. (2005). *When and How to Develop Domain-Specific Languages*. ACM Computing Surveys 37(4).
- Necula, G. C. (1997). *Proof-Carrying Code*. POPL.[†]
- Omar, C., Kurilova, D., Nistor, L., Chung, B., Potanin, A., & Aldrich, J. (2014). *Safely Composable Type-Specific Languages*. ECOOP.[†]
- Papi, M. M., Ali, M., Correa, T. L., Perkins, J. H., & Ernst, M. D. (2008). *Practical Pluggable Types for Java*. ISSTA.[†]
- Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.
- Raymond, E. S. (2003). *The Art of UNIX Programming*. Addison-Wesley.
- Renggli, L., Denker, M., & Nierstrasz, O. (2009). *Language Boxes: Bending the Host Language*. SLE.
- Reynolds, J. C. (1980). *Using Category Theory to Design Implicit Conversions and Generic Operators*. Semantics-Directed Compiler Generation, LNCS 94.[†]
- Spinellis, D. (2001). *Notable Design Patterns for Domain-Specific Languages*. Journal of Systems and Software 56(1).
- Syme, D., et al. (2012). *Strongly-Typed Language Support for Internet-Scale Information Sources*. Microsoft Research Technical Report MSR-TR-2012-101.[†]
- Wadler, P., & Findler, R. B. (2009). *Well-Typed Programs Can't Be Blamed*. ESOP.[†]

Software referenced in retrospective notes: `sqlx` (launchbadge, 2020–), `sqlc` (Kyle Conroy et al., 2019–), `jOOQ` (Data Geekery). Retrospective notes cite these as evidence of convergence, not as sources for the 2015 design.