Lexicon

This file defines several terms used by the Swift compiler and standard library source code, tests, and commit messages. See also the LLVM lexicon.

Glossary

abstraction pattern

The unsubstituted generic type of a property or function parameter, which sets constraints on its representation in memory. For example, given the following definitions:

```
struct Foo<T> {
  var value: T
  // Foo.value has abstraction pattern <T> T
}
struct Bar<T, U> {
  var value: (T) -> U
  // Bar.value has abstraction pattern <T, U> (T) -> U
}
struct Bas {
  var value: (Int) -> String
  // Bas.value has abstraction pattern (Int) -> String
}
let transform: (Int) -> String = { "\($0)" }
let foo = Foo<(Int) -> String>(value: transform)
let bar = Bar<Int, String>(value: transform)
let bas = Bas(value: transform)
```

although `foo.value`, `bar.value`, and `bas.value` all have the same function type `(Int) -> String`, they have different abstraction patterns. Because a value of type `Foo` or `Bar` may be used in a generic context and invoke `value` with a parameter or result type of unknown size, the compiler has to pick a more conservative representation for the closure that uses indirect argument passing, whereas `Bas.value` has a fully concrete closure type so can always use a more specialized direct register-based calling convention. The compiler transparently introduces reabstraction conversions when a value is used with a different abstraction pattern. (This is where the infamous "reabstraction thunk helpers" sometimes seen in Swift backtraces come from.)

access path

Broadly, an "access path" is a list of "accesses" which must be chained together to compute some output from an input. For instance, the generics system has a type called a `ConformanceAccessPath` which explains how to, for example, walk from `T: Collection` to `T: Sequence` to `T.Iterator: IteratorProtocol`. There

are several different kinds of "access path" in different parts of the compiler, but they all follow this basic theme.

In the specific context of imports, an "access path" is the `Bar` portion of a scoped import like `import class Foo.Bar`. Theoretically, it could have several identifiers to designate a nested type, although the compiler doesn't currently support this. It can also be empty, matching all top-level declarations in the module.

Note, however, that there has historically been some confusion about the meaning of "access path" with regards to imports. You might see some code use "access path" to include the `Foo` part or even to describe a chain of submodule names where a declaration is not valid at all. (Strictly, the chain of module names is a "module path" and the combination of module path + access path is an "import path".)

See `ImportPath` and the types nested inside it for more on this.

access pattern

Defines how some particular storage (a property or a subscript) is accessed. For example, when accessing a property `let y = a.x`, the compiler could potentially use `get` accessor or the `_read` accessor. Similarly, for a modification like `a.x += 1`, the compiler could use `get` + `set` or it could use `_modify`.

The access pattern can differ for call-sites which can/cannot see the underlying implementation. Clients which cannot see the underlying implementation are said to use the conservative access pattern.

archetype

A placeholder for a generic parameter or an associated type within a generic context. Sometimes known as a "rigid type variable" in formal CS literature. Directly stores its conforming protocols and nested archetypes, if any.

AST

"Abstract syntax tree", although in practice it's more of a directed graph. A parsed representation of code used by a compiler.

bitcode

Serialized LLVM IR.

build wrangler

Apple term for "the person assigned to watch CI this week".

canonical SIL

SIL after the mandatory passes have run. This can be used as input to IRGen to generate LLVM IR or object files.

canonical type

A unique representation of a type, with any sugar removed. These can usually be directly compared to test whether two types are the same; the exception is when generics get involved. In this case you'll need a generic environment. Contrast with sugared type.

Clang importer

The part of the compiler that reads C and Objective-C declarations and exposes them as Swift. Essentially contains a small instance of Clang running inside the Swift compiler, which is also used during IRGen.

conformance

A construct detailing how a particular type conforms to a particular protocol. Represented in the compiler by the ProtocolConformance type at the AST level. See also witness table.

contextual type

1. The expected type for a Swift sub-expression based on the rest of the statement. For example, in the statement `print(6 * 9)`, the contextual type of the expression `6 * 9` is `Any`.
2. The type of a value or declaration from inside a potentially generic context. This type may contain archetypes and cannot be used directly from outside the context. Compare with interface type.

critical edge

An edge in a control flow graph where the destination has multiple predecessors and the source has multiple successors.

currency type

A type that's meant to be commonly passed around and stored, like `Array`, as opposed to a type that's useful for temporary/internal purposes but which you wouldn't normally use in an external interface, like `ArraySlice`. Having broad agreement about the currency type you use for a particular kind of data (e.g. using `Array` to pass around sequential collections) generally makes the whole ecosystem better by reducing artificial barriers to passing data from one system to another, and it gives algorithm writers an obvious target to ensure

they optimize for. That's where the analogy to currency comes from: agreeing on a currency type improves the flow of information in a program in some of the same ways that agreeing on a currency improves the flow of trade in an economy.

customization point

Informal term for a protocol requirement that has a default implementation, i.e. one that conforming types don't *have* to implement but have the option to "customize".

dependency sink

Any request that uses a matching dependency source to write dependency edges into the referenced name trackers. For example, a request that performs direct lookup will write the name being looked up into the name tracker associated with the file that issued the lookup request. The request evaluator automatically determines the appropriate tracker for a dependency sink to write into based on the current active dependency source request.

dependency source

Any request that defines a scope under which reference dependencies may be registered. For example, a request to type check an entire file is a dependency source. Dependency sources are automatically managed by the request evaluator as request evaluation proceeds. Dependency sources provide one half of the necessary information to complete a full dependency edge. The other half is provided by corresponding dependency sink requests.

DI (definite initialization / definitive initialization)

The feature that no uninitialized variables, constants, or properties will be read by a program, or the analysis pass that operates on SIL to guarantee this. This was discussed on Apple's Swift blog.

DNM

"Do not merge". Placed in PR titles where discussion or analysis is still ongoing.

dup

From "duplicate". As a noun, refers to another filed issue that describes the same bug ("I have a dup of this"); as a verb, the act of marking a bug *as* a duplicate ("Please dup this to the underlying issue"). Sometimes written "dupe". Pronounced the same way as the first syllable of "duplicate", which for most American English speakers is "doop".

existential type

A type that is a protocol composition (including a single protocol and *zero* protocols; the latter is the `Any` type).

existential value

A value of existential type, commonly referred to simply as an "existential".

explicit module build

A module build where all dependency modules (including Clang modules) are passed to the compiler explicitly by an external build system, including any modules in caches. See also: implicit module build and fast dependency scanner.

fast dependency scanner

A Swift compiler mode that scans a Swift module for import declarations and resolves which modules will be loaded. It is based on the clang-scan-deps library within Clang, for (Objective-)C modules, but is extended to also understand textual Swift modules (.swiftinterface files).

The fast dependency scanner outputs a graph of compilation steps which can be used by a build system to schedule explicit module builds.

fragile

Describes a type or function where making changes will break binary compatibility. See LibraryEvolution.rst.

gardening

Describes contributions which fix code that is not executed (such as in a manifesto or README) and written text (correcting typos and grammatical errors).

generic environment

Provides context for interpreting a type that may have generic parameters in it. Generic parameter types are normally just represented as "first generic parameter in the outermost context" (or similar), so it's up to the generic environment to note that that type must be a Collection. (Another way of looking at it is that the generic environment connects interface types with contextual types).

generic signature

A representation of all generic parameters and their requirements. Like types, generic signatures can be canonicalized to be compared directly.

iff

"if and only if". This term comes from mathematics.

implicit module build

A module build where the compiler is free to transparently build dependent modules (including Clang modules), and access modules in different caches as necessary. For example, if a textual Swift module (.swiftinterface file) for a dependency does not have a corresponding binary Swift module (.swiftmodulea file), the compiler may transparently build a binary Swift module from the textual one as a cache for future compiler jobs, without involving any external build system that invoked the compiler. See also: explicit module build.

interface type

The type of a value or declaration outside its generic context. These types are written using "formal" generic types, which only have meaning when combined with a particular generic declaration's "generic signature". Unlike contextual types, interface types store conformances and requirements in the generic signature and not in the types themselves. They can be compared across declarations but cannot be used directly from within the context.

irrefutable pattern

A pattern that always matches. These patterns either bind to a variable or perform structural modification, e.x.:

1. `case _:.`
2. `case let x:.`
3. `case (_, _):.`

IR

1. "intermediate representation": a generic term for a format representing code in a way that is easy for a compiler or tool to manipulate.
2. "LLVM IR": a particular IR used by the LLVM libraries for optimization and generation of machine code.

IUO (implicitly unwrapped optional)

A type like Optional, but it implicitly converts to its wrapped type. If the value is `nil` during such a conversion, the program traps just as it would when a normal Optional is force-unwrapped. IUOs implicitly convert to and from normal Optionals with the same wrapped type.

IWYU (include what you use)

The accepted wisdom that implementation files (`.cpp`, `.c`, `.m`, `.mm`) should explicitly `#include` or `#import` the headers they use. Doing so prevents compilation errors when header files are included in a different order, or when header files are modified to use forward declarations instead of direct includes.

LGTM

"Looks good to me." Used in code review to indicate approval with no further comments.

LLVM IR

See IR.

lvalue

Pronounced "L-value". Refers to an expression that can be assigned to or passed `inout`. The term originally comes from C; the "L" refers to the "l"eft side of an assignment operator. See also rvalue.

main module

The module for the file or files currently being compiled.

mandatory passes / mandatory optimizations

Transformations over SIL that run immediately after SIL generation. Once all mandatory passes have run (and if no errors are found), the SIL is considered canonical.

metatype

The type of a value representing a type. Greg Parker has a good explanation of Objective-C's "metaclasses"; because Swift has types that are *not* classes, a more general term is used.

We also sometimes refer to a value representing a type as a "metatype object" or just "metatype", usually within low-level contexts like IRGen and LLDB. This is technically incorrect (it's just a "type object"), but the malapropism happened early in the project and has stuck around.

model

A type that conforms to a particular protocol. Sometimes "concrete model". Example: "Array and Set are both models of CollectionType".

module

Has *many* uses in the Swift world. We may want to rename some of them. #1 and #2 are the most common.

1. A unit of API distribution and grouping. The `import` declaration brings modules into scope. Represented as ModuleDecl in the compiler.
2. A compilation unit; that is, source files that are compiled together. These files may contain cross-references. Represented as "the main module" (a specific ModuleDecl).
3. (as "SIL module") A container for SIL to be compiled together, along with various context for the compilation.
4. (as "LLVM module") A collection of LLVM IR to be compiled together. Always created in an LLVMContext.
5. A file containing serialized AST and SIL information for a source file or entire compilation unit. Often "swiftmodule file", with "swiftmodule" pronounced as a single word.
6. (as "Clang module") A set of self-contained C-family header files. Represented by a ClangModuleUnit in the Swift compiler, each of which is contained in its own ModuleDecl. For more information, see Clang's documentation for Modules.
7. Shorthand for a "precompiled module file"; effectively "precompiled headers" for an entire Clang module. Never used directly by Swift. See also module cache.

module cache

Clang's cache directory for precompiled module files. As cache files, these are not forward-compatible, and so cannot be loaded by different versions of Clang (or programs using Clang, like the Swift compiler). Normally this is fine, but occasionally a development compiler will not have proper version information and may try to load older module files, resulting in crashes in `clang::ASTReader`.

NFC

"No functionality change." Written in commit messages that are intended to have no change on the compiler or library's behavior, though for some this refers to having the *same* implementation and for others merely an *equivalent* one. "NFC" is typically used to explain why a patch has no included testcase, since the Swift project requires testcases for all patches that change functionality.

open existential

An existential value with its dynamic type pulled out, so that the compiler can do something with it.

overlay

A wrapper library that is implicitly imported "on top of" another library. It contains an `@_exported` import of the underlying library, but it augments it with additional APIs which, for one reason or another, are not included in the underlying library directly.

There are two kinds of overlays:

A "clang overlay" (the older kind, so it's often just called an "overlay") is a Swift library that adds Swift-specific functionality to a C-family library or framework. Clang overlays are used with system libraries that cannot be modified to add Swift features. A clang overlay has the same module name as the underlying library and can do a few special things that normal modules can't, like adding required initializers to classes. If a module has a clang overlay, the Clang Importer will always load it unless it is actually compiling the overlay itself. Apple has a number of clang overlays for its own SDKs in `stdlib/public/Darwin/`.

A "separately-imported overlay" is a separate module with its own name. Unlike a clang overlay, it can be imported in some SourceFiles and not others. When the compiler processes import declarations, it determines which separately-imported overlays need to be imported and then adds them to the list of imports for that file; name lookup also knows to look through the overlay when it looks for declarations in the underlying module. Separately-imported overlays are used to implement the "cross-import overlays" feature, which is used to augment a module with additional functionality when it is imported alongside another module.

parent type

The type in which a given declaration is nested. For example:

```
struct Outer {
  struct Inner {
  }
}
```

`Outer` is the parent type of `Inner`.

Note that the terms "parent type" and "superclass" refer to completely different concepts.

PCH

Precompiled header, a type of file ending in .pch. A precompiled header is like a precompiled module, in the sense that it's the same file format and is just a cache file produced by clang and read by `clang::ASTReader`. The difference is that PCH files are not "modular": they do not correspond to a named module, and cannot be read in any order or imported by module-name; rather they must

be the first file parsed by the compiler. PCHs are used only to accelerate the process of reading C/C++/Objective-C headers, such as the bridging headers read in by the `-import-objc-header` command-line flag to swiftc.

PR

1. "Problem Report": An issue reported in LLVM's bug tracker. See also SR.
2. "pull request"

primary file

The file currently being compiled, as opposed to the other files that are only needed for context. See also Whole-Module Optimization.

QoI

"Quality of implementation." The term is meant to describe not how well-engineered a particular implementation is, but how much value it provides to users beyond a sort of minimum expectation. Good diagnostics are a matter of QoI, as is good unoptimized performance. For example, a comment like "FIXME: QoI could be improved here" is suggesting that there's some sort of non-mandatory work that could be done that would improve the behavior of the compiler–it is not just a general statement that the code needs to be improved.

It's possible that this term was originally "quality of life", written as "Qol", referring to the experience of end users. At some point along its history, the lowercase "L" was misinterpreted as an uppercase "i", and a new meaning derived. Swift inherited this term from LLVM, which got it from GCC.

Radar

Apple's bug-tracking system, or an issue reported on that system.

raw SIL

SIL just after being generated, not yet in a form that can be used for IR generation. See mandatory passes.

reabstraction

An implicit representation change that occurs when a value is used with a different abstraction pattern from its current representation.

realization

The process of initializing an ObjC class for use by the ObjC runtime. This consists of allocating runtime tracking data, fixing up method lists and attaching

categories.

This is distinct from the initialization performed by `+initialize`, which happens only when the first message (other than `+load`) is sent to the class.

The order of operations is: realization, followed by `+load` (if present), followed by `+initialize`. There are few cases where these can happen at different times.

- Common case (no `+load` or special attributes): Realization is lazy and happens when the first message is sent to a class. After that, `+initialize` is run.
- If the class has a `+load` method: `+load`, as the name suggests, runs at load time; it is the ObjC equivalent of a static initializer in C++. For such a class, realization eagerly happens at load time before `+load` runs. (Fun aside: C++ static initializers run after `+load`.) `+initialize` still runs lazily on the first message.
- If the class is marked `@_objc_non_lazy_realization`: Realization happens at load time. `+initialize` still runs lazily on the first message.

It's possible to create a class that is realized but not initialized by using a runtime function like `objc_getClass` before the class has been used.

See also: Mike Ash's blog post on Objective-C Class Loading and Initialization, which covers `+load` and `+initialize` in more detail.

refutable pattern

A pattern that may not always match. These include patterns such as:

1. Isa check, e.g. `case let x as String:`.
2. Enum case check: e.g. `case .none:`.
3. Expr pattern: e.g. `case foo():`.

resilient

Describes a type or function where making certain changes will not break binary compatibility. See LibraryEvolution.rst.

runtime

Code that implements a language's dynamic features that aren't just compiled down to plain instructions. For example, Swift's runtime library includes support for dynamic casting and for the Mirror-based reflection.

rvalue

Pronounced "R-value". Represents an expression that can be used as a value; in Swift this is nearly every expression, so we don't use the term very often. The

term originally comes from C; the "R" refers to the "r"ight side of an assignment operator. Contrast with lvalue.

script mode

The parsing mode that allows top-level imperative code in a source file.

Sema

Short for 'Semantic Analysis', the compiler pass that performs type checking, validation, and expression rewriting before SILGen.

SIL

"Swift Intermediate Language". A high-level IR used by the Swift compiler for flow-sensitive diagnostics, optimization, and LLVM IR generation.

SR

An issue reported on bugs.swift.org. A backronym for "Swift Report"; really the name is derived from LLVM's idiomatic use of "PR" ("Problem Report") for its bugs. We didn't go with "PR" for Swift because we wanted to be able to unambiguously reference LLVM bugs.

stdlib

"Standard library". Sometimes this just means the "Swift" module (also known as "swiftCore"); sometimes it means everything in the stdlib/ directory. Pronounced "stid-lib" or "ess-tee-dee-lib".

sugared type

A type that may have been written in a more convenient way, using special language syntax or a typealias. (For example, `Int?` is the sugared form of `Optional<Int>`.) Sugared types preserve information about the form and use of the type even though the behavior usually does not change (except for things like access control). Contrast with canonical type.

TBD

Text-based dynamic library files (TBDs) are a textual representation of the information in a dynamic library / shared library that is required by the static linker.

Apple's SDKs originally used Mach-O Dynamic Library Stubs. Mach-O Dynamic Library Stubs are dynamic library files, but with all the text and data stripped out.

thunk

In the Swift compiler, a synthesized function whose only purpose is to perform some kind of adjustment in order to call another function. For example, Objective-C and Swift have different calling conventions, so the Swift compiler generates a thunk for use in Objective-C that calls through to the real Swift implementation.

trap

A deterministic runtime failure. Can be used as both as a noun ("Using an out-of-bounds index on an Array results in a trap") and a verb ("Force-unwrapping a nil Optional will trap").

type metadata

The runtime representation of a type, and everything you can do with it. Like a `Class` in Objective-C, but for any type.

USR

A Unified Symbol Resolution (USR) is a string that identifies a particular entity (function, class, variable, etc.) within a program. USRs can be compared across translation units to determine, e.g., when references in one translation refer to an entity defined in another translation unit.

VWT (value witness table)

A runtime structure that describes how to do basic operations on an unknown value, like "assign", "copy", and "destroy". (For example, does copying this value require any retains?)

Only conceptually related to a witness table.

vtable (virtual dispatch table)

A map attached to a class of which implementation to use for each overridable method in the class. Unlike an Objective-C method table, vtable keys are just offsets, making lookup much simpler at the cost of dynamism and duplicated information about *non*-overridden methods.

WIP

"Work-in-progress". Placed in PR titles to indicate that the PR is not ready for review or merging.

witness

The value or type that satisfies a protocol requirement.

witness table

The SIL (and runtime) representation of a conformance; essentially a vtable but for a protocol instead of a class.

Only conceptually related to a value witness table.

WMO (whole-module optimization)

A compilation mode where all files in a module are compiled in a single process. In this mode there is no `primary file`; all files are parsed, type-checked, and optimized together at the SIL level. LLVM optimization and object file generation may happen all together or in separate threads.