Background

Swift's diagnostic style is inherited from Clang, which itself was loosely based on GCC's. These diagnostics aim to fit a terse but clear description of an issue into a single line, ideally suitable for display both on the command line and in IDEs. Additional information is provided via "notes", which are always attached to a primary diagnostic.

The Clang site has an older comparison between Clang's diagnostics and GCC's that shows where the attention to detail has been in that compiler.

This document describes several *current* guidelines for diagnostics in the Swift compiler and accompanying tools. It does not discuss potential future directions for diagnostics, such as following the examples of the Elm or Rust compilers to provide more information.

Errors vs. Warnings

Swift diagnostics are classified into errors, warnings, and notes. Notes are always considered "attached" to the immediately preceding error or warning, so the primary distinction is between errors and warnings. In the Swift compiler, a particular diagnostic should be a warning if the intent of the code is clear *and* it won't immediately result in a crash. This allows users to continue to build during refactoring without having to clean up all issues first.

For a warning, consider whether there might be a legitimate need to write the code in question, even if it's rare. In this case, there should be a way to silence the warning, and a note with a fix-it suggesting this. For example, the "unused function result" warning suggests assigning to `_` to make it clear that this is intentional.

Unlike Clang and many other compilers, Swift compiler warnings cannot be disabled as part of the invocation or through a standard construct in the source. This was a deliberate choice in order to keep the language from fracturing into dialects, but has been controversial throughout Swift's history.

Clang also has a kind of diagnostic called a "remark", which represents information about the build that does not indicate an issue. Swift does not currently have remarks, but there's no reason why it couldn't.

Grammar and Phrasing

- Swift diagnostics should be a single phrase or sentence, with no period at the end. If it's important to include a second idea in the diagnostic, use a semicolon to separate the two parts.
- Swift diagnostics are written in a terse abbreviated style similar to English newspaper headlines or recipes. Omit words that make the diagnostic longer without adding information, such as grammatical words like "the".

- *Do* include information that shows that the compiler understands the code. For example, referring to "instance method 'foo(bar:)' " is usually unnecessary, but may increase implicit trust in the compiler and decrease the developer's mental load.
  - *Don't* include information that might show that the compiler *doesn't* understand the code. For example, don't *assume* that this particular value is an instance method when it might be a property with a function type. However, *do* include details that might show how the compiler is *misunderstanding* the code.
- If there is a plausible fix or likely intended meaning, include that in the diagnostic whether or not there's a fix-it.
- When applicable, phrase diagnostics as rules rather than reporting that the compiler failed to do something. Example:
  - Normal: "cannot call 'super.init' outside of an initializer"
  - Better: " 'super.init' cannot be called outside of an initializer"
- When referring to attributes by name, use *either* "the 'foo' attribute" or " '@foo' ", rather than "the '@foo' attribute".
- Match the tone and phrasing of other diagnostics. Some common phrases:
  - "<noun> '<name>' " (e.g. "type 'Set' ")
  - "value of type <type>"
  - "did you mean...?"
  - "...to silence this warning"
  - "...here" (for a purely locational note)
- If possible, it is best to include the name of the type or function that has the error, e.g. "non-actor type 'Nope' cannot ... " is better than "non-actor type cannot ... ". It helps developers relate the error message to the specific type the error is about, even if the error would highlight the appropriate line / function in other ways.

Locations and Highlights

- Diagnostics are always emitted at a particular line and column. Try to be specific.
- Use highlights to build on the single location of the diagnostic, but use them sparingly. A diagnostic with *everything* on the line highlighted is no better than a diagnostic with nothing highlighted.

Fix-its

- Fix-its can span multiple lines, but may not be displayed in all contexts if they do. (In particular, the command-line display of a diagnostic currently only shows the line the diagnostic is on.)

- Fix-its must be in the same file as the diagnostic they are attached to. (However, a note can be in a different file than the primary error or warning.)
- If a fix-it is placed on an error or warning, it must be the single, obvious, and very likely correct way to fix the issue.
  - Ideally, the compiler or other tool will recover as if the user had applied the fix-it.
- Conversely, if a note has a fix-it, the note should describe the action the fix-it is taking and why.
- If a diagnostic has multiple notes with fix-its, the different notes should be treated as alternatives. In general, the first option should be the safest one. (It's also okay to have additional notes that do not provide fix-its, but don't drown the developer in notes.)
- If a warning or error has a fix-it, its notes should not have fix-its.
- Try to find something better than "add parentheses" to indicate that a warning should be silenced. Parentheses don't have anything to do with the actual problem, and if too many warnings do this the developer may end up silencing more than they meant to.
- Use Xcode-syntax placeholders in fix-its as necessary: `<#placeholder#>`. It's better to offer a fix-it with a placeholder in it than no fix-it at all.

The correct spelling of this feature is "fix-it" rather than "fixit". In camelcased contexts, use "FixIt" or "fixIt".

"Editor Mode"

The Swift compiler has a setting (under LangOptions) called `DiagnosticsEditorMode`. When set, diagnostics should be customized for an interactive editor that can display and apply complex fix-its, and worry less about the appearance in build logs and command-line environments.

Most diagnostics have no reason to change behavior under editor mode. An example of an exception is the "protocol requirements not satisfied diagnostic"; on the command line, it may be better to show all unsatisfied requirements, while in an IDE a single multi-line fix-it would be preferred.

Educational Notes

Educational notes are short-form documentation attached to a diagnostic which explain relevant language concepts. They are intended to further Swift's goal of progressive disclosure by providing a learning resource at the point of use when encountering an error message for the first time. In very limited circumstances, they also allow the main diagnostic message to use precise terminology (e.g. nominal types) which would otherwise be too unfriendly for beginners.

When outputting diagnostics on the command line, educational notes will be printed after the main diagnostic body if enabled using the `-print-educational-notes` driver option. When presented in an IDE, it's expected they will be collapsed under a disclosure arrow, info button, or similar to avoid cluttering output.

Educational notes should: - Explain a single language concept. This makes them easy to reuse across related diagnostics and helps keep them clear, concise, and easy to understand. - Be written in unabbreviated English. These are longer-form messages compared to the main diagnostic, so there's no need to omit needless words and punctuation. - Not generally exceed 3-4 paragraphs. Educational notes should be clear and easily digestible. Messages which are too long also have the potential to create UX issues on the command line. - Be accessible. Educational notes should be beginner friendly and avoid assuming unnecessary prior knowledge. The goal is not only to help users understand what a diagnostic is telling them, but also to turn errors and warnings into "teachable moments". - Include references to relevant chapters of *The Swift Programming Language*. - Be written in Markdown, but avoid excessive markup which negatively impacts the terminal UX.

Quick-Start Guide for Contributing New Educational Notes

Adding new educational notes is a great way to get familiar with the process of contributing to Swift, while also making a big impact!

To add a new educational note: 1. Follow the directions in the README to checkout the Swift sources locally. Being able to build the Swift compiler is recommended, but not required, when contributing a new note. 2. Identify a diagnostic to write an educational note for. To associate an educational note with a diagnostic name, you'll need to know its internal identifier. The easiest way to do this is to write a small program which triggers the diagnostic, and run it using the `-debug-diagnostic-names` compiler flag. This flag will cause the internal diagnostic identifier to be printed after the diagnostic message in square brackets. 3. Find any closely related diagnostics. Sometimes, what appears to be one diagnostic from a user's perspective may have multiple variations internally. After determining a diagnostic's internal identifier, run a search for it in the compiler source. You should find: - An entry in a `Diagnostics*.def` file describing the diagnostic. If there are any closely related diagnostics the note should also be attached to, they can usually be found nearby. - Each point in the compiler source where the diagnostic is emitted. This can be helpful in determining the exact circumstances which cause it to be emitted. 4. Add a new Markdown file in the `userdocs/diagnostics/` directory in the swift repository containing the contents of the note. When writing a note, keep the writing guidelines from the section above in mind. The existing notes in the directory are another useful guide. 5. Associate the note with the appropriate diagnostics in `EducationalNotes.def`. An entry like `EDUCATIONAL_NOTES(property_wrapper_failable_init,`

"property-wrapper-requirements.md") will associate the note with filename property-wrapper-requirements.md with the diagnostic having an internal identifier of property_wrapper_failable_init. 6. If possible, rebuild the compiler and try recompiling your test program with -print-educational-notes. Your new note should appear after the diagnostic in the terminal. 7. That's it! The new note is now ready to be submitted as a pull request on GitHub.

If you run into any issues or have questions while following the steps above, feel free to post a question on the Swift forums or open a work-in-progress pull request on GitHub.

Format Specifiers

(This section is specific to the Swift compiler's diagnostic engine.)

- %0, %1, etc - Formats the specified diagnostic argument based on its type.
- %select{a|b|c}0 - Chooses from a list of alternatives, separated by vertical bars, based on the value of the given argument. In this example, a value of 2 in diagnostic argument 0 would result in "c" being output. The argument to the %select may be an integer, enum, or StringRef. If it's a StringRef, the specifier acts as an emptiness check.
- %s0 - Produces an "s" if the given argument is anything other than 1, as meant for an English plural. This isn't particularly localizable without a more general %plural form, but most diagnostics try to avoid cases where a plural/singular distinction would be necessary in the first place.
- %error - Represents a branch in a %select that should never be taken. In debug builds of the compiler this produces an assertion failure.
- %% - Emits a literal percent sign.

Diagnostic Verifier

(This section is specific to the Swift compiler's diagnostic engine.)

If the -verify frontend flag is used, the Swift compiler will check emitted diagnostics against specially formatted comments in the source. This feature is used extensively throughout the test suite to ensure diagnostics are emitted with the correct message and source location.

-verify parses all ordinary source files passed as inputs to the compiler to look for expectation comments. If you'd like to check for diagnostics in additional files, like swiftinterfaces or even Objective-C headers, specify them with -verify-additional-file <filename>. By default, -verify considers any diagnostic at <unknown>:0 (that is, any diagnostic emitted with an invalid source location) to be unexpected; you can disable this by passing -verify-ignore-unknown.

An expected diagnostic is denoted by a comment which begins with `expected-error`, `expected-warning`, `expected-note`, or `expected-remark`. It is followed by:

- (Optional) Location information. By default, the comment will match any diagnostic emitted on the same line. However, it's possible to override this behavior and/or specify column information as well. `// expected-error@-1 ...` looks for an error on the previous line, `// expected-warning@+1:3 ...` looks for a warning on the next line at the third column, and `// expected-note@:7 ...` looks for a note on the same line at the seventh column.
- (Optional) A match count which specifies how many times the diagnostic is expected to appear. This may be a positive integer or `*`, which allows for zero or more matches. The match count must be surrounded by whitespace if present. For example, `// expected-error 2 ...` looks for two matching errors, and `// expected-warning * ...` looks for any number of matching warnings.
- (Required) The expected error message. The message should be enclosed in double curly braces and should not include the `error:`/`warning:`/`note:`/`remark:` prefix. For example, `// expected-error {{invalid redeclaration of 'y'}}` would match an error with that message on the same line. The expected message does not need to match the emitted message verbatim. As long as the expected message is a substring of the original message, they will match.
- (Optional) Expected fix-its. These are each enclosed in double curly braces and appear after the expected message. An expected fix-it consists of a column range followed by the text it's expected to be replaced with. For example, `let r : Int i = j // expected-error{{consecutive statements}} {{12-12=;}}` will match a fix-it attached to the consecutive statements error which inserts a semicolon at column 12, just after the 't' in 'Int'.
  - Insertions are represented by identical start and end locations: `{{3-3=@objc }}`. Deletions are represented by empty replacement text: `{{3-9=}}`.
  - Line offsets are also permitted; for instance, `{{-1:12-+1:42=}}` would specify a fix-it that deleted everything between column 12 on the previous line and column 42 on the next line. (If the sign is omitted, it specifies an absolute line number, not an offset.)
  - By default, the verifier ignores any fix-its that are *not* expected; the special `{{none}}` specifier tells it to verify that the diagnostic it's attached to has *only* the fix-its specified and no others.
  - If two (or more) expected fix-its are juxtaposed with nothing (or whitespace) between them, then both must be present for the verifier

to match. If two (or more) expected fix-its have `||` between them, then one of them must be present for the verifier to match. `||` binds more tightly than juxtaposition: `{{1-1=a}} {{2-2=b}} || {{2-2=c}} {{3-3=d}} {{none}}` will only match if there is either a set of three fix-its that insert `a`, `b`, and `d`, or a set of three fix-its that insert `a`, `c`, and `d`. (Without the `{{none}}`, it would also permit all four fix-its, but only because one of the four would be unmatched and ignored.)

- (Optional) Expected educational notes. These appear as a comma separated list after the expected message, enclosed in double curly braces and prefixed by 'educational-notes='. For example, `{{educational-notes=some-note,some-other-note}}` will verify the educational notes with filenames `some-note` and `some-other-note` appear. Do not include the file extension when specifying note names.