orphan:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)Modules.rst, line 3)**
>
> Unknown interpreted text role "term".
>
> ```
> .. default-role:: term
> ```

A *module* is the primary unit of code sharing in Swift. This document describes the experience of using modules in Swift: what they are and what they provide for the user.

> **Warning**
>
> This document was used in planning Swift 1.0; it has not been kept up to date.

High-Level Overview

A module contains declarations

The primary purpose of a module is to provide declarations of types, functions, and global variables that are present in a library. *Importing* <*import*> the module gives access to these declarations and allows them to be used in your code.

```
import Chess
import Foundation
```

You can also selectively import certain declarations from a module:

```
import func Chess.createGreedyPlayer
import class Foundation.NSRegularExpression
```

> **Comparison with Other Languages**
>
> Importing a module is much like importing a library in Ruby, Python, or Perl, importing a class in Java, or including a header file in a C-family language. However, unlike C, module files are not textually included and must be valid programs on their own, and may not be in a textual format at all. Unlike Java, declarations in a module are not visible at all until imported. And unlike the dynamic languages mentioned, importing a module cannot automatically cause any code to be run.

Imported declarations can be accessed with qualified or unqualified lookup

Once a module has been imported, its declarations are available for use within the current source file. These declarations can be referred to by name, or by *qualifying* <*qualified name*> them with the name of the module:

```
func playChess(_ blackPlayer : Chess.Player, whitePlayer : Chess.Player) {
  var board = Board() // refers to Chess.Board
}
```

Modules provide a unique context for declarations

A declaration in a module is unique; it is never the same as a declaration with the same name in another module (with one caveat

described below). This means that two types `Chess.Board` and `Xiangqi.Board` can exist in the same program, and each can be referred to as `Board` as long as the other is not visible. If more than one imported module declares the same name, the full *qualified name* can be used for disambiguation.

> **Note**
>
> This is accomplished by including the module name in the *mangled name* of a declaration. Therefore, it is an ABI-breaking change to change the name of a module containing a public declaration.

> **Warning**
>
> The one exception to this rule is declarations that must be compatible with Objective-C. Such declarations follow the usual Objective-C rules for name conflicts: all classes must have unique names, all protocols must have unique names, and all constructors, methods, and properties must have unique names within their class (including inherited methods and properties).

Modules may contain code

In addition to declarations, modules may contain implementations of the functions they define. The compiler may choose to use this information when optimizing a user's program, usually by inlining the module code into a caller. In some cases [1], the compiler may even use a module's function implementations to produce more effective diagnostics.

Modules can also contain *autolinking* information, which the compiler passes on to the linker. This can be used to specify which library implements the declarations in the module.

[1] Specifically, code marked with the `@_transparent` attribute is required to be "transparent" to the compiler: it *must* be inlined and will affect diagnostics.

Modules can "re-export" other modules

> **Warning**
>
> This feature is likely to be modified in the future.

Like any other body of code, a module may depend on other modules in its implementation. The module implementer may also choose to *re-export* these modules, meaning that anyone who imports the first module will also have access to the declarations in the re-exported modules.

```
@exported import AmericanCheckers
```

As an example, the "Cocoa" *framework* on OS X exists only to re-export three other frameworks: AppKit, Foundation, and CoreData.

Just as certain declarations can be selectively imported from a module, so too can they be selectively re-exported, using the same syntax:

```
@exported import class AmericanCheckers.Board
```

Modules are uniquely identified by their name

Module names exist in a global namespace and must be unique. Like type names, module names are conventionally capitalized.

> **TODO**
>
> While this matches the general convention for Clang, there are advantages to being able to rename a module for lookup purposes, even if the ABI name stays the same. It would also be nice to avoid having people stick prefixes on their module names the way they currently do for Objective-C classes.

> **Note**
>
> Because access into a module and access into a type look the same, it is bad style to declare a type with the same name as a top-level module used in your program:
>
> ```
> // Example 1:
> import Foundation
> import struct BuildingConstruction.Foundation
>
> var firstSupport = Foundation.SupportType() // from the struct or from the module?
>
>
> // Example 2:
> ```

```
import Foundation
import BuildingConstruction

Foundation.SupportType() // from the class or from the module?
```

In both cases, the type takes priority over the module, but this should still be avoided.

> **TODO**
>
> Can we enforce this in the compiler? It seems like there's no way around Example 2, and indeed Example 2 is probably doing the wrong thing.

`import`

As shown above, a module is imported using the `import` keyword, followed by the name of the module:

```
import AppKit
```

To import only a certain declaration from the module, you use the appropriate declaration keyword:

```
import class AppKit.NSWindow
import func AppKit.NSApplicationMain
import var AppKit.NSAppKitVersionNumber
import typealias AppKit.NSApplicationPresentationOptions
```

- `import typealias` has slightly special behavior: it will match any type other than a protocol, regardless of how the type is declared in the imported module.
- `import class`, `struct`, and `enum` will succeed even if the name given is a typealias for a type of the appropriate kind.
- `import func` will bring in all overloads of the named function.
- Using a keyword that doesn't match the named declaration is an error.

> **TODO**
>
> There is currently no way to selectively import extensions or operators.

Multiple source files

Most programs are broken up into multiple source files, and these files may depend on each other. To facilitate this design, declarations in *all* source files in a module (including the "main module" for an executable) are implicitly visible in each file's context. It is almost as if all these files had been loaded with `import`, but with a few important differences:

- The declarations in other files belong to the module being built, just like those in the current file. Therefore, if you need to refer to them by qualified name, you need to use the name of the module being built.
- A module is a fully-contained entity: it may depend on other modules, but those other modules can't depend on it. Source files within a module may have mutual dependencies.

> **FIXME**
>
> This wouldn't belong in the user model at all except for the implicit visibility thing. Is there a better way to talk about this?

Ambiguity

Because two different modules can declare the same name, it is sometimes necessary to use a *qualified name* to refer to a particular declaration:

```
import Chess
import Xiangqi

if userGame == "chess" {
  Chess.playGame()
} else if userGame == "xiangqi" {
  Xiangqi.playGame()
}
```

Here, both modules declare a function named `playGame` that takes no arguments, so we have to disambiguate by "qualifying" the function name with the appropriate module.

These are the rules for resolving name lookup ambiguities:

1. Declarations in the current source file are best.
2. Declarations from other files in the same module are better than declarations from imports.

3. Declarations from selective imports are better than declarations from non-selective imports. (This may be used to give priority to a particular module for a given name.)
4. Every source file implicitly imports the core standard library as a non-selective import.
5. If the name refers to a function, normal overload resolution may resolve ambiguities.

Submodules

> **Warning**
>
> This feature was never implemented, or even fully designed.

For large projects, it is usually desirable to break a single application or framework into subsystems, which Swift calls "submodules". A submodule is a development-time construct used for grouping within a module. By default, declarations within a submodule are considered "submodule-private", which means they are only visible within that submodule (rather than across the entire module). These declarations will not conflict with declarations in other submodules that may have the same name.

Declarations explicitly marked "whole-module" or "API" are still visible across the entire module (even if declared within a submodule), and must have a unique name within that space.

The *qualified name* of a declaration within a submodule consists of the top-level module name, followed by the submodule name, followed by the declaration.

> **Note**
>
> Submodules are an opportunity feature for Swift 1.0.

> **TODO**
>
> We need to decide once and for all whether implicit visibility applies across submodule boundaries, i.e. "can I access the public Swift.AST.Module from Swift.Sema without an import, or do I have to say `import Swift.AST`?"
>
> Advantages of module-wide implicit visibility:
>
> - Better name conflict checking. (The alternative is a linker error, or worse *no* linker error if the names have different manglings.)
> - Less work if things move around.
> - Build time performance is consistent whether or not you use this feature.
>
> Advantages of submodule-only implicit visibility:
>
> - Code completion will include names of public things you don't care about.
> - We haven't actually tested the build time performance of any large Swift projects, so we don't know if we can actually handle targets that contain hundreds of files.
> - Could be considered desirable to force declaring your internal dependencies explicitly.
> - In this mode, we could allow two "whole-module" declarations to have the same name, since they won't. (We could allow this in the other mode too but then the qualified name would always be required.)
>
> Both cases still use "submodule-only" as the default access control, so this only affects the implicit visibility of whole-module and public declarations.

Import Search Paths

> **FIXME**
>
> Write this section. Can source files be self-contained modules? How does -i mode work? Can the "wrong" module be found when looking for a dependency (i.e. can I substitute my own Foundation and expect AppKit to work)? How are modules stored on disk? How do hierarchical module names work?

Interoperability with Objective-C via Clang

The compiler has the ability to interoperate with C and Objective-C by importing *Clang modules <Clang module>*. This feature of the Clang compiler was developed to provide a "semantic import" extension to the C family of languages. The Swift compiler uses this to expose declarations from C and Objective-C as if they used native Swift types.

In all the examples above, `import AppKit` has been using this mechanism: the module found with the name "AppKit" is generated from the Objective-C AppKit framework.

Clang Submodules

Clang also has a concept of "submodules", which are essentially hierarchically- named modules. Unlike Swift's :ref:`submodules`,

Clang submodules are visible from outside the module. It is conventional for a top-level Clang module to re-export all of its submodules, but sometimes certain submodules are specified to require an explicit import:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)Modules.rst, line 338); *backlink***
>
> Unknown interpreted text role "ref".

```
import OpenGL.GL3
```

Module Overlays

> **Warning**
>
> This feature has mostly been removed from Swift; it's only in use in the "overlay" libraries bundled with Swift itself.

If a source file in module A includes `import A`, this indicates that the source file is providing a replacement or overlay for an external module. In most cases, the source file will *re-export* the underlying module, but add some convenience APIs to make the existing interface more Swift-friendly.

This replacement syntax (using the current module name in an import) cannot be used to overlay a Swift module, because :ref:`module-naming`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)Modules.rst, line 358); *backlink***
>
> Unknown interpreted text role "ref".

Multiple source files, part 2

In migrating from Objective-C to Swift, it is expected that a single program will contain a mix of sources. The compiler therefore allows importing a single Objective-C header, exposing its declarations to the main source file by constructing a sort of "ad hoc" module. These can then be used like any other declarations imported from C or Objective-C.

> **Note**
>
> This is describing the feature that eventually became "bridging headers" for app targets.

Accessing Swift declarations from Objective-C

> **Warning**
>
> This never actually happened; instead, we went with "generated headers" output by the Swift compiler.

Using the new `@import` syntax, Objective-C translation units can import Swift modules as well. Swift declarations will be mirrored into Objective-C and can be called natively, just as Objective-C declarations are mirrored into Swift for *Clang modules <Clang module>*. In this case, only the declarations compatible with Objective-C will be visible.

> **TODO**
>
> We need to actually do this, but it requires working on a branch of Clang, so we're pushing it back in the schedule as far as possible. The workaround is to manually write header files for imported Swift classes.

> **TODO**
>
> Importing Swift sources from within the same target is a goal, but there are many difficulties. How do you name a file to be imported? What if the file itself depends on another Objective-C header? What if there's a mutual dependency across the language boundary? (That's a problem in both directions, since both Clang modules and Swift modules are only supposed to be exposed once they've been type-checked.)

Glossary

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)Modules.rst, line 406)**
>
> Unknown directive type "glossary".

```
.. glossary::

  autolinking
    A technique where linking information is included in compiled object files,
    so that external dependencies can be recorded without having to explicitly
    specify them at link time.

  Clang module
    A module whose contents are generated from a C-family header or set of
    headers. See Clang's Modules__ documentation for more information.

    __ http://clang.llvm.org/docs/Modules.html

  framework
    A mechanism for library distribution on OS X. Traditionally contains header
    files describing the library's API, a binary file containing the
    implementation, and a directory containing any resources the library may
    need.

    Frameworks are also used on iOS, but as of iOS 7 custom frameworks cannot
    be created by users.

  import
    To locate and read a module, then make its declarations available in the
    current context.

  library
    Abstractly, a collection of APIs for a programmer to use, usually with a
    common theme. Concretely, the file containing the implementation of these
    APIs.

  mangled name
    A unique, internal name for a type or value. The term is most commonly used
    in C++; see Wikipedia__ for some examples. Swift's name mangling scheme is
    not the same as C++'s but serves a similar purpose.

    __ https://en.wikipedia.org/wiki/Name_mangling#C.2B.2B

  qualified name
    A multi-piece name like ``Foundation.NSWindow``, which names an entity
    within a particular context. This document is concerned with the case where
    the context is the name of an imported module.

  re-export
    To directly expose the API of one module through another module. Including
    the latter module in a source file will behave as if the user had also
    included the former module.

  serialized module
    A particular encoding of a module that contains declarations that have
    already been processed by the compiler. It may also contain implementations
    of some function declarations in `SIL` form.

  SIL
    "Swift Intermediate Language", a stable IR for the distribution of
    inlineable code.
```