Swift Generics

Scripting-language-like flexibility,
without sacrificing compile-time checks

Other kinds of type constraints

- Primitive/Value-Type
    - func add(lhs: Int, rhs: Int) -> Int
- Inheritance
    - func pushViewController(vc: UIViewController
- Protocol Conformance
    - var delegate: UITableViewDelegate?

Generics as Type Constraints

Express type relationships

You can use placeholders to specify where one or more types all need to be the same; and/or place constraints on one or more types.

Generics as Type Constraints

Express type relationships

```
func debugIt<T>(what: T) -> T {
  println("let's debug: \(what)")
  return what
}
```

Whatever kind of thing you give to this function, it will give you back a variable of the same type

In this case, idiomatically, you could probably expect the func to return the same thing it was given

Generics : Principles

You can achieve some similar effects in Objective-C, but it relies on using the “id” type, and doing run-time checks.

In Swift, with generics, you have the compiler working with you to enforce your design decisions at compile-time

Generics - Quick Example

```
//Objective-C
NSMutableArray* myNumbers = [NSMutableArray new];
[myNumbers addObject:@10];
[myNumbers addObject:@33];
[myNumbers addObject:@"mhuahahahaha"];
[myObjects enumerateObjectsUsingBlock:^(NSNumber* obj, NSUInteger idx, BOOL *stop) {
    [obj floatValue] + 1.0;
    //WHOOPS! I forgot to put a RUNTIME check to ensure everyone was using this as I expected
}];


//Swift
var myNumbers = [Int]
myNumbers.append(10)
myNumbers.append(33)
myNumbers.append("mhuahhahaha")//STOP RIGHT THERE!
//Compiler will not permit this
```

Generic Functions

Placeholders go after function name:

```
func functionName<TypePlaceholder>
```

Beyond that, you can use/not use the type placeholder anywhere else in the function definition where you would use a type:

```
func fizz<T>(a: Int, b: Int) -> Int {
    //Well, that was pointlessly generic...
    return 0
}
```

Generic Functions (contd.)

When invoking a generic function, you are not permitted to explicitly resolve the generic placeholder: It’s done implicitly with argument types

```
func doStuff<A>(stuff: A) -> A

doStuff(6)
//doStuff<Int>(6)//Not OK =(
```

Generic Types

Type placeholders go after the Type identifier.

```
enum Optional<T> {...}
struct Coordinates<T> {...}
class Thing<T> {...}
```

Generic Types (contd.)

You are permitted, but not required to explicitly specialize generic types at declaration.
So, you can say:

```
var foo = Optional<Int>.Some(10)
```

or:

```
var foo = Optional.Some(10)
```

Generic Types (contd.)

From the last example you may have guessed, this leaves the developer some leeway in style:
[all equivalent]

```
var baz = Optional<Float>.Some(10)
var baz = Optional.Some(Float(10))
var baz : Float? = Optional.Some(10)
var baz : Float? = 10
```

Generic Protocols

In the context of protocols, Swift has “Associated Types”

Standard Library Example:

```
protocol GeneratorType {

    /// The type of element generated by `self`.
    typealias Element

    /// Advance to the next element and return it, or `nil` if no next
    /// element exists.
    ///
    /// Requires: `next()` has not been applied to a copy of `self`
    /// since the copy was made, and no preceding call to `self.next()`
    /// has returned `nil`.  Specific implementations of this protocol
    /// are encouraged to respond to violations of this requirement by
    /// calling `preconditionFailure("...")`.
    mutating func next() -> Element?
}
```

We’ll revisit these in the playground

Protocols - Self

In a related way, protocols can make use of “Self” to specify that conformers must use their own type.

```
protocol AbsoluteValuable : SignedNumberType {

    /// Returns the absolute value of `x`
    static func abs(x: Self) -> Self
}
```

Associated Type Protocols

If you have an associated type, or use “Self” you will no longer be able to do things like:

```
if (someObj is SomeProtocol) {
```

You can only use it as a generic constraint

```
func doTheThing<T : SomeProtocol>(withWhat: T) {
```

Swift STD Library Generics

- struct Array<T>

- enum ImplicitlyUnwrappedOptional<T> : Reflectable, NilLiteralConvertible
    - Aren’t you glad they gave us “!”?

- struct Dictionary<Key: Hashable, Value>
    - Name your type-placeholders as makes sense

- prefix func !<T : BooleanType>(a: T) -> Bool
    - This style, (vs “a: BooleanType”) means that BooleanType is free to be an associated types protocol / use “Self”

Advanced Pt. 1/3

Multiple Type Constraints with Generics; the `where` clause:

```
func veryPickyFunction<T where T : Equatable, T : Comparable, T : SequenceType>(doStuff:
T)
```

[Swift Documentation]

Advanced Pt. 2/3

Swift uses “reified” generics, instead of type-erasure.
This means you are permitted to write something like this:

```
class SomeClass<A, B> {
    func doSomething(thing: A) {
        println("A")
    }
    func doSomething(thing: B) {
        println("B")
    }
}
```

In a type-erasure language (like Java), those would have the same signature

Advanced Pt. 3/3

Swift compiler is capable of generic specialization, but only does so if it thinks it’s more efficient. So this function:

```
func addEm(lhs: T, rhs: T) -> T {
    return lhs + rhs
}
addEm(1, 2)
addEm(3, 6)
```

might be secretly re-written as:

```
func addEm(lhs: Int, rhs: Int) -> Int
```

but it also might not =)

To the Playground!