What's New in Swift 4.1

Conditional Protocol Conformance

Marc Prud'hommeaux – marc@glimpse.io

Swift Office Hours – Intrepid, Boston, April 2018

The Humble "Either" Type:

```
/// A type that is either a T or a U
public enum Either<T, U> {
    case left(T)
    case right(U)
}
```

Either a This or a That...

Either a Cat or a Dog...

Either a String or an Int...

Taking it for a Test Drive:

```
let x: Either<String, Int> = .left("Foo")
let y: Either<String, Int> = .right(2)
let z: Either<String, Int> = .left("Foo")

let xEqualsY = x == y // should be false
let xEqualsZ = x == z // should be true
```

Test Drive Fail:

```
let x: Either<String, Int> = .left("Foo")
let y: Either<String, Int> = .right(2)
let z: Either<String, Int> = .left("Foo")
```

~~let xEqualsY = x == y~~

~~let xEqualsZ = x == z~~

Error: *Binary operator '==' cannot be applied to two 'Either<String, Int>' operands*

Automatic Implementation of equals & hashCode

```
/// A type that is either a T or a U, requiring that both types be equatable
public enum Either<T, U> : Equatable where T: Equatable, U: Equatable {
    case left(T)
    case right(U)
}

let x: Either<String, Int> = .left("Foo")
let y: Either<String, Int> = .right(2)
let z: Either<String, Int> = .left("Foo")

x == y // true
x == z // false
```

Problem Solved!

Implementing Equals for Either:

```
/// A type that is either a T or a U
public enum Either<T, U> {
    case left(T)
    case right(U)
}

/// When both T and U can be compared, then Either<T, U> can be compared
public extension Either where T: Equatable, U: Equatable {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.left(let a), .left(let b)): return a == b
        case (.right(let a), .right(let b)): return a == b
        case (.right, .left): return false
        case (.left, .right): return false
        }
    }
}
```

Test Drive (2nd try):

```
let x: Either<String, Int> = .left("Foo")
let y: Either<String, Int> = .right(2)
let z: Either<String, Int> = .left("Foo")

let xEqualsY = x == y // true
let xEqualsZ = x == z // false
```

Continuing the test drive:

```
let xs: Array<Either<String, Int>> = [x]
let ys = [y]
let zs = [z]


xs == ys // I would expect false
xs == zs // I would expect true
```

Another Test Drive Failure!

```
let xs: Array<Either<String, Int>> = [x]
let ys = [y]
let zs = [z]
```

```
xs == ys // I would expect false
xs == zs // I would expect true
```

Error: *“'<Self where Self : Equatable> (Self.Type) -> (Self, Self) -> Bool' requires that 'Either<String, Int>' conform to 'Equatable'”*

i.e.: “Types don’t automatically adopt a protocol just by satisfying its requirements. They must always explicitly declare their adoption of the protocol.”

Swift 4.0- Limitation:

```
extension Either where T: Equatable, U: Equatable {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right, .left), (.left, .right): return false
        }
    }
}
```

Swift 4.1+ Conditional Protocol Conformance:

```
extension Either : Equatable where T: Equatable, U: Equatable {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right, .left), (.left, .right): return false
        }
    }
}
```

And Now It Works:

```
let xs: Array<Either<String, Int>> = [x]
let ys = [y]
let zs = [z]


xs == ys // false
xs == zs // true
```

Furthermore:

```
let xos: Array<Optional<Either<String, Int>>> = [x, nil]
let yos = [y, nil]
let zos = [z, nil]


xos == yos // false
xos == zos // true
```

And Even?

```
let xot: Set<Optional<Either<String, Int>>> = Set(arrayLiteral: x, nil)
let yot = Set(arrayLiteral: y, nil)
let zot = Set(arrayLiteral: z, nil)


xot == yot // should be false
xot == zot // should be true
```

Not Quite Yet...

```
let xot: Set<Optional<Either<String, Int>>> = Set(arrayLiteral: x, nil)
```

Error: *type 'Optional<Either<String, Int>>' does not conform to protocol 'Hashable'*

Solution: Make Either Hashable

```
extension Either : Hashable where T: Hashable, U: Hashable {
    public var hashValue: Int {
        switch self {
        case .left(let x): return x.hashValue
        case .right(let y): return y.hashValue
        }
    }
}
```

Does It Work Yet?

```
let xot: Set<Optional<Either<String, Int>>> = Set(arrayLiteral: x, nil)
```

Error: *type 'Optional<Either<String, Int>>' does not conform to protocol 'Hashable'*

Optional does not have a conditional Hashable conformance…

```
extension Optional : Hashable where Wrapped : Hashable {
    public var hashValue: Int {
        switch self {
        case .some(let x): return x.hashValue
        case .none: return 0 // or whatevs…
        }
    }
}
```

Success!

```
let xot: Set<Optional<Either<String, Int>>> = Set(arrayLiteral: x, nil)
let yot = Set(arrayLiteral: y, nil)
let zot = Set(arrayLiteral: z, nil)


xot == yot // false
xot == zot // true
```

```
/// A type that is either a T or a U
public enum Either<T, U> {
    case left(T)
    case right(U)
}

extension Either : Equatable where T: Equatable, U: Equatable {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right, .left), (.left, .right): return false
        }
    }
}

extension Either : Hashable where T: Hashable, U: Hashable {
    public var hashValue: Int {
        switch self {
        case .left(let x): return x.hashValue
        case .right(let y): return y.hashValue
        }
    }
}

extension Optional : Hashable where Wrapped : Hashable {
    public var hashValue: Int {
        switch self {
        case .some(let x): return x.hashValue
        case .none: return 0
        }
    }
}
```

Limitation:

Say you want:

```
Either<Int, Int>.left(1) == Either<Int, Int>.right(1)
```

You could implement:

```
extension Either : Equatable where T: Equatable, U: Equatable, T == U {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right(let a), .left(let b)): return a == b
        case (.left(let a), .right(let b)): return a == b
        }
    }
}
```

But the you lose some conformity:

```swift
extension Either : Equatable where T: Equatable, U: Equatable, T == U {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right(let a), .left(let b)): return a == b
        case (.left(let a), .right(let b)): return a == b
        }
    }
}

Either<Int, Int>.left(1) == Either<Int, Int>.right(1)
Either<Int, String>.left(1) == Either<Int, String>.right("Foo")
```

Error: *error: binary operator '==(:_:)' cannot be applied to operands of type 'Either<Int, String>' and 'Either<Int, String>'*

```
extension Either : Equatable where T: Equatable, U: Equatable, T == U {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right(let a), .left(let b)): return a == b
        case (.left(let a), .right(let b)): return a == b
        }
    }
}

extension Either : Equatable where T: Equatable, U: Equatable {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right, .left), (.left, .right): return false
        }
    }
}

Either<Int, Int>.left(1) == Either<Int, Int>.right(1) // should succeed
Either<Int, String>.left(1) == Either<Int, String>.right("Foo") // should fail
```

```
extension Either : Equatable where T: Equatable, U: Equatable, T == U {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right(let a), .left(let b)): return a == b
        case (.left(let a), .right(let b)): return a == b
        }
    }
}

extension Either : Equatable where T: Equatable, U: Equatable {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right, .left), (.left, .right): return false
        }
    }
}
```

Error: *error: redundant conformance of 'Either<T, U>' to protocol 'Equatable'*

```
extension Either : Equatable where T: Equatable, U: Equatable, T == U {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right(let a), .left(let b)): return a == b
        case (.left(let a), .right(let b)): return a == b
        }
    }
}

extension Either : Equatable where T: Equatable, U: Equatable, T != U {
    public static func ==(lhs: Either<T, U>, rhs: Either<T, U>) -> Bool {
        switch (lhs, rhs) {
        case (.right(let a), .right(let b)): return a == b
        case (.left(let a), .left(let b)): return a == b
        case (.right, .left), (.left, .right): return false
        }
    }
}
```

Error: *error: redundant conformance of 'Either<T, U>' to protocol 'Equatable'*

Standard Library Equatable Implementations

```
extension Optional: Equatable where Wrapped: Equatable { }
extension Array: Equatable where Element: Equatable { }
extension ArraySlice: Equatable where Element: Equatable { }
extension ContiguousArray: Equatable where Element: Equatable { }
extension Dictionary: Equatable where Value: Equatable { }
```

Standard Library Hashable Implementations

```
extension Optional: Hashable where Wrapped: Hashable { }
extension Array: Hashable where Element: Hashable { }
extension ArraySlice: Hashable where Element: Hashable { }
extension ContiguousArray: Hashable where Element: Hashable { }
extension Dictionary: Hashable where Value: Hashable { }
extension Range: Hashable where Bound: Hashable { }
extension ClosedRange: Hashable where Bound: Hashable { }
```

Further Reading:

https://github.com/apple/swift-evolution/blob/master/proposals/0143-conditional-conformances.md

https://swift.org/blog/conditional-conformance/

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

Questions?

Marc Prud'hommeaux / marc@glimpse.io