Swift Generics

The 5 Stages of PATs

David Hart

iOS Developer at Atipik by 🌞

David Hart

SwiftPM Contributor by 🌝


atipik


Redux

Redux

Single source of truth

State is read-only

Changes are made with pure functions

Redux

```
struct AppState {
    var todos: [UUID: Todo]
}
```

```
struct Todo {
    let uuid: UUID
    var text: String
}

protocol Action {
    associatedtype State
    func reduce(_ state: inout State)
}
```

Redux

```
struct CreateTodo: Action {
    let uuid: UUID

    func reduce(_ state: inout AppState) {
        state.todos[uuid] = Todo(uuid: uuid, text: "New Todo")
    }
}

struct SetTodoText: Action {
    let uuid: UUID
    let text: String

    func reduce(_ state: inout AppState) {
        state.todos[uuid]!.text = text
    }
}
```

Redux

```
func duplicate(todo: Todo) -> [Action] {
    let uuid = UUID()
    return [
        CreateTodo(uuid: uuid),
        SetTodoText(uuid: uuid, text: todo.text)
    ]
}
```

Redux

```
func duplicate(todo: Todo) -> [Action] {
```

Protocol 'Action' can only be used as a generic constraint because it has Self or associated type requirements

```
        SetTodoText(uuid: uuid, text: todo.text)
    ]
}
```

Anger

Denial

Bargaining

5 stages of PATs

Depression

Acceptance

Denial

"I must be doing something wrong."

Developer Tools #WWDC15

Protocol-Oriented Programming in Swift

Session 408

Dave Abrahams Professor of Blowing-Your-Mind

Two Worlds of Protocols

Without Self Requirement	With Self Requirement
`func precedes(other: Ordered) -> Bool`	`func precedes(other: Self) -> Bool`
Usable as a type	Only usable as a generic constraint
`func sort(inout a: [Ordered])`	`func sort<T : Ordered>(inout a: [T])`
Think "heterogeneous"	Think "homogeneous"
Every model must deal with all others	Models are free from interaction
Dynamic dispatch	Static dispatch
Less optimizable	More optimizable

Anger

"I'm never using protocols ever again!"

If all you have is ~~a hammer~~ Protocol-Oriented Programming, everything looks like a ~~nail~~ protocol.

Classes

```
open class Action<State> {
    open func reduce(_ state: inout State) {
        // Override in subclass.
    }
}
```

Classes

```
final class SetTodoText: Action<AppState> {
    let uuid: UUID
    let text: String

    init(uuid: UUID, text: String) {
        self.uuid = uuid
        self.text = text
    }

    override func reduce(_ state: inout AppState) {
        state.todos[uuid]!.text = text
    }
}
```

When to Use Classes

They do have their place…

You *want* implicit sharing when

- Copying or comparing instances doesn't make sense (e.g., Window)
- Instance lifetime is tied to external effects (e.g., TemporaryFile)
- Instances are just "sinks"—write-only conduits to external state (e.g., CGContext)

Bargaining 

"I wish protocols were generic instead of having associated types."

— Dave DeLong

Generic protocols

One of the most commonly requested features is the ability to parameterize protocols themselves. For example, a protocol that indicates that the `Self` type can be constructed from some specified type `T`:

```
protocol ConstructibleFromValue<T> {
  init(_ value: T)
}
```

Implicit in this feature is the ability for a given type to conform to the protocol in two different ways. A `Real` type might be constructible from both `Float` and `Double`, e.g.,

```
struct Real { ... }
extension Real : ConstructibleFrom<Float> {
  init(_ value: Float) { ... }
}
extension Real : ConstructibleFrom<Double> {
  init(_ value: Double) { ... }
}
```

Most of the requests for this feature actually want a different feature. They tend to use a parameterized `Sequence` as an example, e.g.,

```
protocol Sequence<Element> { ... }

func foo(strings: Sequence<String>) {  /// works on any sequence containing Strings
  // ...
}
```

The actual requested feature here is the ability to say "Any type that conforms to `Sequence` whose `Element` type is `String`", which is covered by the section on "Generalized existentials", below.

More importantly, modeling `Sequence` with generic parameters rather than associated types is tantalizing but wrong: you

Generic Protocols

```
protocol Action<State> {
    func reduce(_ state: inout State)
}
```

```
// from
func sort<C: Collection>(_ todos: C) where C.Element == Todo { … }

// to
func sort(_ todos: Collection<Todo>) { … }

// actually
func sort<I>(_ todos: Collection<Todo, I>) { … }
```

Generic Protocols

```
protocol ConvertibleFrom<T> {
    init(_ value: T)
}
```

```
struct Real { ... }

extension Real: ConstructibleFrom<Float> {
  init(_ value: Float) { ... }
}

extension Real: ConstructibleFrom<Double> {
  init(_ value: Double) { ... }
}
```

Depression 

"They're these parasitic things that end up infesting your code in ways you really don't want."

— Dave DeLong

AnyIterator

AnyHashable

AnySequence

AnyBidirectionalCollection

Type-Erased Wrappers

AnyKeyPath

AnyCollection

AnyIndex

AnyRandomAccessCollection

Type-Erased Wrappers

```
protocol Action {
    associatedtype State
    func reduce(_ state: inout State)
}
```

```
struct AnyAction<State>: Action {
    private let reduceClosure: (inout State) -> Void

    init<A: Action>(_ action: A) where A.State == State {
        reduceClosure = { action.reduce(&$0) }
    }

    func reduce(_ state: inout State) {
        reduceClosure(&state)
    }
}
```

Type-Erased Wrappers

```
func duplicate(todo: Todo) -> [AnyAction<AppState>] {
    let uuid = UUID()
    return [
        AnyAction(CreateTodo(uuid: uuid)),
        AnyAction(SetTodoText(uuid: uuid, text: todo.text))
    ]
}
```

Acceptance 

"I wish the compiler generated type erasers for me."

— Dave DeLong

Generalized existentials

The restrictions on existential types came from an implementation limitation, but it is reasonable to allow a value of protocol type even when the protocol has Self constraints or associated types. For example, consider `IteratorProtocol` again and how it could be used as an existential:

```
protocol IteratorProtocol {
  associatedtype Element
  mutating func next() -> Element?
}

let it: IteratorProtocol = ...
it.next()   // if this is permitted, it could return an "Any?", i.e., the existential that wraps the act
```

Additionally, it is reasonable to want to constrain the associated types of an existential, e.g., "a `Sequence` whose element type is `String`" could be expressed by putting a where clause into `protocol<...>` or `Any<...>` (per "Renaming `protocol<...>` to `Any<...>`"):

```
let strings: Any<Sequence where .Iterator.Element == String> = ["a", "b", "c"]
```

The leading `.` indicates that we're talking about the dynamic type, i.e., the `Self` type that's conforming to the `Sequence` protocol. There's no reason why we cannot support arbitrary `where` clauses within the `Any<...>`. This very-general syntax is a bit unwieldy, but common cases can easily be wrapped up in a generic typealias (see the section "Generic typealiases" above):

```
typealias AnySequence<Element> = Any<Sequence where .Iterator.Element == Element>
let strings: AnySequence<String> = ["a", "b", "c"]
```

Opening existentials

Generalized Existentials

```
typealias Codable = Encodable & Decodable

typealias TableViewController = UIViewController & UITableViewDelegate

typealias AnyCollection<T> = Collection where .Element == T

typealias AnyAction<S> = Action where .State == S
```

Type-Erased Wrappers

```
func duplicate(todo: Todo) -> [AnyAction<AppState>] {
    let uuid = UUID()
    return [
        CreateTodo(uuid: uuid),
        SetTodoText(uuid: uuid, text: todo.text)
    ]
}
```

The 5 Stages of PATs

PATs are a PITA

OOP is not dead

Generic Protocols

Type-erasers FTW

Generalized Existentials


JUL
17

Follow up

Generics Manifesto

https://github.com/apple/swift/blob/master/docs/GenericsManifesto.md

Generalized Existentials Proposal

https://github.com/austinzheng/swift-evolution/blob/az-existentials/proposals/XXXX-enhanced-existentials.md

Rust - Advanced Traits

https://doc.rust-lang.org/book/second-edition/ch19-03-advanced-traits.html

📖 Advanced Swift

https://www.objc.io/books/advanced-swift/

🙋‍♂️ Come talk to me

AppBuilders

@icanzilb

Thanks!

hartbit

dhartbit