Protocols

A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.

You define protocols in a very similar way to classes, structures, and enumerations:

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

```
struct SomeStructure: FirstProtocol,
AnotherProtocol {
    // structure definition goes here
}
```

Property requirements are always declared as variable properties, prefixed with the `var` keyword. Gettable and settable properties are indicated by writing `{ get set }` after their type declaration, and gettable properties are indicated by writing `{ get }`.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Protocol Inheritance

A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol InheritingProtocol: SomeProtocol,
AnotherProtocol {
    // protocol definition goes here
}
```

Here's an example of a protocol that inherits the `TextRepresentable` protocol from above:

```
protocol PrettyTextRepresentable:
TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

This example defines a new protocol, `PrettyTextRepresentable`, which inherits from `TextRepresentable`. Anything that adopts `PrettyTextRepresentable` must satisfy all of the requirements enforced by `TextRepresentable`, *plus* the additional requirements enforced by `PrettyTextRepresentable`. In this example, `PrettyTextRepresentable` adds a single requirement to provide a gettable property called `prettyTextualDescription` that returns a `String`.

Protocol Extensions

Protocols can be extended to provide method, initializer, subscript, and computed property implementations to

conforming types.

For example, the `RandomNumberGenerator` protocol can be extended to provide a `randomBool()` method, which uses the result of the required `random()` method to return a random `Bool` value:

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

Providing Default Implementations

You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol.

For example, the `PrettyTextRepresentable` protocol, which inherits the `TextRepresentable` protocol can provide a default implementation of its required `prettyTextualDescription` property to simply return the result of accessing the `textualDescription` property:

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```