Protocol Types Cannot Conform to Protocols

In Swift, a protocol that does not have `Self` or associated type requirements can be used as a type. You can use a variable or constant of a protocol type, also called an **existential type**, to hold a value of any conforming type:

```
protocol Animal {
    func makeNoise()
    static var species: String { get }
}
struct Dog: Animal {
    func makeNoise() { print("Woof") }
    static var species: String = "Canus familiaris"
}
struct Cat: Animal {
    func makeNoise() { print("Meow") }
    static var species: String = "Felis catus"
}

var animal: Animal // `Animal` is used here as a type.
animal = Dog()
animal.makeNoise() // Prints "Woof".
animal = Cat()
animal.makeNoise() // Prints "Meow".
```

Notice that it is possible to invoke the method `makeNoise()` on a value of type `Animal`, just as it is possible to do so on a value of concrete type `Dog` or `Cat`. However, the static property `species` is not available for the existential type:

```
print(Dog.species)    // Prints "Canus familiaris"
print(Cat.species)    // Prints "Felis catus"
print(Animal.species) // error: static member 'species' cannot be used...
```

Since a type conforms to a protocol only when it satisfies *all* of that protocol's requirements, the existential type `Animal` does not conform to the protocol `Animal` because it cannot satisfy the protocol's requirement for the static property `species`:

```
func declareAnimalSpecies<T: Animal>(_ animal: T) {
    animal.makeNoise()
    print("My species is known as \(T.species)")
}

let dog = Dog()
declareAnimalSpecies(dog)
// Prints:
// "Woof"
// "My species is known as Canus familiaris"
```

```
declareAnimalSpecies(animal)
// error: protocol type 'Animal' cannot conform to 'Animal'...
```

In general, any initializers, static members, and associated types required by a protocol can be used only via conforming concrete types. Although Swift allows a protocol that requires initializers or static members to be used as a type, that type *does not and cannot* conform to the protocol itself.

Currently, even if a protocol `P` requires no initializers or static members, the existential type `P` does not conform to `P` (with exceptions below). This restriction allows library authors to add such requirements (initializers or static members) to an existing protocol without breaking their users' source code.

Exceptions

When used as a type, the Swift protocol `Error` conforms to itself; `@objc` protocols with no static requirements can also be used as types that conform to themselves.

Alternatives

Concrete types that *do* conform to protocols can provide functionality similar to that of existential types. For example, the standard library provides the `AnyHashable` type for `Hashable` values. Manual implementation of such **type erasure** can require specific knowledge of the semantic requirements for each protocol involved and is beyond the scope of this discussion.

In certain scenarios, you might avoid any need for manual type erasure by reworking generic APIs to use existential types instead:

```
func declareAnimalSpeciesDynamically(_ animal: Animal) {
    animal.makeNoise()
    print("My species is known as \(type(of: animal).species)")
}

declareAnimalSpeciesDynamically(animal)
// Prints:
// "Meow"
// "My species is known as Felis catus"
```

(Note that there is a distinction between the *static* type of a value as given by the generic parameter `T` and the *dynamic* type of a value obtained by invoking `type(of:)`. For example, the static type of `animal` is `Animal`, while its dynamic type is `Cat`. Therefore, the two functions `declareAnimalSpecies(_:)` and `declareAnimalSpeciesDynamically(_:)` are not exact replacements of each other.)

The same technique might be applicable to members of generic types:

```swift
// Instead of...
struct Habitat<T: Animal> {
    var animal: T
}
// ...consider:
struct Habitat {
    var animal: Animal
}
```

For more on using existential types, see Protocols as Types in *The Swift Programming Language*.