ON THIS PAGE

# Opaque Types

A function or method with an opaque return type hides its return value's type information. Instead of providing a concrete type as the function's return type, the return value is described in terms of the protocols it supports. Hiding type information is useful at boundaries between a module and code that calls into the module, because the underlying type of the return value can remain private. Unlike returning a value whose type is a protocol type, opaque types preserve type identity—the compiler has access to the type information, but clients of the module don't.

## The Problem That Opaque Types Solve

For example, suppose you're writing a module that draws ASCII art shapes. The basic characteristic of an ASCII art shape is a `draw()` function that returns the string representation of that shape, which you can use as the requirement for the `Shape` protocol:

```
protocol Shape {
    func draw() -> String
}

struct Triangle: Shape {
    var size: Int
    func draw() -> String {
        var result = [String]()
        for length in 1...size {
            result.append(String(repeating: "*", count:
  length))
```

```
        }
        return result.joined(separator: "\n")
    }
}
let smallTriangle = Triangle(size: 3)
print(smallTriangle.draw())
// *
// **
// ***
```

You could use generics to implement operations like flipping a shape vertically, as shown in the code below. However, there's an important limitation to this approach: The flipped result exposes the exact generic types that were used to create it.

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}
let flippedTriangle = FlippedShape(shape: smallTriangle)
print(flippedTriangle.draw())
// ***
// **
// *
```

This approach to defining a `JoinedShape<T: Shape, U: Shape>` structure that joins two shapes together vertically, like the code below shows, results in types like `JoinedShape<FlippedShape<Triangle>, Triangle>` from joining a flipped triangle with another triangle.

```
struct JoinedShape<T: Shape, U: Shape>: Shape {
    var top: T
    var bottom: U
    func draw() -> String {
        return top.draw() + "\n" + bottom.draw()
    }
}
```

```
let joinedTriangles = JoinedShape(top: smallTriangle,
  bottom: flippedTriangle)
print(joinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *
```

Exposing detailed information about the creation of a shape allows types that aren't meant to be part of the ASCII art module's public interface to leak out because of the need to state the full return type. The code inside the module could build up the same shape in a variety of ways, and other code outside the module that uses the shape shouldn't have to account for the implementation details about the list of transformations. Wrapper types like `JoinedShape` and `FlippedShape` don't matter to the module's users, and they shouldn't be visible. The module's public interface consists of operations like joining and flipping a shape, and those operations return another `Shape` value.

# Returning an Opaque Type

You can think of an opaque type like being the reverse of a generic type. Generic types let the code that calls a function pick the type for that function's parameters and return value in a way that's abstracted away from the function implementation. For example, the function in the following code returns a type that depends on its caller:

```
func max<T>(_ x: T, _ y: T) -> T where T: Comparable { ... }
```

The code that calls `max(_:_:)` chooses the values for `x` and `y`, and the type of those values determines the concrete type of `T`. The calling code can use any type that conforms to the `Comparable` protocol. The code inside the function is written in a general way so it can handle whatever type the caller provides. The implementation of `max(_:_:)` uses only functionality that all `Comparable` types share.

Those roles are reversed for a function with an opaque return type. An opaque type lets the function implementation pick the type for the value it returns in a way that's abstracted away from the code that calls the function. For example, the function in the following example returns a trapezoid without exposing the underlying type of that shape.

```
struct Square: Shape {
    var size: Int
    func draw() -> String {
        let line = String(repeating: "*", count: size)
        let result = Array<String>(repeating: line, count:
  size)
        return result.joined(separator: "\n")
    }
}

func makeTrapezoid() -> some Shape {
    let top = Triangle(size: 2)
    let middle = Square(size: 2)
    let bottom = FlippedShape(shape: top)
    let trapezoid = JoinedShape(
        top: top,
        bottom: JoinedShape(top: middle, bottom: bottom)
    )
    return trapezoid
}
let trapezoid = makeTrapezoid()
print(trapezoid.draw())
// *
// **
// **
// **
// **
// *
```

The `makeTrapezoid()` function in this example declares its return type as `some Shape`; as a result, the function returns a value of some given type that conforms to the `Shape` protocol, without specifying any particular concrete type. Writing `makeTrapezoid()` this way lets it express the fundamental aspect of its public interface—the value it returns is a shape—without making the specific types that the shape is made from a part of its public interface. This implementation uses two triangles and a square, but the function could be rewritten to draw a trapezoid in a variety of other ways without changing its return type.

This example highlights the way that an opaque return type is like the reverse of a generic type. The code inside `makeTrapezoid()` can return any type it needs to, as long as that type conforms to the `Shape` protocol, like the calling code does for a generic function. The code that calls the function needs to be written in a general way, like the implementation of a generic function, so that it can work with any `Shape` value that's returned by `makeTrapezoid()`.

You can also combine opaque return types with generics. The functions in the following code both return a value of some type that conforms to the `Shape` protocol.

```
func flip<T: Shape>(_ shape: T) -> some Shape {
    return FlippedShape(shape: shape)
}
func join<T: Shape, U: Shape>(_ top: T, _ bottom: U) -> some
  Shape {
    JoinedShape(top: top, bottom: bottom)
}

let opaqueJoinedTriangles = join(smallTriangle,
  flip(smallTriangle))
print(opaqueJoinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *
```

The value of `opaqueJoinedTriangles` in this example is the same as `joinedTriangles` in the generics example in the The Problem That Opaque Types Solve section earlier in this chapter. However, unlike the value in that example, `flip(_:)` and `join(_:_:)` wrap the underlying types that the generic shape operations return in an opaque return type, which prevents those types from being visible. Both functions are generic because the types they rely on are generic, and the type parameters to the function pass along the type information needed by `FlippedShape` and `JoinedShape`.

If a function with an opaque return type returns from multiple places, all of the possible return values must have the same type. For a generic function, that return type can use the function's generic type parameters, but it must still be a single type. For example, here's an *invalid* version of the shape-flipping function that includes a special case for squares:

```
func invalidFlip<T: Shape>(_ shape: T) -> some Shape {
    if shape is Square {
        return shape // Error: return types don't match
    }
    return FlippedShape(shape: shape) // Error: return types
  don't match
}
```

If you call this function with a `Square`, it returns a `Square`; otherwise, it returns a `FlippedShape`. This violates the requirement to return values of only one type and makes `invalidFlip(_:)` invalid code. One way to fix `invalidFlip(_:)` is to move the special case for squares into the implementation of `FlippedShape`, which lets this function always return a `FlippedShape` value:

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        if shape is Square {
            return shape.draw()
        }
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}
```

The requirement to always return a single type doesn't prevent you from using generics in an opaque return type. Here's an example of a function that incorporates its type parameter into the underlying type of the value it returns:

```
func `repeat`<T: Shape>(shape: T, count: Int) -> some
  Collection {
    return Array<T>(repeating: shape, count: count)
}
```

In this case, the underlying type of the return value varies depending on `T`: Whatever shape is passed it, `repeat(shape:count:)` creates and returns an array of that shape. Nevertheless, the return value always has the same underlying type of `[T]`, so it follows the requirement that functions with opaque return types must return values of only a single type.

# Differences Between Opaque Types and Protocol Types

Returning an opaque type looks very similar to using a protocol type as the return type of a function, but these two kinds of return type differ in whether they preserve type identity. An opaque type refers to one specific type, although the caller of the function isn't able to see which type; a protocol type can refer to any type that conforms to the protocol. Generally speaking, protocol types give you more flexibility about the underlying types of the values they store, and opaque types let you make stronger guarantees about those underlying types.

For example, here's a version of `flip(_:)` that returns a value of protocol type instead of using an opaque return type:

```
func protoFlip<T: Shape>(_ shape: T) -> Shape {
    return FlippedShape(shape: shape)
}
```

This version of `protoFlip(_:)` has the same body as `flip(_:)`, and it always returns a value of the same type. Unlike `flip(_:)`, the value that `protoFlip(_:)` returns isn't required to always have the same type—it just has to conform to the `Shape` protocol. Put another way, `protoFlip(_:)` makes a much looser API contract with its caller than `flip(_:)` makes. It reserves the flexibility to return values of multiple types:

```
func protoFlip<T: Shape>(_ shape: T) -> Shape {
    if shape is Square {
        return shape
    }

    return FlippedShape(shape: shape)
}
```

The revised version of the code returns an instance of `Square` or an instance of `FlippedShape`, depending on what shape is passed in. Two flipped shapes returned by this function might have completely different types. Other valid versions of this function could return values of different types when flipping multiple instances of the same shape. The less specific return type information from `protoFlip(_:)` means that many operations that depend on type information aren't available on the returned value. For example, it's not possible to write an `==` operator comparing results returned by this function.

```
let protoFlippedTriangle = protoFlip(smallTriangle)
let sameThing = protoFlip(smallTriangle)
protoFlippedTriangle == sameThing  // Error
```

The error on the last line of the example occurs for several reasons. The immediate issue is that the `Shape` doesn't include an `==` operator as part of its protocol requirements. If you try adding one, the next issue you'll encounter is that the `==` operator needs to know the types of its left-hand and right-hand arguments. This sort of operator usually takes arguments of type `Self`, matching whatever concrete type adopts the protocol, but adding a `Self` requirement to the protocol doesn't allow for the type erasure that happens when you use the protocol as a type.

Using a protocol type as the return type for a function gives you the flexibility to return any type that conforms to the protocol. However, the cost of that flexibility is that some operations aren't possible on the returned values. The example shows how the == operator isn't available—it depends on specific type information that isn't preserved by using a protocol type.

Another problem with this approach is that the shape transformations don't nest. The result of flipping a triangle is a value of type `Shape`, and the `protoFlip(_:)` function takes an argument of some type that conforms to the `Shape` protocol. However, a value of a protocol type doesn't conform to that protocol; the value returned by `protoFlip(_:)` doesn't conform to `Shape`. This means code like `protoFlip(protoFlip(smallTriange))` that applies multiple transformations is invalid because the flipped shape isn't a valid argument to `protoFlip(_:)`.

In contrast, opaque types preserve the identity of the underlying type. Swift can infer associated types, which lets you use an opaque return value in places where a protocol type can't be used as a return value. For example, here's a version of the `Container` protocol from Generics:

```
protocol Container {
    associatedtype Item
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
extension Array: Container { }
```

You can't use `Container` as the return type of a function because that protocol has an associated type. You also can't use it as constraint a generic return type because there isn't enough information outside the function body to infer what the generic type needs to be.

```
// Error: Protocol with associated types can't be used as a
  return type.
func makeProtocolContainer<T>(item: T) -> Container {
    return [item]
}

// Error: Not enough information to infer C.
func makeProtocolContainer<T, C: Container>(item: T) -> C {
```

```
    return [item]
}
```

Using the opaque type `some Container` as a return type expresses the desired API contract—the function returns a container, but declines to specify the container's type:

```
func makeOpaqueContainer<T>(item: T) -> some Container {
    return [item]
}
let opaqueContainer = makeOpaqueContainer(item: 12)
let twelve = opaqueContainer[0]
print(type(of: twelve))
// Prints "Int"
```

The type of `twelve` is inferred to be `Int`, which illustrates the fact that type inference works with opaque types. In the implementation of `makeOpaqueContainer(item:)`, the underlying type of the opaque container is `[T]`. In this case, `T` is `Int`, so the return value is an array of integers and the `Item` associated type is inferred to be `Int`. The subscript on `Container` returns `Item`, which means that the type of `twelve` is also inferred to be `Int`.

‹ Generics Automatic Reference Counting ›

BETA SOFTWARE

This documentation contains preliminary information about an API or technology in development. This information is subject to change, and software implemented according to this documentation should be tested with final operating system software.

Learn more about using Apple's beta software