Swift has a variety of types, including structs, classes, enums, protocols, and generics. Each type has its own unique properties and uses.

Structs

Structs are value types, which means that they are copied whenever they are passed to a function or assigned to a variable. Structs are typically used to represent small, immutable pieces of data, such as a point in 2D space or a user's name.

Swift
```swift
struct Point {
    var x: Double
    var y: Double
}

var point1 = Point(x: 10.0, y: 20.0)
var point2 = point1

point1.x = 30.0

print(point1.x) // 30.0
print(point2.x) // 10.0
```

In this example, the Point struct is used to represent a point in 2D space. The point1 and point2 variables are both assigned the same Point struct. When the point1.x property is changed, the point2.x property is not affected, because structs are copied whenever they are passed to a function or assigned to a variable.

Classes

Classes are reference types, which means that they are not copied when they are passed to a function or assigned to a variable. Instead, a reference to the class is passed or assigned. Classes are typically used to represent complex data structures, such as a user account or a view in a user interface.

Swift
```swift
class User {
    var name: String
    var email: String
    init(name: String, email: String) {
        self.name = name
        self.email = email
    }
}

var user1 = User(name: "John Doe", email: "john.doe@example.com")
var user2 = user1

user1.name = "Jane Doe"

print(user1.name) // Jane Doe
print(user2.name) // Jane Doe
```

In this example, the User class is used to represent a user account. The user1 and user2 variables are both assigned the same User class. When the user1.name property is changed, the user2.name property is also affected, because classes are reference types.

Enums

Enums are used to represent a fixed set of values. For example, an enum could be used to represent the different states of a traffic light (red, yellow, and green).

Swift
```
enum TrafficLight {
    case red
    case yellow
    case green
}

var trafficLight = TrafficLight.red

switch trafficLight {
case .red:
    print("Stop")
case .yellow:
    print("Caution")
case .green:
    print("Go")
}
```

In this example, the TrafficLight enum is used to represent the different states of a traffic light. The trafficLight variable is assigned the red state. When the trafficLight variable is passed to the switch statement, the appropriate case is executed based on the current state of the traffic light.

Protocols

Protocols define a set of methods that a type must implement. Protocols can be used to implement polymorphism and dependency injection.

Swift
```swift
protocol Vehicle {
    func drive()
    func brake()
}

class Car: Vehicle {
    func drive() {
        print("Driving car")
    }

    func brake() {
        print("Braking car")
    }
}

class Bicycle: Vehicle {
    func drive() {
        print("Pedaling bicycle")
    }

    func brake() {
        print("Squeezing bicycle brakes")
    }
}

func drive(_ vehicle: Vehicle) {
    vehicle.drive()
}

let car = Car()
let bicycle = Bicycle()

drive(car) // Driving car
drive(bicycle) // Pedaling bicycle
```

In this example, the Vehicle protocol defines a set of methods that a vehicle must implement. The Car and Bicycle classes both implement the Vehicle protocol. The drive(_:) function can be used to drive any type that implements the Vehicle protocol.

Generics

Generics allow you to write code that can be used with different types. Generics are often used to implement data structures, such as arrays and dictionaries.

Swift
```swift
struct Array<T> {
    private var elements: [T] = []

    mutating func append(_ element: T) {
        elements.append(element)
    }

    func count() -> Int {
        return elements.count
    }
}
```

The Array<T> type can be used to store an array of values of any type. To create an Array<T> object, you need to specify the type argument when you create the object. For example, the following code creates an Array<Int> object:

Swift
```swift
let array = Array<Int>()
```

The array object can be used to store an array of integers.

Generics are a powerful tool that can be used to write reusable and flexible code. By using generics, you can avoid writing duplicate code and make your code more efficient.

Functions

Functions are blocks of code that can be executed and reused. Functions can take parameters and return values. Functions are a good way to organize your code and make it more modular.

Swift
```swift
func sum(a: Int, b: Int) -> Int {
    return a + b
}
```

```swift
let result = sum(a: 1, b: 2)
print(result) // 3
```

In this example, the sum() function takes two integers as input and returns the sum of the two integers. The result variable is assigned the result of calling the sum() function with the arguments 1 and 2. The print() function is then used to print the value of the result variable to the console.

Functions can also be generic, which means that they can work with different types. For example, the following function can be used to sum two values of any type that conforms to the Numeric protocol:

Swift
```swift
func sum<T: Numeric>(a: T, b: T) -> T {
    return a + b
}

let result1 = sum(a: 1, b: 2) // 3
let result2 = sum(a: 1.0, b: 2.0) // 3.0
```

In this example, the sum() function takes two values of any type that conforms to the Numeric protocol as input and returns the sum of the two values. The Numeric protocol defines a set of methods that must be implemented by any type that wants to conform to the protocol. These methods include methods for adding, subtracting, multiplying, and dividing values.

Functions can also have multiple parameters and return values. For example, the following function takes two strings as input and returns a concatenated string:

Swift
```swift
func concatenate(a: String, b: String) -> String {
    return a + b
}

let result = concatenate(a: "Hello", b: " world!")
print(result) // Hello world!
```

In this example, the concatenate() function takes two strings as input and returns a concatenated string. The result variable is assigned the result of calling the concatenate() function with the arguments "Hello" and " world!". The print() function is then used to print the value of the result variable to the console.

Functions are a powerful tool that can be used to write reusable and efficient code.