# Chapter 20: Classes

## Section 20.1: Defining a Class

You define a class like this:

```
class Dog {}
```

A class can also be a subclass of another class:

```
class Animal {}
class Dog: Animal {}
```

In this example, `Animal` could also be a protocol that `Dog` conforms to.

## Section 20.2: Properties and Methods

Classes can define properties that instances of the class can use. In this example, `Dog` has two properties: `name` and `dogYearAge`:

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

You can access the properties with dot syntax:

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

Classes can also define methods that can be called on the instances, they are declared similar to normal functions, just inside the class:

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

Calling methods also uses dot syntax:

```
dog.bark()
```

## Section 20.3: Reference Semantics

Classes are **reference types**, meaning that multiple variables can refer to the same instance.

```
class Dog {
    var name = ""
}

let firstDog = Dog()
firstDog.name = "Fido"
```

```
let otherDog = firstDog  // otherDog points to
```

same Dog instance otherDog.name = "Rover" // modifying otherDog **also modifies firstDog** print(firstDog.name) // prints "Rover"

Because classes are reference types, even if the class is a constant, its variable properties can still be modified.

```
class Dog {
    var name: String // name is a variable property.
    let age: Int // age is a constant property.
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let constantDog = Dog(name: "Rover", age: 5)// This instance is a constant.
var variableDog = Dog(name: "Spot", age 7)// This instance is a variable.

constantDog.name = "Fido" // Not an error because name is a variable property.
constantDog.age = 6 // Error because age is a constant property.
constantDog = Dog(name: "Fido", age: 6)
/* The last one is an error because you are changing the actual reference, not
just what the reference points to. */

variableDog.name = "Ace" // Not an error because name is a variable property.
variableDog.age = 8 // Error because age is a constant property.
variableDog = Dog(name: "Ace", age: 8)
/* The last one is not an error because variableDog is a variable instance and
therefore the actual reference can be changed. */
```

Test whether two objects are *identical* (point to the exact same instance) using `===`:

```
class Dog: Equatable {
    let name: String
    init(name: String) { self.name = name }
}

// Consider two dogs equal if their names are equal.
func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.name == rhs.name
}

// Create two Dog instances which have the same name.
let spot1 = Dog(name: "Spot")
let spot2 = Dog(name: "Spot")

spot1 == spot2   // true, because the dogs are equal
spot1 != spot2   // false

spot1 === spot2  // false, because the dogs are different instances
spot1 !== spot2  // true
```

# Section 20.4: Classes and Multiple Inheritance

Swift does not support multiple inheritance. That is, you cannot inherit from more than one class.

```
class Animal { ... }
class Pet { ... }
```

```
class Dog: Animal, Pet { ... } // This will result in a compiler error.
```

Instead you are encouraged to use composition when creating your types. This can be accomplished by using protocols.

## Section 20.5: deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // initialize timer
    }

    deinit {
        // code
        timer.invalidate()
    }
}
```