Inheritance

A class can *inherit* methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a *subclass*, and the class it inherits from is known as its *superclass*. Inheritance is a fundamental behavior that differentiates classes from other types in Swift.

Classes in Swift can call and access methods, properties, and subscripts belonging to their superclass and can provide their own overriding versions of those methods, properties, and subscripts to refine or modify their behavior. Swift helps to ensure your overrides are correct by checking that the override definition has a matching superclass definition.

Classes can also add property observers to inherited properties in order to be notified when the value of a property changes. Property observers can be added to any property, regardless of whether it was originally defined as a stored or computed property.

Defining a Base Class

Any class that does not inherit from another class is known as a *base class*.

> NOTE
>
> Swift classes do not inherit from a universal base class. Classes you define without specifying a superclass automatically become base classes for you to build upon.

The example below defines a base class called `Vehicle`. This base class defines a stored property called `currentSpeed`, with a default value of `0.0` (inferring a property type of `Double`). The `currentSpeed` property's value is used by a read-only computed `String` property called `description` to create a description of the vehicle.

The `Vehicle` base class also defines a method called `makeNoise`. This method does not actually do anything for a base `Vehicle` instance, but will be customized by subclasses of `Vehicle` later on:

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necessarily make a noise
    }
}
```

You create a new instance of `Vehicle` with *initializer syntax*, which is written as a `TypeName` followed by empty parentheses:

```
let someVehicle = Vehicle()
```

Having created a new `Vehicle` instance, you can access its `description` property to print a human-readable description of the vehicle's current speed:

```
print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

The `Vehicle` class defines common characteristics for an arbitrary vehicle, but is not much use in itself. To make it more useful, you need to refine it to describe more specific kinds of vehicles.

Subclassing

Subclassing is the act of basing a new class on an existing class. The subclass inherits characteristics from the existing class, which you can then refine. You can also add new characteristics to the subclass.

To indicate that a subclass has a superclass, write the subclass name before the superclass name, separated by a colon:

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

The following example defines a subclass called `Bicycle`, with a superclass of `Vehicle`:

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

The new `Bicycle` class automatically gains all of the characteristics of `Vehicle`, such as its `currentSpeed` and `description` properties and its `makeNoise()` method.

In addition to the characteristics it inherits, the `Bicycle` class defines a new stored property, `hasBasket`, with a default value of `false` (inferring a type of `Bool` for the property).

By default, any new `Bicycle` instance you create will not have a basket. You can set the `hasBasket` property to `true` for a particular `Bicycle` instance after that instance is created:

```
let bicycle = Bicycle()
bicycle.hasBasket = true
```

You can also modify the inherited `currentSpeed` property of a `Bicycle` instance, and query the instance's inherited `description` property:

```
bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

Subclasses can themselves be subclassed. The next example creates a subclass of `Bicycle` for a two-seater bicycle known as a "tandem":

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

`Tandem` inherits all of the properties and methods from `Bicycle`, which in turn inherits all of the properties and methods from `Vehicle`. The `Tandem` subclass also adds a new stored property called `currentNumberOfPassengers`, with a default value of `0`.

If you create an instance of `Tandem`, you can work with any of its new and inherited properties, and query the read-only `description` property it inherits from `Vehicle`:

```
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

Overriding

A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as *overriding*.

To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the `override` keyword. Doing so clarifies that you intend to provide an override and have not provided a matching definition by mistake. Overriding by accident can cause unexpected behavior, and any overrides without the `override` keyword are diagnosed as an error when your code is compiled.

The `override` keyword also prompts the Swift compiler to check that your overriding class's superclass (or one of its parents) has a declaration that matches the one you provided for the override. This check ensures that your overriding definition is correct.

Accessing Superclass Methods, Properties, and Subscripts

When you provide a method, property, or subscript override for a subclass, it is sometimes useful to use the existing superclass implementation as part of your override. For example, you can refine the behavior of that existing implementation, or store a modified value in an existing inherited variable.

Where this is appropriate, you access the superclass version of a method, property, or subscript by using the `super` prefix:

- An overridden method named `someMethod()` can call the superclass version of `someMethod()` by calling `super.someMethod()` within the overriding method implementation.
- An overridden property called `someProperty` can access the superclass version of `someProperty` as `super.someProperty` within the overriding getter or setter implementation.
- An overridden subscript for `someIndex` can access the superclass version of the same subscript as `super[someIndex]` from within the overriding subscript implementation.

Overriding Methods

You can override an inherited instance or type method to provide a tailored or alternative implementation of the method within your subclass.

The following example defines a new subclass of `Vehicle` called `Train`, which overrides the `makeNoise()` method that `Train` inherits from `Vehicle`:

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

If you create a new instance of `Train` and call its `makeNoise()` method, you can see that the `Train` subclass version of the method is called:

```
let train = Train()
train.makeNoise()
// Prints "Choo Choo"
```

Overriding Properties

You can override an inherited instance or type property to provide your own custom getter and setter for that property, or to add property observers to enable the overriding property to observe when the underlying property value changes.

Overriding Property Getters and Setters

You can provide a custom getter (and setter, if appropriate) to override *any* inherited property, regardless of whether the inherited property is implemented as a stored or computed property at source. The stored or computed nature of an inherited property is not known by a subclass—it only knows that the inherited property has a certain name and type. You must always state both the name and the type of the property you are overriding, to enable the compiler to check that your override matches a superclass property with the same name and type.

You can present an inherited read-only property as a read-write property by providing both a getter and a

setter in your subclass property override. You cannot, however, present an inherited read-write property as a read-only property.

> NOTE
>
> If you provide a setter as part of a property override, you must also provide a getter for that override. If you don't want to modify the inherited property's value within the overriding getter, you can simply pass through the inherited value by returning `super.someProperty` from the getter, where `someProperty` is the name of the property you are overriding.

The following example defines a new class called `Car`, which is a subclass of `Vehicle`. The `Car` class introduces a new stored property called `gear`, with a default integer value of `1`. The `Car` class also overrides the `description` property it inherits from `Vehicle`, to provide a custom description that includes the current gear:

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
```

The override of the `description` property starts by calling `super.description`, which returns the `Vehicle` class's `description` property. The `Car` class's version of `description` then adds some extra text onto the end of this description to provide information about the current gear.

If you create an instance of the `Car` class and set its `gear` and `currentSpeed` properties, you can see that its `description` property returns the tailored description defined within the `Car` class:

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

Overriding Property Observers

You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented. For more information on property observers, see Property Observers.

> NOTE
>
> You cannot add property observers to inherited constant stored properties or inherited read-only computed properties. The value of these properties cannot be set, and so it is not appropriate to provide a `willSet` or `didSet` implementation as part of an override.
>
> Note also that you cannot provide both an overriding setter and an overriding property observer for the same property. If you want to observe changes to a property's value, and you are already providing a custom setter for that property, you can simply observe any value changes from within the custom setter.

The following example defines a new class called `AutomaticCar`, which is a subclass of `Car`. The `AutomaticCar` class represents a car with an automatic gearbox, which automatically selects an appropriate gear to use based on the current speed:

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

Whenever you set the `currentSpeed` property of an `AutomaticCar` instance, the property's `didSet` observer sets the instance's `gear` property to an appropriate choice of gear for the new speed. Specifically, the property observer chooses a gear that is the new `currentSpeed` value divided by `10`, rounded down to the nearest integer, plus `1`. A speed of `35.0` produces a gear of `4`:

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

Preventing Overrides

You can prevent a method, property, or subscript from being overridden by marking it as *final*. Do this by writing the `final` modifier before the method, property, or subscript's introducer keyword (such as `final var`, `final func`, `final class func`, and `final subscript`).

Any attempt to override a final method, property, or subscript in a subclass is reported as a compile-time error. Methods, properties, or subscripts that you add to a class in an extension can also be marked as final within the extension's definition.

You can mark an entire class as final by writing the `final` modifier before the `class` keyword in its class definition (`final class`). Any attempt to subclass a final class is reported as a compile-time error.