```swift
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
    }

    // nested Rank enumeration
    enum Rank: Int {
        case two = 2, three, four, five, six, seven, eight, nine, ten
        case jack, queen, king, ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
            switch self {
            case .ace:
                return Values(first: 1, second: 11)
            case .jack, .queen, .king:
                return Values(first: 10, second: nil)
            default:
                return Values(first: self.rawValue, second: nil)
            }
        }
    }

    // BlackjackCard properties and methods
    let rank: Rank, suit: Suit
    var description: String {
        var output = "suit is \(suit.rawValue),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
            output += " or \(second)"
        }
        return output
    }
}
```

The `Suit` enumeration describes the four common playing card suits, together with a raw `Character` value to represent their symbol.

The `Rank` enumeration describes the thirteen possible playing card ranks, together with a raw `Int` value to represent their face value. (This raw `Int` value is not used for the Jack, Queen, King, and Ace cards.)

As mentioned above, the `Rank` enumeration defines a further nested structure of its own, called `Values`. This structure encapsulates the fact that most cards have one value, but the Ace card has two values. The `Values` structure defines two properties to represent this:

- `first`, of type `Int`
- `second`, of type `Int?`, or "optional `Int`"


```swift
func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Prints "200"

completionHandlers.first?()
print(instance.x)
// Prints "100"
```

Autoclosures

An *autoclosure* is a closure that is automatically created to wrap an expression that's being passed as an argument to a function. It doesn't take any arguments, and when it's called, it returns the value of the expression that's wrapped inside of it. This syntactic convenience lets you omit braces around a function's parameter by writing a normal expression instead of an explicit closure.

It's common to *call* functions that take autoclosures, but it's not common to *implement* that kind of function. For example, the `assert(condition:message:file:line:)` function takes an autoclosure for its `condition` and `message` parameters; its `condition` parameter is evaluated only in debug builds and its `message` parameter is evaluated only if `condition` is `false`.

An autoclosure lets you delay evaluation, because the code inside isn't run until you call the closure. Delaying evaluation is useful for code that has side effects or is computationally expensive, because it lets you control when that code is evaluated. The code below shows how a closure delays evaluation.

```swift
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

Even though the first element of the `customersInLine` array is removed by the code inside the closure, the array element isn't removed until the closure is actually called. If the closure is never called, the expression inside the closure is never evaluated, which means the array element is never removed. Note that the type of `customerProvider` is not `String` but `() -> String`—a function with no parameters that returns a string.

You get the same behavior of delayed evaluation when you pass a closure as an argument to a function.

```swift
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// Prints "Now serving Alex!"
```

The `serve(customer:)` function in the listing above takes an explicit closure that returns a customer's name. The version of `serve(customer:)` below performs the same operation but, instead of taking an explicit closure, it takes an autoclosure by marking its parameter's type with the `@autoclosure` attribute. Now you can call the function as if it took a `String` argument instead of a closure. The argument is automatically converted to a closure, because the `customerProvider` parameter's type is marked with the `@autoclosure` attribute.

```swift
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Prints "Now serving Ewa!"
```

> NOTE
>
> Overusing autoclosures can make your code hard to understand. The context and function name should make it clear that evaluation is being deferred.

If you want an autoclosure that is allowed to escape, use both the `@autoclosure` and `@escaping` attributes. The `@escaping` attribute is described above in Escaping Closures.

Nested Types

Enumerations are often created to support a specific class or structure's functionality. Similarly, it can be convenient to define utility classes and structures purely for use within the context of a more complex type. To accomplish this, Swift enables you to define *nested types*, whereby you nest supporting enumerations, classes, and structures within the definition of the type they support.

To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required.

Nested Types in Action

The example below defines a structure called `BlackjackCard`, which models a playing card as used in the game of Blackjack. The `BlackjackCard` structure contains two nested enumeration types called `Suit` and `Rank`.

In Blackjack, the Ace cards have a value of either one or eleven. This feature is represented by a structure called `Values`, which is nested within the `Rank` enumeration:

```swift
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
    }
}

// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called Ghostbusters, dir. Ivan Reitman
// Hello, Michael
```

NOTE

The `Any` type represents values of any type, including optional types. Swift gives you a warning if you use an optional value where a value of type `Any` is expected. If you really do need to use an optional value as an `Any` value, you can use the `as` operator to explicitly cast the optional to `Any`, as shown below.

```swift
let optionalNumber: Int? = 3
things.append(optionalNumber)        // Warning
things.append(optionalNumber as Any) // No warning
```



```swift
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping
  () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!"
```

In the code above, instead of calling the closure passed to it as its `customerProvider` argument, the `collectCustomerProviders(_:)` function appends the closure to the `customerProviders` array. The array is declared outside the scope of the function, which means the closures in the array can be executed after the function returns. As a result, the value of the `customerProvider` argument must be allowed to escape the function's scope.

Enumerations

An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

If you are familiar with C, you will know that C enumerations assign related names to a set of integer values. Enumerations in Swift are much more flexible, and do not have to provide a value for each case of the enumeration. If a value (known as a "raw" value) *is* provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type.

Alternatively, enumeration cases can specify associated values of *any* type to be stored along with each different case value, much as unions or variants do in other languages. You can define a common set of related cases as part of one enumeration, each of which has a different set of values of appropriate types associated with it.

Enumerations in Swift are first-class types in their own right. They adopt many features traditionally supported only by classes, such as computed properties to provide additional information about the enumeration's current value, and instance methods to provide functionality related to the values the enumeration represents. Enumerations can also define initializers to provide an initial case value; can be extended to expand their functionality beyond their original implementation; and can conform to protocols to provide standard functionality.

For more about these capabilities, see Properties, Methods, Initialization, Extensions, and Protocols.

Enumeration Syntax

You introduce enumerations with the `enum` keyword and place their entire definition within a pair of braces:

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

Here's an example for the four main points of a compass:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

The values defined in an enumeration (such as `north`, `south`, `east`, and `west`) are its *enumeration cases*. You use the `case` keyword to introduce new enumeration cases.

NOTE


```
var things = [Any]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

The `things` array contains two `Int` values, two `Double` values, a `String` value, a tuple of type `(Double, Double)`, the movie "Ghostbusters", and a closure expression that takes a `String` value and returns another `String` value.

To discover the specific type of a constant or variable that is known only to be of type `Any` or `AnyObject`, you can use an `is` or `as` pattern in a `switch` statement's cases. The example below iterates over the items in the `things` array and queries the type of each item with a `switch` statement. Several of the `switch` statement's cases bind their matched value to a constant of the specified type to enable its value to be printed:

```swift
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

// Movie: Casablanca, dir. Michael Curtiz
// Song: Blue Suede Shoes, by Elvis Presley
// Movie: Citizen Kane, dir. Orson Welles
// Song: The One And Only, by Chesney Hawkes
// Song: Never Gonna Give You Up, by Rick Astley
```

The example starts by trying to downcast the current `item` as a `Movie`. Because `item` is a `MediaItem` instance, it's possible that it *might* be a `Movie`; equally, it's also possible that it might be a `Song`, or even just a base `MediaItem`. Because of this uncertainty, the `as?` form of the type cast operator returns an *optional* value when attempting to downcast to a subclass type. The result of `item as? Movie` is of type `Movie?`, or "optional `Movie`".

Downcasting to `Movie` fails when applied to the `Song` instances in the library array. To cope with this, the example above uses optional binding to check whether the optional `Movie` actually contains a value (that is, to find out whether the downcast succeeded.) This optional binding is written "`if let movie = item as? Movie`", which can be read as:

"Try to access `item` as a `Movie`. If this is successful, set a new temporary constant called `movie` to the value stored in the returned optional `Movie`."

If the downcasting succeeds, the properties of `movie` are then used to print a description for that `Movie` instance, including the name of its `director`. A similar principle is used to check for `Song` instances, and to print an appropriate description (including `artist` name) whenever a `Song` is found in the library.

> NOTE
>
> Casting does not actually modify the instance or change its values. The underlying instance remains the same; it is simply treated and accessed as an instance of the type to which it has been cast.

Type Casting for Any and AnyObject

Swift provides two special types for working with nonspecific types:

- `Any` can represent an instance of any type at all, including function types.
- `AnyObject` can represent an instance of any class type.

Use `Any` and `AnyObject` only when you explicitly need the behavior and capabilities they provide. It is always better to be specific about the types you expect to work with in your code.

Here's an example of using `Any` to work with a mix of different types, including function types and nonclass types. The example creates an array called `things`, which can store values of type `Any`:

> Unlike C and Objective-C, Swift enumeration cases are not assigned a default integer value when they are created. In the `CompassPoint` example above, `north`, `south`, `east` and `west` do not implicitly equal `0`, `1`, `2` and `3`. Instead, the different enumeration cases are fully-fledged values in their own right, with an explicitly defined type of `CompassPoint`.

Multiple cases can appear on a single line, separated by commas:

```swift
enum Planet {
    case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

Each enumeration definition defines a new type. Like other types in Swift, their names (such as `CompassPoint` and `Planet`) should start with a capital letter. Give enumeration types singular rather than plural names, so that they read as self-evident:

```swift
var directionToHead = CompassPoint.west
```

The type of `directionToHead` is inferred when it is initialized with one of the possible values of `CompassPoint`. Once `directionToHead` is declared as a `CompassPoint`, you can set it to a different `CompassPoint` value using a shorter dot syntax:

```swift
directionToHead = .east
```

The type of `directionToHead` is already known, and so you can drop the type when setting its value. This makes for highly readable code when working with explicitly typed enumeration values.

Matching Enumeration Values with a Switch Statement

You can match individual enumeration values with a `switch` statement:

```swift
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Prints "Watch out for penguins"
```

You can read this code as:

"Consider the value of `directionToHead`. In the case where it equals `.north`, print `"Lots of planets have a north"`. In the case where it equals `.south`, print `"Watch out for penguins"`."

...and so on.

As described in Control Flow, a `switch` statement must be exhaustive when considering an enumeration's cases. If the `case` for `.west` is omitted, this code does not compile, because it does not consider the complete list of `CompassPoint` cases. Requiring exhaustiveness ensures that enumeration cases are not accidentally omitted.

When it is not appropriate to provide a `case` for every enumeration case, you can provide a `default` case to cover any cases that are not addressed explicitly:

```
let somePlanet = Planet.earth
switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// Prints "Mostly harmless"
```

Iterating over Enumeration Cases

For some enumerations, it's useful to have a collection of all of that enumeration's cases. You enable this by writing `: CaseIterable` after the enumeration's name. Swift exposes a collection of all the cases as an `allCases` property of the enumeration type. Here's an example:

```
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
// Prints "3 beverages available"
```

In the example above, you write `Beverage.allCases` to access a collection that contains all of the cases of the `Beverage` enumeration. You can use `allCases` like any other collection—the collection's elements are instances of the enumeration type, so in this case they're `Beverage` values. The example above counts how many cases there are, and the example below uses a `for` loop to iterate over all the cases.

```
for beverage in Beverage.allCases {
    print(beverage)
}
// coffee
// tea
// juice
```

The syntax used in the examples above marks the enumeration as conforming to the `CaseIterable` protocol. For information about protocols, see Protocols.

Associated Values


```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \(movieCount) movies and \(songCount)
  songs")
// Prints "Media library contains 2 movies and 3 songs"
```

This example iterates through all items in the `library` array. On each pass, the `for-in` loop sets the `item` constant to the next `MediaItem` in the array.

`item is Movie` returns `true` if the current `MediaItem` is a `Movie` instance and `false` if it is not. Similarly, `item is Song` checks whether the item is a `Song` instance. At the end of the `for-in` loop, the values of `movieCount` and `songCount` contain a count of how many `MediaItem` instances were found of each type.

Downcasting

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to *downcast* to the subclass type with a *type cast operator* (`as?` or `as!`).

Because downcasting can fail, the type cast operator comes in two different forms. The conditional form, `as?`, returns an optional value of the type you are trying to downcast to. The forced form, `as!`, attempts the downcast and force-unwraps the result as a single compound action.

Use the conditional form of the type cast operator (`as?`) when you are not sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be `nil` if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (`as!`) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.

The example below iterates over each `MediaItem` in `library`, and prints an appropriate description for each item. To do this, it needs to access each item as a true `Movie` or `Song`, and not just as a `MediaItem`. This is necessary in order for it to be able to access the `director` or `artist` property of a `Movie` or `Song` for use in the description.

In this example, each item in the array might be a `Movie`, or it might be a `Song`. You don't know in advance which actual class to use for each item, and so it is appropriate to use the conditional form of the type cast operator (`as?`) to check the downcast each time through the loop:

```swift
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

The final snippet creates a constant array called `library`, which contains two `Movie` instances and three `Song` instances. The type of the `library` array is inferred by initializing it with the contents of an array literal. Swift's type checker is able to deduce that `Movie` and `Song` have a common superclass of `MediaItem`, and so it infers a type of `[MediaItem]` for the `library` array:

```swift
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// the type of "library" is inferred to be [MediaItem]
```

The items stored in `library` are still `Movie` and `Song` instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as `MediaItem`, and not as `Movie` or `Song`. In order to work with them as their native type, you need to *check* their type, or *downcast* them to a different type, as described below.

Checking Type

Use the *type check operator* (`is`) to check whether an instance is of a certain subclass type. The type check operator returns `true` if the instance is of that subclass type and `false` if it is not.

The example below defines two variables, `movieCount` and `songCount`, which count the number of `Movie` and `Song` instances in the `library` array:

The examples in the previous section show how the cases of an enumeration are a defined (and typed) value in their own right. You can set a constant or variable to `Planet.earth`, and check for this value later. However, it is sometimes useful to be able to store *associated values* of other types alongside these case values. This enables you to store additional custom information along with the case value, and permits this information to vary each time you use that case in your code.

You can define Swift enumerations to store associated values of any given type, and the value types can be different for each case of the enumeration if needed. Enumerations similar to these are known as *discriminated unions*, *tagged unions*, or *variants* in other programming languages.

For example, suppose an inventory tracking system needs to track products by two different types of barcode. Some products are labeled with 1D barcodes in UPC format, which uses the numbers `0` to `9`. Each barcode has a "number system" digit, followed by five "manufacturer code" digits and five "product code" digits. These are followed by a "check" digit to verify that the code has been scanned correctly:

Other products are labeled with 2D barcodes in QR code format, which can use any ISO 8859-1 character and can encode a string up to 2,953 characters long:

It would be convenient for an inventory tracking system to be able to store UPC barcodes as a tuple of four integers, and QR code barcodes as a string of any length.

In Swift, an enumeration to define product barcodes of either type might look like this:

```swift
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

This can be read as:

"Define an enumeration type called `Barcode`, which can take either a value of `upc` with an associated value of type (`Int`, `Int`, `Int`, `Int`), or a value of `qrCode` with an associated value of type `String`."

This definition does not provide any actual `Int` or `String` values—it just defines the *type* of associated values that `Barcode` constants and variables can store when they are equal to `Barcode.upc` or `Barcode.qrCode`.

New barcodes can then be created using either type:

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

This example creates a new variable called `productBarcode` and assigns it a value of `Barcode.upc` with an associated tuple value of `(8, 85909, 51226, 3)`.

The same product can be assigned a different type of barcode:

```
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

At this point, the original `Barcode.upc` and its integer values are replaced by the new `Barcode.qrCode` and its string value. Constants and variables of type `Barcode` can store either a `.upc` or a `.qrCode` (together with their associated values), but they can only store one of them at any given time.

The different barcode types can be checked using a switch statement, as before. This time, however, the associated values can be extracted as part of the switch statement. You extract each associated value as a constant (with the `let` prefix) or a variable (with the `var` prefix) for use within the `switch` case's body:

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

If all of the associated values for an enumeration case are extracted as constants, or if all are extracted as variables, you can place a single `var` or `let` annotation before the case name, for brevity:

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Raw Values

The barcode example in Associated Values shows how cases of an enumeration can declare that they store associated values of different types. As an alternative to associated values, enumeration cases can come prepopulated with default values (called *raw values*), which are all of the same type.

Here's an example that stores raw ASCII values alongside named enumeration cases:

Type Casting

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the `is` and `as` operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

You can also use type casting to check whether a type conforms to a protocol, as described in Checking for Protocol Conformance.

Defining a Class Hierarchy for Type Casting

You can use type casting with a hierarchy of classes and subclasses to check the type of a particular class instance and to cast that instance to another class within the same hierarchy. The three code snippets below define a hierarchy of classes and an array containing instances of those classes, for use in an example of type casting.

The first snippet defines a new base class called `MediaItem`. This class provides basic functionality for any kind of item that appears in a digital media library. Specifically, it declares a `name` property of type `String`, and an `init name` initializer. (It is assumed that all media items, including all movies and songs, will have a name.)

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

The next snippet defines two subclasses of `MediaItem`. The first subclass, `Movie`, encapsulates additional information about a movie or film. It adds a `director` property on top of the base `MediaItem` class, with a corresponding initializer. The second subclass, `Song`, adds an `artist` property and initializer on top of the base class:

The above example uses a `defer` statement to ensure that the `open(_:)` function has a corresponding call to `close(_:)`.

> NOTE
>
> You can use a `defer` statement even when no error handling code is involved.

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

Here, the raw values for an enumeration called `ASCIIControlCharacter` are defined to be of type `Character`, and are set to some of the more common ASCII control characters. `Character` values are described in Strings and Characters.

Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.

> NOTE
>
> Raw values are *not* the same as associated values. Raw values are set to prepopulated values when you first define the enumeration in your code, like the three ASCII codes above. The raw value for a particular enumeration case is always the same. Associated values are set when you create a new constant or variable based on one of the enumeration's cases, and can be different each time you do so.

Implicitly Assigned Raw Values

When you're working with enumerations that store integer or string raw values, you don't have to explicitly assign a raw value for each case. When you don't, Swift will automatically assign the values for you.

For instance, when integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn't have a value set, its value is `0`.

The enumeration below is a refinement of the earlier `Planet` enumeration, with integer raw values to represent each planet's order from the sun:

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune
}
```

In the example above, `Planet.mercury` has an explicit raw value of `1`, `Planet.venus` has an implicit raw value of `2`, and so on.

When strings are used for raw values, the implicit value for each case is the text of that case's name.

The enumeration below is a refinement of the earlier `CompassPoint` enumeration, with string raw values to represent each direction's name:

```
enum CompassPoint: String {
    case north, south, east, west
}
```

In the example above, `CompassPoint.south` has an implicit raw value of `"south"`, and so on.

You access the raw value of an enumeration case with its `rawValue` property:

```swift
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

Initializing from a Raw Value

If you define an enumeration with a raw-value type, the enumeration automatically receives an initializer that takes a value of the raw value's type (as a parameter called `rawValue`) and returns either an enumeration case or `nil`. You can use this initializer to try to create a new instance of the enumeration.

This example identifies Uranus from its raw value of 7:

```swift
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals Planet.uranus
```

Not all possible `Int` values will find a matching planet, however. Because of this, the raw value initializer always returns an *optional* enumeration case. In the example above, `possiblePlanet` is of type `Planet?`, or "optional `Planet`."

> NOTE
>
> The raw value initializer is a failable initializer, because not every raw value will return an enumeration case. For more information, see Failable Initializers.

If you try to find a planet with a position of 11, the optional `Planet` value returned by the raw value initializer will be `nil`:

```swift
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
    switch somePlanet {
    case .earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")
}
// Prints "There isn't a planet at position 11"
```

This example uses optional binding to try to access a planet with a raw value of 11. The statement `if let somePlanet = Planet(rawValue: 11)` creates an optional `Planet`, and sets `somePlanet` to the value of that optional `Planet` if it can be retrieved. In this case, it is not possible to retrieve a planet with a position of 11, and so the `else` branch is executed instead.

Recursive Enumerations


```swift
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk() { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

Disabling Error Propagation

Sometimes you know a throwing function or method won't, in fact, throw an error at runtime. On those occasions, you can write `try!` before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

For example, the following code uses a `loadImage(atPath:)` function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

```swift
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Specifying Cleanup Actions

You use a `defer` statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any necessary cleanup that should be performed regardless of *how* execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as `return` or `break`. For example, you can use a `defer` statement to ensure that file descriptors are closed and manually allocated memory is freed.

A `defer` statement defers execution until the current scope is exited. This statement consists of the `defer` keyword and the statements to be executed later. The deferred statements may not contain any code that would transfer control out of the statements, such as a `break` or a `return` statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they're written in your source code. That is, the code in the first `defer` statement executes last, the code in the second `defer` statement executes second to last, and so on. The last `defer` statement in source code order executes first.

```swift
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

```
func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Invalid selection, out of stock, or not enough money.")
    }
}

do {
    try nourish(with: "Beet-Flavored Chips")
} catch {
    print("Unexpected non-vending-machine-related error: \(error)")
}
// Prints "Invalid selection, out of stock, or not enough money."
```

In the `nourish(with:)` function, if `vend(itemNamed:)` throws an error that's one of the cases of the `VendingMachineError` enumeration, `nourish(with:)` handles the error by printing a message. Otherwise, `nourish(with:)` propagates the error to its call site. The error is then caught by the general `catch` clause.

Converting Errors to Optional Values

You use `try?` to handle an error by converting it to an optional value. If an error is thrown while evaluating the `try?` expression, the value of the expression is `nil`. For example, in the following code `x` and `y` have the same value and behavior:

```
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

If `someThrowingFunction()` throws an error, the value of `x` and `y` is `nil`. Otherwise, the value of `x` and `y` is the value that the function returned. Note that `x` and `y` are an optional of whatever type `someThrowingFunction()` returns. Here the function returns an integer, so `x` and `y` are optional integers.

Using `try?` lets you write concise error handling code when you want to handle all errors in the same way. For example, the following code uses several approaches to fetch data, or returns `nil` if all of the approaches fail.

A *recursive enumeration* is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing `indirect` before it, which tells the compiler to insert the necessary layer of indirection.

For example, here is an enumeration that stores simple arithmetic expressions:

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression, ArithmeticExpression)
    indirect case multiplication(ArithmeticExpression,
  ArithmeticExpression)
}
```

You can also write `indirect` before the beginning of the enumeration to enable indirection for all of the enumeration's cases that have an associated value:

```
indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression, ArithmeticExpression)
    case multiplication(ArithmeticExpression, ArithmeticExpression)
}
```

This enumeration can store three kinds of arithmetic expressions: a plain number, the addition of two expressions, and the multiplication of two expressions. The `addition` and `multiplication` cases have associated values that are also arithmetic expressions—these associated values make it possible to nest expressions. For example, the expression `(5 + 4) * 2` has a number on the right-hand side of the multiplication and another expression on the left-hand side of the multiplication. Because the data is nested, the enumeration used to store the data also needs to support nesting—this means the enumeration needs to be recursive. The code below shows the `ArithmeticExpression` recursive enumeration being created for `(5 + 4) * 2`:

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum,
  ArithmeticExpression.number(2))
```

A recursive function is a straightforward way to work with data that has a recursive structure. For example, here's a function that evaluates an arithmetic expression:

```swift
func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// Prints "18"
```

This function evaluates a plain number by simply returning the associated value. It evaluates an addition or multiplication by evaluating the expression on the left-hand side, evaluating the expression on the right-hand side, and then adding them or multiplying them.

You write a pattern after `catch` to indicate what errors that clause can handle. If a `catch` clause doesn't have a pattern, the clause matches any error and binds the error to a local constant named `error`. For more information about pattern matching, see Patterns.

For example, the following code matches against all three cases of the `VendingMachineError` enumeration.

```swift
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded)
  coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

In the above example, the `buyFavoriteSnack(person:vendingMachine:)` function is called in a `try` expression, because it can throw an error. If an error is thrown, execution immediately transfers to the `catch` clauses, which decide whether to allow propagation to continue. If no pattern is matched, the error gets caught by the final `catch` clause and is bound to a local `error` constant. If no error is thrown, the remaining statements in the `do` statement are executed.

The `catch` clauses don't have to handle every possible error that the code in the `do` clause can throw. If none of the `catch` clauses handle the error, the error propagates to the surrounding scope. However, the propagated error must be handled by *some* surrounding scope. In a nonthrowing function, an enclosing `do-catch` clause must handle the error. In a throwing function, either an enclosing `do-catch` clause or the caller must handle the error. If the error propagates to the top-level scope without being handled, you'll get a runtime error.

For example, the above example can be written so any error that isn't a `VendingMachineError` is instead caught by the calling function:

```swift
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine)
  throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

In this example, the `buyFavoriteSnack(person: vendingMachine:)` function looks up a given person's favorite snack and tries to buy it for them by calling the `vend(itemNamed:)` method. Because the `vend(itemNamed:)` method can throw an error, it's called with the `try` keyword in front of it.

Throwing initializers can propagate errors in the same way as throwing functions. For example, the initializer for the `PurchasedSnack` structure in the listing below calls a throwing function as part of the initialization process, and it handles any errors that it encounters by propagating them to its caller.

```swift
struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine: VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}
```

Handling Errors Using Do-Catch

You use a `do-catch` statement to handle errors by running a block of code. If an error is thrown by the code in the `do` clause, it is matched against the `catch` clauses to determine which one of them can handle the error.

Here is the general form of a `do-catch` statement:

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
} catch {
    statements
}
```


Structures and Classes

Structures and *classes* are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.

Unlike other programming languages, Swift doesn't require you to create separate interface and implementation files for custom structures and classes. In Swift, you define a structure or class in a single file, and the external interface to that class or structure is automatically made available for other code to use.

> NOTE
>
> An instance of a class is traditionally known as an *object*. However, Swift structures and classes are much closer in functionality than in other languages, and much of this chapter describes functionality that applies to instances of *either* a class or a structure type. Because of this, the more general term *instance* is used.

Comparing Structures and Classes

Structures and classes in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

For more information, see Properties, Methods, Subscripts, Initialization, Extensions, and Protocols.

Classes have additional capabilities that structures don't have:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

For more information, see Inheritance, Type Casting, Deinitialization, and Automatic Reference Counting.

The additional capabilities that classes support come at the cost of increased complexity. As a general guideline, prefer structures and enumerations because they're easier to reason about, and use classes when they're appropriate or necessary. In practice, this means most of the custom data types you define will be structures and enumerations. For a more detailed comparison, see Choosing Between Structures and Classes.

Definition Syntax

Structures and classes have a similar definition syntax. You introduce structures with the `struct` keyword and classes with the `class` keyword. Both place their entire definition within a pair of braces:

```
struct SomeStructure {
    // structure definition goes here
}
class SomeClass {
    // class definition goes here
}
```

> NOTE
>
> Whenever you define a new structure or class, you define a new Swift type. Give types `UpperCamelCase` names (such as `SomeStructure` and `SomeClass` here) to match the capitalization of standard Swift types (such as `String`, `Int`, and `Bool`). Give properties and methods `lowerCamelCase` names (such as `frameRate` and `incrementCount`) to differentiate them from type names.

Here's an example of a structure definition and a class definition:

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

The example above defines a new structure called `Resolution`, to describe a pixel-based display resolution. This structure has two stored properties called `width` and `height`. Stored properties are constants or variables that are bundled up and stored as part of the structure or class. These two properties are inferred to be of type `Int` by setting them to an initial integer value of `0`.

The example above also defines a new class called `VideoMode`, to describe a specific video mode for video display. This class has four variable stored properties. The first, `resolution`, is initialized with a new `Resolution` structure instance, which infers a property type of `Resolution`. For the other three properties, new `VideoMode` instances will be initialized with an `interlaced` setting of `false` (meaning "noninterlaced video"), a playback frame rate of `0.0`, and an optional `String` value called `name`. The `name` property is automatically given a default value of `nil`, or "no `name` value", because it's of an optional type.

Structure and Class Instances

The `Resolution` structure definition and the `VideoMode` class definition only describe what a `Resolution` or `VideoMode` will look like. They themselves don't describe a specific resolution or video mode. To do that, you need to create an instance of the structure or class.

The syntax for creating instances is very similar for both structures and classes:


```
struct Item {
    var price: Int
    var count: Int
}

class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0

    func vend(itemNamed name: String) throws {
        guard let item = inventory[name] else {
            throw VendingMachineError.invalidSelection
        }

        guard item.count > 0 else {
            throw VendingMachineError.outOfStock
        }

        guard item.price <= coinsDeposited else {
            throw VendingMachineError.insufficientFunds(coinsNeeded:
  item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}
```

The implementation of the `vend(itemNamed:)` method uses `guard` statements to exit the method early and throw appropriate errors if any of the requirements for purchasing a snack aren't met. Because a `throw` statement immediately transfers program control, an item will be vended only if all of these requirements are met.

Because the `vend(itemNamed:)` method propagates any errors it throws, any code that calls this method must either handle the errors—using a `do-catch` statement, `try?`, or `try!`—or continue to propagate them. For example, the `buyFavoriteSnack(person:vendingMachine:)` in the example below is also a throwing function, and any errors that the `vend(itemNamed:)` method throws will propagate up to the point where the `buyFavoriteSnack(person:vendingMachine:)` function is called.

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a `do-catch` statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

When a function throws an error, it changes the flow of your program, so it's important that you can quickly identify places in your code that can throw errors. To identify these places in your code, write the `try` keyword—or the `try?` or `try!` variation—before a piece of code that calls a function, method, or initializer that can throw an error. These keywords are described in the sections below.

> NOTE
>
> Error handling in Swift resembles exception handling in other languages, with the use of the `try`, `catch` and `throw` keywords. Unlike exception handling in many languages—including Objective-C—error handling in Swift does not involve unwinding the call stack, a process that can be computationally expensive. As such, the performance characteristics of a `throw` statement are comparable to those of a `return` statement.

Propagating Errors Using Throwing Functions

To indicate that a function, method, or initializer can throw an error, you write the `throws` keyword in the function's declaration after its parameters. A function marked with `throws` is called a *throwing function*. If the function specifies a return type, you write the `throws` keyword before the return arrow (`->`).

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

A throwing function propagates errors that are thrown inside of it to the scope from which it's called.

> NOTE
>
> Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

In the example below, the `VendingMachine` class has a `vend(itemNamed:)` method that throws an appropriate `VendingMachineError` if the requested item is not available, is out of stock, or has a cost that exceeds the current deposited amount:


```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Structures and classes both use initializer syntax for new instances. The simplest form of initializer syntax uses the type name of the class or structure followed by empty parentheses, such as `Resolution()` or `VideoMode()`. This creates a new instance of the class or structure, with any properties initialized to their default values. Class and structure initialization is described in more detail in Initialization.

Accessing Properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (`.`), without any spaces:

```
print("The width of someResolution is \(someResolution.width)")
// Prints "The width of someResolution is 0"
```

In this example, `someResolution.width` refers to the `width` property of `someResolution`, and returns its default initial value of `0`.

You can drill down into subproperties, such as the `width` property in the `resolution` property of a `VideoMode`:

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is 0"
```

You can also use dot syntax to assign a new value to a variable property:

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \
  (someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is now 1280"
```

Memberwise Initializers for Structure Types

All structures have an automatically generated *memberwise initializer*, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:

```
let vga = Resolution(width: 640, height: 480)
```

Unlike structures, class instances don't receive a default memberwise initializer. Initializers are described in more detail in Initialization.

Structures and Enumerations Are Value Types

A *value type* is a type whose value is *copied* when it's assigned to a variable or constant, or when it's passed to a function.

You've actually been using value types extensively throughout the previous chapters. In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays and dictionaries—are value types, and are implemented as structures behind the scenes.

All structures and enumerations are value types in Swift. This means that any structure and enumeration instances you create—and any value types they have as properties—are always copied when they are passed around in your code.

> NOTE
>
> Collections defined by the standard library like arrays, dictionaries, and strings use an optimization to reduce the performance cost of copying. Instead of making a copy immediately, these collections share the memory where the elements are stored between the original instance and any copies. If one of the copies of the collection is modified, the elements are copied just before the modification. The behavior you see in your code is always as if a copy took place immediately.

Consider this example, which uses the `Resolution` structure from the previous example:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

This example declares a constant called `hd` and sets it to a `Resolution` instance initialized with the width and height of full HD video (1920 pixels wide by 1080 pixels high).

It then declares a variable called `cinema` and sets it to the current value of `hd`. Because `Resolution` is a structure, a *copy* of the existing instance is made, and this new copy is assigned to `cinema`. Even though `hd` and `cinema` now have the same width and height, they are two completely different instances behind the scenes.

Next, the `width` property of `cinema` is amended to be the width of the slightly wider 2K standard used for digital cinema projection (2048 pixels wide and 1080 pixels high):

```
cinema.width = 2048
```

Checking the `width` property of `cinema` shows that it has indeed changed to be `2048`:

```
print("cinema is now \(cinema.width) pixels wide")
// Prints "cinema is now 2048 pixels wide"
```

However, the `width` property of the original `hd` instance still has the old value of `1920`:

```
print("hd is still \(hd.width) pixels wide")
// Prints "hd is still 1920 pixels wide"
```

When `cinema` was given the current value of `hd`, the *values* stored in `hd` were copied into the new `cinema` instance. The end result is two completely separate instances that contain the same numeric values. However, because they are separate instances, setting the width of `cinema` to `2048` doesn't affect the width stored in `hd`, as shown in the figure below:

Error Handling

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.

Some operations aren't guaranteed to always complete execution or produce a useful output. Optionals are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure, so that your code can respond accordingly.

As an example, consider the task of reading and processing data from a file on disk. There are a number of ways this task can fail, including the file not existing at the specified path, the file not having read permissions, or the file not being encoded in a compatible format. Distinguishing among these different situations allows a program to resolve some errors and to communicate to the user any errors it can't resolve.

> NOTE
>
> Error handling in Swift interoperates with error handling patterns that use the `NSError` class in Cocoa and Objective-C. For more information about this class, see Handling Cocoa Errors in Swift.

Representing and Throwing Errors

In Swift, errors are represented by values of types that conform to the `Error` protocol. This empty protocol indicates that a type can be used for error handling.

Swift enumerations are particularly well suited to modeling a group of related error conditions, with associated values allowing for additional information about the nature of an error to be communicated. For example, here's how you might represent the error conditions of operating a vending machine inside a game:

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a `throw` statement to throw an error. For example, the following code throws an error to indicate that five additional coins are needed by the vending machine:

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

Handling Errors

When an error is thrown, some surrounding piece of code must be responsible for handling the error—for example, by correcting the problem, trying an alternative approach, or informing the user of the failure.

```swift
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence?.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "John's street name is Laurel Street."
```

In this example, the attempt to set the `address` property of `john.residence` will succeed, because the value of `john.residence` currently contains a valid `Address` instance.

Chaining on Methods with Optional Return Values

The previous example shows how to retrieve the value of a property of optional type through optional chaining. You can also use optional chaining to call a method that returns a value of optional type, and to chain on that method's return value if needed.

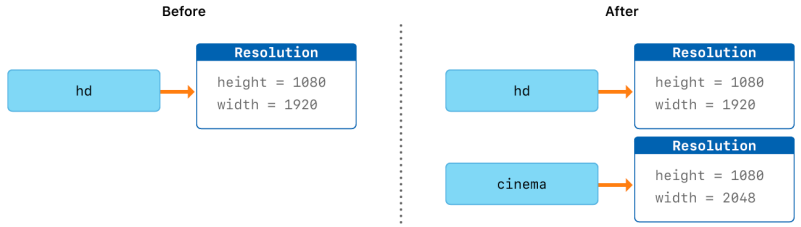
The example below calls the `Address` class's `buildingIdentifier()` method through optional chaining. This method returns a value of type `String?`. As described above, the ultimate return type of this method call after optional chaining is also `String?`:

```swift
if let buildingIdentifier = john.residence?.address?.buildingIdentifier()
  {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
```

If you want to perform further optional chaining on this method's return value, place the optional chaining question mark *after* the method's parentheses:

```swift
if let beginsWithThe =
    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")
    } else {
        print("John's building identifier does not begin with \"The\".")
    }
}
// Prints "John's building identifier begins with "The"."
```

> NOTE
>
> In the example above, you place the optional chaining question mark *after* the parentheses, because the optional value you are chaining on is the `buildingIdentifier()` method's return value, and not the `buildingIdentifier()` method itself.



The same behavior applies to enumerations:

```swift
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
        self = .north
    }
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()

print("The current direction is \(currentDirection)")
print("The remembered direction is \(rememberedDirection)")
// Prints "The current direction is north"
// Prints "The remembered direction is west"
```

When `rememberedDirection` is assigned the value of `currentDirection`, it's actually set to a copy of that value. Changing the value of `currentDirection` thereafter doesn't affect the copy of the original value that was stored in `rememberedDirection`.

Classes Are Reference Types

Unlike value types, *reference types* are *not* copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used.

Here's an example, using the `VideoMode` class defined above:

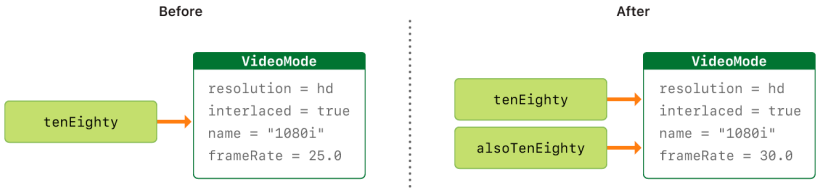
```swift
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

This example declares a new constant called `tenEighty` and sets it to refer to a new instance of the `VideoMode` class. The video mode is assigned a copy of the HD resolution of `1920` by `1080` from before. It's set to be interlaced, and is given a name of `"1080i"`. Finally, it's set to a frame rate of `25.0` frames per second.

Next, `tenEighty` is assigned to a new constant, called `alsoTenEighty`, and the frame rate of `alsoTenEighty` is modified:

```swift
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Because classes are reference types, `tenEighty` and `alsoTenEighty` actually both refer to the *same* `VideoMode` instance. Effectively, they are just two different names for the same single instance, as shown in the figure below:



Checking the `frameRate` property of `tenEighty` shows that it correctly reports the new frame rate of `30.0` from the underlying `VideoMode` instance:

```swift
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// Prints "The frameRate property of tenEighty is now 30.0"
```

This example also shows how reference types can be harder to reason about. If `tenEighty` and `alsoTenEighty` were far apart in your program's code, it could be difficult to find all the ways that the video mode is changed. Wherever you use `tenEighty`, you also have to think about the code that uses `alsoTenEighty`, and vice versa. In contrast, value types are easier to reason about because all of the code that interacts with the same value is close together in your source files.

Note that `tenEighty` and `alsoTenEighty` are declared as *constants*, rather than variables. However, you can still change `tenEighty.frameRate` and `alsoTenEighty.frameRate` because the values of the `tenEighty` and `alsoTenEighty` constants themselves don't actually change. `tenEighty` and `alsoTenEighty` themselves don't "store" the `VideoMode` instance—instead, they both *refer* to a `VideoMode` instance behind the scenes. It's the `frameRate` property of the underlying `VideoMode` that is changed, not the values of the constant references to that `VideoMode`.

Identity Operators

Because classes are reference types, it's possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same isn't true for structures and enumerations, because they are always copied when they are assigned to a constant or variable, or passed to a function.)

It can sometimes be useful to find out whether two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

- Identical to (`===`)
- Not identical to (`!==`)

Use these operators to check whether two constants or variables refer to the same single instance:

The example above defines a dictionary called `testScores`, which contains two key-value pairs that map a `String` key to an array of `Int` values. The example uses optional chaining to set the first item in the `"Dave"` array to `91`; to increment the first item in the `"Bev"` array by `1`; and to try to set the first item in an array for a key of `"Brian"`. The first two calls succeed, because the `testScores` dictionary contains keys for `"Dave"` and `"Bev"`. The third call fails, because the `testScores` dictionary does not contain a key for `"Brian"`.

Linking Multiple Levels of Chaining

You can link together multiple levels of optional chaining to drill down to properties, methods, and subscripts deeper within a model. However, multiple levels of optional chaining do not add more levels of optionality to the returned value.

To put it another way:

- If the type you are trying to retrieve is not optional, it will become optional because of the optional chaining.
- If the type you are trying to retrieve is *already* optional, it will not become *more* optional because of the chaining.

Therefore:

- If you try to retrieve an `Int` value through optional chaining, an `Int?` is always returned, no matter how many levels of chaining are used.
- Similarly, if you try to retrieve an `Int?` value through optional chaining, an `Int?` is always returned, no matter how many levels of chaining are used.

The example below tries to access the `street` property of the `address` property of the `residence` property of `john`. There are *two* levels of optional chaining in use here, to chain through the `residence` and `address` properties, both of which are of optional type:

```swift
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

The value of `john.residence` currently contains a valid `Residence` instance. However, the value of `john.residence.address` is currently `nil`. Because of this, the call to `john.residence?.address?.street` fails.

Note that in the example above, you are trying to retrieve the value of the `street` property. The type of this property is `String?`. The return value of `john.residence?.address?.street` is therefore also `String?`, even though two levels of optional chaining are applied in addition to the underlying optional type of the property.

If you set an actual `Address` instance as the value for `john.residence.address`, and set an actual value for the address's `street` property, you can access the value of the `street` property through multilevel optional chaining:

```swift
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

The optional chaining question mark in this subscript call is placed immediately after `john.residence`, before the subscript brackets, because `john.residence` is the optional value on which optional chaining is being attempted.

Similarly, you can try to set a new value through a subscript with optional chaining:

```swift
john.residence?[0] = Room(name: "Bathroom")
```

This subscript setting attempt also fails, because `residence` is currently `nil`.

If you create and assign an actual `Residence` instance to `john.residence`, with one or more `Room` instances in its `rooms` array, you can use the `Residence` subscript to access the actual items in the `rooms` array through optional chaining:

```swift
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \(firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "The first room name is Living Room."
```

Accessing Subscripts of Optional Type

If a subscript returns a value of optional type—such as the key subscript of Swift's `Dictionary` type—place a question mark *after* the subscript's closing bracket to chain on its optional return value:

```swift
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80,
  94, 81]
```



```swift
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode
  instance.")
}
// Prints "tenEighty and alsoTenEighty refer to the same VideoMode
  instance."
```

Note that *identical to* (represented by three equals signs, or ===) doesn't mean the same thing as *equal to* (represented by two equals signs, or ==). *Identical to* means that two constants or variables of class type refer to exactly the same class instance. *Equal to* means that two instances are considered equal or equivalent in value, for some appropriate meaning of *equal*, as defined by the type's designer.

When you define your own custom structures and classes, it's your responsibility to decide what qualifies as two instances being equal. The process of defining your own implementations of the "equal to" and "not equal to" operators is described in Equivalence Operators.

Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use *pointers* to refer to addresses in memory. A Swift constant or variable that refers to an instance of some reference type is similar to a pointer in C, but isn't a direct pointer to an address in memory, and doesn't require you to write an asterisk (*) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift. The standard library provides pointer and buffer types that you can use if you need to interact with pointers directly—see Manual Memory Management.

Properties

Properties associate values with a particular class, structure, or enumeration. Stored properties store constant and variable values as part of an instance, whereas computed properties calculate (rather than store) a value. Computed properties are provided by classes, structures, and enumerations. Stored properties are provided only by classes and structures.

Stored and computed properties are usually associated with instances of a particular type. However, properties can also be associated with the type itself. Such properties are known as type properties.

In addition, you can define property observers to monitor changes in a property's value, which you can respond to with custom actions. Property observers can be added to stored properties you define yourself, and also to properties that a subclass inherits from its superclass.

Stored Properties

In its simplest form, a stored property is a constant or variable that is stored as part of an instance of a particular class or structure. Stored properties can be either *variable stored properties* (introduced by the `var` keyword) or *constant stored properties* (introduced by the `let` keyword).

You can provide a default value for a stored property as part of its definition, as described in Default Property Values. You can also set and modify the initial value for a stored property during initialization. This is true even for constant stored properties, as described in Assigning Constant Properties During Initialization.

The example below defines a structure called `FixedLengthRange`, which describes a range of integers whose range length cannot be changed after it is created:

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Instances of `FixedLengthRange` have a variable stored property called `firstValue` and a constant stored property called `length`. In the example above, `length` is initialized when the new range is created and cannot be changed thereafter, because it is a constant property.

Stored Properties of Constant Structure Instances

If you create an instance of a structure and assign that instance to a constant, you cannot modify the instance's properties, even if they were declared as variable properties:

This method does not specify a return type. However, functions and methods with no return type have an implicit return type of `Void`, as described in Functions Without Return Values. This means that they return a value of `()`, or an empty tuple.

If you call this method on an optional value with optional chaining, the method's return type will be `Void?`, not `Void`, because return values are always of an optional type when called through optional chaining. This enables you to use an `if` statement to check whether it was possible to call the `printNumberOfRooms()` method, even though the method does not itself define a return value. Compare the return value from the `printNumberOfRooms` call against `nil` to see if the method call was successful:

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

The same is true if you attempt to set a property through optional chaining. The example above in Accessing Properties Through Optional Chaining attempts to set an `address` value for `john.residence`, even though the `residence` property is `nil`. Any attempt to set a property through optional chaining returns a value of type `Void?`, which enables you to compare against `nil` to see if the property was set successfully:

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

Accessing Subscripts Through Optional Chaining

You can use optional chaining to try to retrieve and set a value from a subscript on an optional value, and to check whether that subscript call is successful.

> NOTE
>
> When you access a subscript on an optional value through optional chaining, you place the question mark *before* the subscript's brackets, not after. The optional chaining question mark always follows immediately after the part of the expression that is optional.

The example below tries to retrieve the name of the first room in the `rooms` array of the `john.residence` property using the subscript defined on the `Residence` class. Because `john.residence` is currently `nil`, the subscript call fails:

```swift
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

Because `john.residence` is `nil`, this optional chaining call fails in the same way as before.

You can also attempt to set a property's value through optional chaining:

```swift
let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

In this example, the attempt to set the `address` property of `john.residence` will fail, because `john.residence` is currently `nil`.

The assignment is part of the optional chaining, which means none of the code on the right-hand side of the = operator is evaluated. In the previous example, it's not easy to see that `someAddress` is never evaluated, because accessing a constant doesn't have any side effects. The listing below does the same assignment, but it uses a function to create the address. The function prints "Function was called" before returning a value, which lets you see whether the right-hand side of the = operator was evaluated.

```swift
func createAddress() -> Address {
    print("Function was called.")

    let someAddress = Address()
    someAddress.buildingNumber = "29"
    someAddress.street = "Acacia Road"

    return someAddress
}
john.residence?.address = createAddress()
```

You can tell that the `createAddress()` function isn't called, because nothing is printed.

Calling Methods Through Optional Chaining

You can use optional chaining to call a method on an optional value, and to check whether that method call is successful. You can do this even if that method does not define a return value.

The `printNumberOfRooms()` method on the `Residence` class prints the current value of `numberOfRooms`. Here's how the method looks:

```swift
func printNumberOfRooms() {
    print("The number of rooms is \(numberOfRooms)")
}
```



```swift
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable
  property
```

Because `rangeOfFourItems` is declared as a constant (with the `let` keyword), it is not possible to change its `firstValue` property, even though `firstValue` is a variable property.

This behavior is due to structures being *value types*. When an instance of a value type is marked as a constant, so are all of its properties.

The same is not true for classes, which are *reference types*. If you assign an instance of a reference type to a constant, you can still change that instance's variable properties.

Lazy Stored Properties

A *lazy stored property* is a property whose initial value is not calculated until the first time it is used. You indicate a lazy stored property by writing the `lazy` modifier before its declaration.

> NOTE
>
> You must always declare a lazy property as a variable (with the `var` keyword), because its initial value might not be retrieved until after instance initialization completes. Constant properties must always have a value *before* initialization completes, and therefore cannot be declared as lazy.

Lazy properties are useful when the initial value for a property is dependent on outside factors whose values are not known until after an instance's initialization is complete. Lazy properties are also useful when the initial value for a property requires complex or computationally expensive setup that should not be performed unless or until it is needed.

The example below uses a lazy stored property to avoid unnecessary initialization of a complex class. This example defines two classes called `DataImporter` and `DataManager`, neither of which is shown in full:

```
class DataImporter {
    /*
    DataImporter is a class to import data from an external file.
    The class is assumed to take a nontrivial amount of time to
  initialize.
    */
    var filename = "data.txt"
    // the DataImporter class would provide data importing functionality
  here
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // the DataManager class would provide data management functionality
  here
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// the DataImporter instance for the importer property has not yet been
  created
```

The `DataManager` class has a stored property called `data`, which is initialized with a new, empty array of `String` values. Although the rest of its functionality is not shown, the purpose of this `DataManager` class is to manage and provide access to this array of `String` data.

Part of the functionality of the `DataManager` class is the ability to import data from a file. This functionality is provided by the `DataImporter` class, which is assumed to take a nontrivial amount of time to initialize. This might be because a `DataImporter` instance needs to open a file and read its contents into memory when the `DataImporter` instance is initialized.

It is possible for a `DataManager` instance to manage its data without ever importing data from a file, so there is no need to create a new `DataImporter` instance when the `DataManager` itself is created. Instead, it makes more sense to create the `DataImporter` instance if and when it is first used.

Because it is marked with the `lazy` modifier, the `DataImporter` instance for the `importer` property is only created when the `importer` property is first accessed, such as when its `filename` property is queried:

```
print(manager.importer.filename)
// the DataImporter instance for the importer property has now been
  created
// Prints "data.txt"
```

> NOTE
>
> If a property marked with the `lazy` modifier is accessed by multiple threads simultaneously and the property has not yet been initialized, there is no guarantee that the property will be initialized only once.

As a shortcut to accessing its `rooms` array, this version of `Residence` provides a read-write subscript that provides access to the room at the requested index in the `rooms` array.

This version of `Residence` also provides a method called `printNumberOfRooms`, which simply prints the number of rooms in the residence.

Finally, `Residence` defines an optional property called `address`, with a type of `Address?`. The `Address` class type for this property is defined below.

The `Room` class used for the `rooms` array is a simple class with one property called `name`, and an initializer to set that property to a suitable room name:

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

The final class in this model is called `Address`. This class has three optional properties of type `String?`. The first two properties, `buildingName` and `buildingNumber`, are alternative ways to identify a particular building as part of an address. The third property, `street`, is used to name the street for that address:

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
        } else {
            return nil
        }
    }
}
```

The `Address` class also provides a method called `buildingIdentifier()`, which has a return type of `String?`. This method checks the properties of the address and returns `buildingName` if it has a value, or `buildingNumber` concatenated with `street` if both have values, or `nil` otherwise.

Accessing Properties Through Optional Chaining

As demonstrated in Optional Chaining as an Alternative to Forced Unwrapping, you can use optional chaining to access a property on an optional value, and to check if that property access is successful.

Use the classes defined above to create a new `Person` instance, and try to access its `numberOfRooms` property as before:

```swift
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

Defining Model Classes for Optional Chaining

You can use optional chaining with calls to properties, methods, and subscripts that are more than one level deep. This enables you to drill down into subproperties within complex models of interrelated types, and to check whether it is possible to access properties, methods, and subscripts on those subproperties.

The code snippets below define four model classes for use in several subsequent examples, including examples of multilevel optional chaining. These classes expand upon the `Person` and `Residence` model from above by adding a `Room` and `Address` class, with associated properties, methods, and subscripts.

The `Person` class is defined in the same way as before:

```swift
class Person {
    var residence: Residence?
}
```

The `Residence` class is more complex than before. This time, the `Residence` class defines a variable property called `rooms`, which is initialized with an empty array of type `[Room]`:

```swift
class Residence {
    var rooms = [Room]()
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \(numberOfRooms)")
    }
    var address: Address?
}
```

Because this version of `Residence` stores an array of `Room` instances, its `numberOfRooms` property is implemented as a computed property, not a stored property. The computed `numberOfRooms` property simply returns the value of the `count` property from the `rooms` array.

Stored Properties and Instance Variables

If you have experience with Objective-C, you may know that it provides *two* ways to store values and references as part of a class instance. In addition to properties, you can use instance variables as a backing store for the values stored in a property.

Swift unifies these concepts into a single property declaration. A Swift property does not have a corresponding instance variable, and the backing store for a property is not accessed directly. This approach avoids confusion about how the value is accessed in different contexts and simplifies the property's declaration into a single, definitive statement. All information about the property—including its name, type, and memory management characteristics—is defined in a single location as part of the type's definition.

Computed Properties

In addition to stored properties, classes, structures, and enumerations can define *computed properties*, which do not actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

```swift
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
                  size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// Prints "square.origin is now at (10.0, 10.0)"
```

This example defines three structures for working with geometric shapes:

- `Point` encapsulates the x- and y-coordinate of a point.

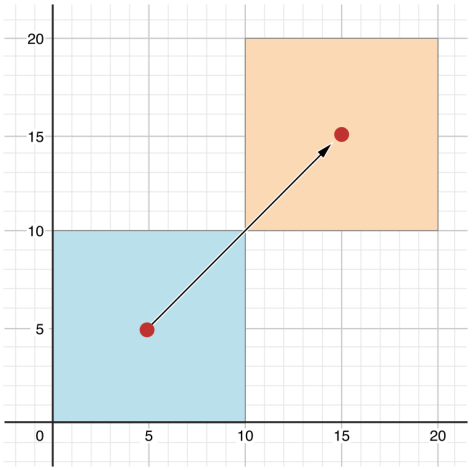
- `Size` encapsulates a `width` and a `height`.
- `Rect` defines a rectangle by an origin point and a size.

The `Rect` structure also provides a computed property called `center`. The current center position of a `Rect` can always be determined from its `origin` and `size`, and so you don't need to store the center point as an explicit `Point` value. Instead, `Rect` defines a custom getter and setter for a computed variable called `center`, to enable you to work with the rectangle's `center` as if it were a real stored property.

The example above creates a new `Rect` variable called `square`. The `square` variable is initialized with an origin point of `(0, 0)`, and a width and height of `10`. This square is represented by the blue square in the diagram below.

The `square` variable's `center` property is then accessed through dot syntax (`square.center`), which causes the getter for `center` to be called, to retrieve the current property value. Rather than returning an existing value, the getter actually calculates and returns a new `Point` to represent the center of the square. As can be seen above, the getter correctly returns a center point of `(5, 5)`.

The `center` property is then set to a new value of `(15, 15)`, which moves the square up and to the right, to the new position shown by the orange square in the diagram below. Setting the `center` property calls the setter for `center`, which modifies the `x` and `y` values of the stored `origin` property, and moves the square to its new position.



Shorthand Setter Declaration

If a computed property's setter does not define a name for the new value to be set, a default name of `newValue` is used. Here's an alternative version of the `Rect` structure, which takes advantage of this shorthand notation:

If you create a new `Person` instance, its `residence` property is default initialized to `nil`, by virtue of being optional. In the code below, `john` has a `residence` property value of `nil`:

```
let john = Person()
```

If you try to access the `numberOfRooms` property of this person's `residence`, by placing an exclamation mark after `residence` to force the unwrapping of its value, you trigger a runtime error, because there is no `residence` value to unwrap:

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

The code above succeeds when `john.residence` has a non-`nil` value and will set `roomCount` to an `Int` value containing the appropriate number of rooms. However, this code always triggers a runtime error when `residence` is `nil`, as illustrated above.

Optional chaining provides an alternative way to access the value of `numberOfRooms`. To use optional chaining, use a question mark in place of the exclamation mark:

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

This tells Swift to "chain" on the optional `residence` property and to retrieve the value of `numberOfRooms` if `residence` exists.

Because the attempt to access `numberOfRooms` has the potential to fail, the optional chaining attempt returns a value of type `Int?`, or "optional `Int`". When `residence` is `nil`, as in the example above, this optional `Int` will also be `nil`, to reflect the fact that it was not possible to access `numberOfRooms`. The optional `Int` is accessed through optional binding to unwrap the integer and assign the nonoptional value to the `roomCount` variable.

Note that this is true even though `numberOfRooms` is a nonoptional `Int`. The fact that it is queried through an optional chain means that the call to `numberOfRooms` will always return an `Int?` instead of an `Int`.

You can assign a `Residence` instance to `john.residence`, so that it no longer has a `nil` value:

```
john.residence = Residence()
```

`john.residence` now contains an actual `Residence` instance, rather than `nil`. If you try to access `numberOfRooms` with the same optional chaining as before, it will now return an `Int?` that contains the default `numberOfRooms` value of `1`:

Optional Chaining

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be `nil`. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is `nil`, the property, method, or subscript call returns `nil`. Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is `nil`.

> NOTE
>
> Optional chaining in Swift is similar to messaging `nil` in Objective-C, but in a way that works for any type, and that can be checked for success or failure.

Optional Chaining as an Alternative to Forced Unwrapping

You specify optional chaining by placing a question mark (`?`) after the optional value on which you wish to call a property, method or subscript if the optional is non-`nil`. This is very similar to placing an exclamation mark (`!`) after an optional value to force the unwrapping of its value. The main difference is that optional chaining fails gracefully when the optional is `nil`, whereas forced unwrapping triggers a runtime error when the optional is `nil`.

To reflect the fact that optional chaining can be called on a `nil` value, the result of an optional chaining call is always an optional value, even if the property, method, or subscript you are querying returns a nonoptional value. You can use this optional return value to check whether the optional chaining call was successful (the returned optional contains a value), or did not succeed due to a `nil` value in the chain (the returned optional value is `nil`).

Specifically, the result of an optional chaining call is of the same type as the expected return value, but wrapped in an optional. A property that normally returns an `Int` will return an `Int?` when accessed through optional chaining.

The next several code snippets demonstrate how optional chaining differs from forced unwrapping and enables you to check for success.

First, two classes called `Person` and `Residence` are defined:

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

`Residence` instances have a single `Int` property called `numberOfRooms`, with a default value of `1`. `Person` instances have an optional `residence` property of type `Residence?`.


```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Read-Only Computed Properties

A computed property with a getter but no setter is known as a *read-only computed property*. A read-only computed property always returns a value, and can be accessed through dot syntax, but cannot be set to a different value.

> NOTE
>
> You must declare computed properties—including read-only computed properties—as variable properties with the `var` keyword, because their value is not fixed. The `let` keyword is only used for constant properties, to indicate that their values cannot be changed once they are set as part of instance initialization.

You can simplify the declaration of a read-only computed property by removing the `get` keyword and its braces:

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// Prints "the volume of fourByFiveByTwo is 40.0"
```

This example defines a new structure called `Cuboid`, which represents a 3D rectangular box with `width`, `height`, and `depth` properties. This structure also has a read-only computed property called `volume`, which calculates and returns the current volume of the cuboid. It doesn't make sense for `volume` to be settable, because it would be ambiguous as to which values of `width`, `height`, and `depth` should be used for a particular `volume` value. Nonetheless, it is useful for a `Cuboid` to provide a read-only computed property to enable external users to discover its current calculated volume.

Property Observers

Property observers observe and respond to changes in a property's value. Property observers are called every time a property's value is set, even if the new value is the same as the property's current value.

You can add property observers to any stored properties you define, except for lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass. You don't need to define property observers for nonoverridden computed properties, because you can observe and respond to changes to their value in the computed property's setter. Property overriding is described in Overriding.

You have the option to define either or both of these observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

If you implement a `willSet` observer, it's passed the new property value as a constant parameter. You can specify a name for this parameter as part of your `willSet` implementation. If you don't write the parameter name and parentheses within your implementation, the parameter is made available with a default parameter name of `newValue`.

Similarly, if you implement a `didSet` observer, it's passed a constant parameter containing the old property value. You can name the parameter or use the default parameter name of `oldValue`. If you assign a value to a property within its own `didSet` observer, the new value that you assign replaces the one that was just set.

> NOTE
>
> The `willSet` and `didSet` observers of superclass properties are called when a property is set in a subclass initializer, after the superclass initializer has been called. They are not called while a class is setting its own properties, before the superclass initializer has been called.
>
> For more information about initializer delegation, see Initializer Delegation for Value Types and Initializer Delegation for Class Types.

Here's an example of `willSet` and `didSet` in action. The example below defines a new class called `StepCounter`, which tracks the total number of steps that a person takes while walking. This class might be used with input data from a pedometer or other step counter to keep track of a person's exercise during their daily routine.


```
var playerOne: Player? = Player(coins: 100)
print("A new player has joined the game with \(playerOne!.coinsInPurse)
  coins")
// Prints "A new player has joined the game with 100 coins"
print("There are now \(Bank.coinsInBank) coins left in the bank")
// Prints "There are now 9900 coins left in the bank"
```

A new `Player` instance is created, with a request for 100 coins if they are available. This `Player` instance is stored in an optional `Player` variable called `playerOne`. An optional variable is used here, because players can leave the game at any point. The optional lets you track whether there is currently a player in the game.

Because `playerOne` is an optional, it is qualified with an exclamation mark (`!`) when its `coinsInPurse` property is accessed to print its default number of coins, and whenever its `win(coins:)` method is called:

```
playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse)
  coins")
// Prints "PlayerOne won 2000 coins & now has 2100 coins"
print("The bank now only has \(Bank.coinsInBank) coins left")
// Prints "The bank now only has 7900 coins left"
```

Here, the player has won 2,000 coins. The player's purse now contains 2,100 coins, and the bank has only 7,900 coins left.

```
playerOne = nil
print("PlayerOne has left the game")
// Prints "PlayerOne has left the game"
print("The bank now has \(Bank.coinsInBank) coins")
// Prints "The bank now has 10000 coins"
```

The player has now left the game. This is indicated by setting the optional `playerOne` variable to `nil`, meaning "no `Player` instance." At the point that this happens, the `playerOne` variable's reference to the `Player` instance is broken. No other properties or variables are still referring to the `Player` instance, and so it is deallocated in order to free up its memory. Just before this happens, its deinitializer is called automatically, and its coins are returned to the bank.

```
class Bank {
    static var coinsInBank = 10_000
    static func distribute(coins numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receive(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank` keeps track of the current number of coins it holds with its `coinsInBank` property. It also offers two methods—`distribute(coins:)` and `receive(coins:)`—to handle the distribution and collection of coins.

The `distribute(coins:)` method checks that there are enough coins in the bank before distributing them. If there are not enough coins, `Bank` returns a smaller number than the number that was requested (and returns zero if no coins are left in the bank). It returns an integer value to indicate the actual number of coins that were provided.

The `receive(coins:)` method simply adds the received number of coins back into the bank's coin store.

The `Player` class describes a player in the game. Each player has a certain number of coins stored in their purse at any time. This is represented by the player's `coinsInPurse` property:

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}
```

Each `Player` instance is initialized with a starting allowance of a specified number of coins from the bank during initialization, although a `Player` instance may receive fewer than that number if not enough coins are available.

The `Player` class defines a `win(coins:)` method, which retrieves a certain number of coins from the bank and adds them to the player's purse. The `Player` class also implements a deinitializer, which is called just before a `Player` instance is deallocated. Here, the deinitializer simply returns all of the player's coins to the bank:


```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

The `StepCounter` class declares a `totalSteps` property of type `Int`. This is a stored property with `willSet` and `didSet` observers.

The `willSet` and `didSet` observers for `totalSteps` are called whenever the property is assigned a new value. This is true even if the new value is the same as the current value.

This example's `willSet` observer uses a custom parameter name of `newTotalSteps` for the upcoming new value. In this example, it simply prints out the value that is about to be set.

The `didSet` observer is called after the value of `totalSteps` is updated. It compares the new value of `totalSteps` against the old value. If the total number of steps has increased, a message is printed to indicate how many new steps have been taken. The `didSet` observer does not provide a custom parameter name for the old value, and the default name of `oldValue` is used instead.

> NOTE
>
> If you pass a property that has observers to a function as an in-out parameter, the `willSet` and `didSet` observers are always called. This is because of the copy-in copy-out memory model for in-out parameters: The value is always written back to the property at the end of the function. For a detailed discussion of the behavior of in-out parameters, see In-Out Parameters.

Global and Local Variables

The capabilities described above for computing and observing properties are also available to *global variables* and *local variables*. Global variables are variables that are defined outside of any function, method, closure, or type context. Local variables are variables that are defined within a function, method, or closure context.

The global and local variables you have encountered in previous chapters have all been *stored variables*. Stored variables, like stored properties, provide storage for a value of a certain type and allow that value to be set and retrieved.

However, you can also define *computed variables* and define observers for stored variables, in either a global or local scope. Computed variables calculate their value, rather than storing it, and they are written in the same way as computed properties.

> NOTE
>
> Global constants and variables are always computed lazily, in a similar manner to Lazy Stored Properties. Unlike lazy stored properties, global constants and variables do not need to be marked with the `lazy` modifier.
>
> Local constants and variables are never computed lazily.

Type Properties

Instance properties are properties that belong to an instance of a particular type. Every time you create a new instance of that type, it has its own set of property values, separate from any other instance.

You can also define properties that belong to the type itself, not to any one instance of that type. There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called *type properties*.

Type properties are useful for defining values that are universal to *all* instances of a particular type, such as a constant property that all instances can use (like a static constant in C), or a variable property that stores a value that is global to all instances of that type (like a static variable in C).

Stored type properties can be variables or constants. Computed type properties are always declared as variable properties, in the same way as computed instance properties.

> NOTE
>
> Unlike stored instance properties, you must always give stored type properties a default value. This is because the type itself does not have an initializer that can assign a value to a stored type property at initialization time.
>
> Stored type properties are lazily initialized on their first access. They are guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they do not need to be marked with the `lazy` modifier.

Type Property Syntax

In C and Objective-C, you define static constants and variables associated with a type as *global* static variables. In Swift, however, type properties are written as part of the type's definition, within the type's outer curly braces, and each type property is explicitly scoped to the type it supports.

Deinitialization

A *deinitializer* is called immediately before a class instance is deallocated. You write deinitializers with the `deinit` keyword, similar to how initializers are written with the `init` keyword. Deinitializers are only available on class types.

How Deinitialization Works

Swift automatically deallocates your instances when they are no longer needed, to free up resources. Swift handles the memory management of instances through *automatic reference counting* (*ARC*), as described in Automatic Reference Counting. Typically you don't need to perform manual cleanup when your instances are deallocated. However, when you are working with your own resources, you might need to perform some additional cleanup yourself. For example, if you create a custom class to open a file and write some data to it, you might need to close the file before the class instance is deallocated.

Class definitions can have at most one deinitializer per class. The deinitializer does not take any parameters and is written without parentheses:

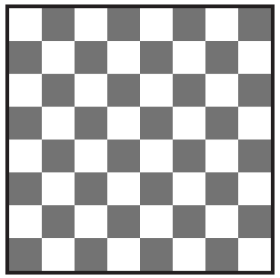
```
deinit {
    // perform the deinitialization
}
```

Deinitializers are called automatically, just before instance deallocation takes place. You are not allowed to call a deinitializer yourself. Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer implementation. Superclass deinitializers are always called, even if a subclass does not provide its own deinitializer.

Because an instance is not deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it is called on and can modify its behavior based on those properties (such as looking up the name of a file that needs to be closed).

Deinitializers in Action

Here's an example of a deinitializer in action. This example defines two new types, `Bank` and `Player`, for a simple game. The `Bank` class manages a made-up currency, which can never have more than 10,000 coins in circulation. There can only ever be one `Bank` in the game, and so the `Bank` is implemented as a class with type properties and methods to store and manage its current state:



To represent this game board, the `Chessboard` structure has a single property called `boardColors`, which is an array of 64 `Bool` values. A value of `true` in the array represents a black square and a value of `false` represents a white square. The first item in the array represents the top left square on the board and the last item in the array represents the bottom right square on the board.

The `boardColors` array is initialized with a closure to set up its color values:

```
struct Chessboard {
    let boardColors: [Bool] = {
        var temporaryBoard = [Bool]()
        var isBlack = false
        for i in 1...8 {
            for j in 1...8 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
    }()
    func squareIsBlackAt(row: Int, column: Int) -> Bool {
        return boardColors[(row * 8) + column]
    }
}
```

Whenever a new `Chessboard` instance is created, the closure is executed, and the default value of `boardColors` is calculated and returned. The closure in the example above calculates and sets the appropriate color for each square on the board in a temporary array called `temporaryBoard`, and returns this temporary array as the closure's return value once its setup is complete. The returned array value is stored in `boardColors` and can be queried with the `squareIsBlackAt(row:column:)` utility function:

```
let board = Chessboard()
print(board.squareIsBlackAt(row: 0, column: 1))
// Prints "true"
print(board.squareIsBlackAt(row: 7, column: 7))
// Prints "false"
```


You define type properties with the `static` keyword. For computed type properties for class types, you can use the `class` keyword instead to allow subclasses to override the superclass's implementation. The example below shows the syntax for stored and computed type properties:

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

> NOTE
>
> The computed type property examples above are for read-only computed type properties, but you can also define read-write computed type properties with the same syntax as for computed instance properties.

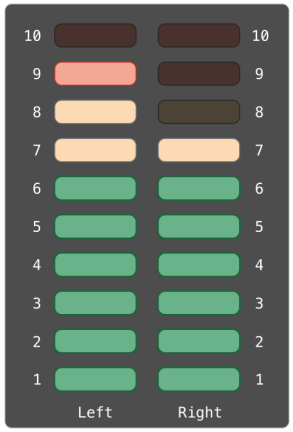
Querying and Setting Type Properties

Type properties are queried and set with dot syntax, just like instance properties. However, type properties are queried and set on the *type*, not on an instance of that type. For example:

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27"
```

The examples that follow use two stored type properties as part of a structure that models an audio level meter for a number of audio channels. Each channel has an integer audio level between `0` and `10` inclusive.

The figure below illustrates how two of these audio channels can be combined to model a stereo audio level meter. When a channel's audio level is `0`, none of the lights for that channel are lit. When the audio level is `10`, all of the lights for that channel are lit. In this figure, the left channel has a current level of `9`, and the right channel has a current level of `7`:



The audio channels described above are represented by instances of the `AudioChannel` structure:

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {
                // cap the new audio level to the threshold level
                currentLevel = AudioChannel.thresholdLevel
            }
            if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                // store this as the new overall maximum input level
                AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}
```

The `AudioChannel` structure defines two stored type properties to support its functionality. The first, `thresholdLevel`, defines the maximum threshold value an audio level can take. This is a constant value of `10` for all `AudioChannel` instances. If an audio signal comes in with a higher value than `10`, it will be capped to this threshold value (as described below).

The second type property is a variable stored property called `maxInputLevelForAllChannels`. This keeps track of the maximum input value that has been received by *any* `AudioChannel` instance. It starts with an initial value of `0`.

The `AudioChannel` structure also defines a stored instance property called `currentLevel`, which represents the channel's current audio level on a scale of `0` to `10`.


```
class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required initializer goes here
    }
}
```

> NOTE
>
> You do not have to provide an explicit implementation of a required initializer if you can satisfy the requirement with an inherited initializer.

Setting a Default Property Value with a Closure or Function

If a stored property's default value requires some customization or setup, you can use a closure or global function to provide a customized default value for that property. Whenever a new instance of the type that the property belongs to is initialized, the closure or function is called, and its return value is assigned as the property's default value.

These kinds of closures or functions typically create a temporary value of the same type as the property, tailor that value to represent the desired initial state, and then return that temporary value to be used as the property's default value.

Here's a skeleton outline of how a closure can be used to provide a default property value:

```
class SomeClass {
    let someProperty: SomeType = {
        // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
    }()
}
```

Note that the closure's end curly brace is followed by an empty pair of parentheses. This tells Swift to execute the closure immediately. If you omit these parentheses, you are trying to assign the closure itself to the property, and not the return value of the closure.

> NOTE
>
> If you use a closure to initialize a property, remember that the rest of the instance has not yet been initialized at the point that the closure is executed. This means that you cannot access any other property values from within your closure, even if those properties have default values. You also cannot use the implicit `self` property, or call any of the instance's methods.

The example below defines a structure called `Chessboard`, which models a board for the game of chess. Chess is played on an 8 x 8 board, with alternating black and white squares.

The `AutomaticallyNamedDocument` overrides its superclass's failable `init?(name:)` initializer with a nonfailable `init(name:)` initializer. Because `AutomaticallyNamedDocument` copes with the empty string case in a different way than its superclass, its initializer does not need to fail, and so it provides a nonfailable version of the initializer instead.

You can use forced unwrapping in an initializer to call a failable initializer from the superclass as part of the implementation of a subclass's nonfailable initializer. For example, the `UntitledDocument` subclass below is always named `"[Untitled]"`, and it uses the failable `init(name:)` initializer from its superclass during initialization.

```
class UntitledDocument: Document {
    override init() {
        super.init(name: "[Untitled]")!
    }
}
```

In this case, if the `init(name:)` initializer of the superclass were ever called with an empty string as the name, the forced unwrapping operation would result in a runtime error. However, because it's called with a string constant, you can see that the initializer won't fail, so no runtime error can occur in this case.

The init! Failable Initializer

You typically define a failable initializer that creates an optional instance of the appropriate type by placing a question mark after the `init` keyword (`init?`). Alternatively, you can define a failable initializer that creates an implicitly unwrapped optional instance of the appropriate type. Do this by placing an exclamation mark after the `init` keyword (`init!`) instead of a question mark.

You can delegate from `init?` to `init!` and vice versa, and you can override `init?` with `init!` and vice versa. You can also delegate from `init` to `init!`, although doing so will trigger an assertion if the `init!` initializer causes initialization to fail.

Required Initializers

Write the `required` modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

You must also write the `required` modifier before every subclass implementation of a required initializer, to indicate that the initializer requirement applies to further subclasses in the chain. You do not write the `override` modifier when overriding a required designated initializer:

The `currentLevel` property has a `didSet` property observer to check the value of `currentLevel` whenever it is set. This observer performs two checks:

- If the new value of `currentLevel` is greater than the allowed `thresholdLevel`, the property observer caps `currentLevel` to `thresholdLevel`.
- If the new value of `currentLevel` (after any capping) is higher than any value previously received by *any* `AudioChannel` instance, the property observer stores the new `currentLevel` value in the `maxInputLevelForAllChannels` type property.

> NOTE
>
> In the first of these two checks, the `didSet` observer sets `currentLevel` to a different value. This does not, however, cause the observer to be called again.

You can use the `AudioChannel` structure to create two new audio channels called `leftChannel` and `rightChannel`, to represent the audio levels of a stereo sound system:

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

If you set the `currentLevel` of the *left* channel to `7`, you can see that the `maxInputLevelForAllChannels` type property is updated to equal `7`:

```
leftChannel.currentLevel = 7
print(leftChannel.currentLevel)
// Prints "7"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "7"
```

If you try to set the `currentLevel` of the *right* channel to `11`, you can see that the right channel's `currentLevel` property is capped to the maximum value of `10`, and the `maxInputLevelForAllChannels` type property is updated to equal `10`:

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)
// Prints "10"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "10"
```

Methods

Methods are functions that are associated with a particular type. Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type. Classes, structures, and enumerations can also define type methods, which are associated with the type itself. Type methods are similar to class methods in Objective-C.

The fact that structures and enumerations can define methods in Swift is a major difference from C and Objective-C. In Objective-C, classes are the only types that can define methods. In Swift, you can choose whether to define a class, structure, or enumeration, and still have the flexibility to define methods on the type you create.

Instance Methods

Instance methods are functions that belong to instances of a particular class, structure, or enumeration. They support the functionality of those instances, either by providing ways to access and modify instance properties, or by providing functionality related to the instance's purpose. Instance methods have exactly the same syntax as functions, as described in Functions.

You write an instance method within the opening and closing braces of the type it belongs to. An instance method has implicit access to all other instance methods and properties of that type. An instance method can be called only on a specific instance of the type it belongs to. It cannot be called in isolation without an existing instance.

Here's an example that defines a simple `Counter` class, which can be used to count the number of times an action occurs:

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

The `Counter` class defines three instance methods:

- `increment()` increments the counter by 1.
- `increment(by: Int)` increments the counter by a specified integer amount.
- `reset()` resets the counter to zero.

The `Counter` class also declares a variable property, `count`, to keep track of the current counter value.

You can override a superclass failable initializer in a subclass, just like any other initializer. Alternatively, you can override a superclass failable initializer with a subclass *nonfailable* initializer. This enables you to define a subclass for which initialization cannot fail, even though initialization of the superclass is allowed to fail.

Note that if you override a failable superclass initializer with a nonfailable subclass initializer, the only way to delegate up to the superclass initializer is to force-unwrap the result of the failable superclass initializer.

> NOTE
>
> You can override a failable initializer with a nonfailable initializer but not the other way around.

The example below defines a class called `Document`. This class models a document that can be initialized with a `name` property that is either a nonempty string value or `nil`, but cannot be an empty string:

```
class Document {
    var name: String?
    // this initializer creates a document with a nil name value
    init() {}
    // this initializer creates a document with a nonempty name value
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
```

The next example defines a subclass of `Document` called `AutomaticallyNamedDocument`. The `AutomaticallyNamedDocument` subclass overrides both of the designated initializers introduced by `Document`. These overrides ensure that an `AutomaticallyNamedDocument` instance has an initial `name` value of `"[Untitled]"` if the instance is initialized without a name, or if an empty string is passed to the `init(name:)` initializer:

```
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
}
```

```swift
class Product {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

class CartItem: Product {
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.quantity = quantity
        super.init(name: name)
    }
}
```

The failable initializer for `CartItem` starts by validating that it has received a `quantity` value of `1` or more. If the `quantity` is invalid, the entire initialization process fails immediately and no further initialization code is executed. Likewise, the failable initializer for `Product` checks the `name` value, and the initializer process fails immediately if `name` is the empty string.

If you create a `CartItem` instance with a nonempty name and a quantity of `1` or more, initialization succeeds:

```swift
if let twoSocks = CartItem(name: "sock", quantity: 2) {
    print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
}
// Prints "Item: sock, quantity: 2"
```

If you try to create a `CartItem` instance with a `quantity` value of `0`, the `CartItem` initializer causes initialization to fail:

```swift
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
    print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
} else {
    print("Unable to initialize zero shirts")
}
// Prints "Unable to initialize zero shirts"
```

Similarly, if you try to create a `CartItem` instance with an empty `name` value, the superclass `Product` initializer causes initialization to fail:

```swift
if let oneUnnamed = CartItem(name: "", quantity: 1) {
    print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")
} else {
    print("Unable to initialize one unnamed product")
}
// Prints "Unable to initialize one unnamed product"
```

Overriding a Failable Initializer

You call instance methods with the same dot syntax as properties:

```swift
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.increment(by: 5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

Function parameters can have both a name (for use within the function's body) and an argument label (for use when calling the function), as described in Function Argument Labels and Parameter Names. The same is true for method parameters, because methods are just functions that are associated with a type.

The self Property

Every instance of a type has an implicit property called `self`, which is exactly equivalent to the instance itself. You use the `self` property to refer to the current instance within its own instance methods.

The `increment()` method in the example above could have been written like this:

```swift
func increment() {
    self.count += 1
}
```

In practice, you don't need to write `self` in your code very often. If you don't explicitly write `self`, Swift assumes that you are referring to a property or method of the current instance whenever you use a known property or method name within a method. This assumption is demonstrated by the use of `count` (rather than `self.count`) inside the three instance methods for `Counter`.

The main exception to this rule occurs when a parameter name for an instance method has the same name as a property of that instance. In this situation, the parameter name takes precedence, and it becomes necessary to refer to the property in a more qualified way. You use the `self` property to distinguish between the parameter name and the property name.

Here, `self` disambiguates between a method parameter called `x` and an instance property that is also called `x`:

```swift
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOf(x: 1.0) {
    print("This point is to the right of the line where x == 1.0")
}
// Prints "This point is to the right of the line where x == 1.0"
```

Without the `self` prefix, Swift would assume that both uses of `x` referred to the method parameter called `x`.

Modifying Value Types from Within Instance Methods

Structures and enumerations are *value types*. By default, the properties of a value type cannot be modified from within its instance methods.

However, if you need to modify the properties of your structure or enumeration within a particular method, you can opt in to *mutating* behavior for that method. The method can then mutate (that is, change) its properties from within the method, and any changes that it makes are written back to the original structure when the method ends. The method can also assign a completely new instance to its implicit `self` property, and this new instance will replace the existing one when the method ends.

You can opt in to this behavior by placing the `mutating` keyword before the `func` keyword for that method:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// Prints "The point is now at (3.0, 4.0)"
```

The `Point` structure above defines a mutating `moveBy(x:y:)` method, which moves a `Point` instance by a certain amount. Instead of returning a new point, this method actually modifies the point on which it is called. The `mutating` keyword is added to its definition to enable it to modify its properties.

Note that you cannot call a mutating method on a constant of structure type, because its properties cannot be changed, even if they are variable properties, as described in Stored Properties of Constant Structure Instances:

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveBy(x: 2.0, y: 3.0)
// this will report an error
```

Assigning to self Within a Mutating Method

Mutating methods can assign an entirely new instance to the implicit `self` property. The `Point` example shown above could have been written in the following way instead:


```
enum TemperatureUnit: Character {
    case kelvin = "K", celsius = "C", fahrenheit = "F"
}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization
  succeeded.")
}
// Prints "This is a defined temperature unit, so initialization
  succeeded."

let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization
  failed.")
}
// Prints "This is not a defined temperature unit, so initialization
  failed."
```

Propagation of Initialization Failure

A failable initializer of a class, structure, or enumeration can delegate across to another failable initializer from the same class, structure, or enumeration. Similarly, a subclass failable initializer can delegate up to a superclass failable initializer.

In either case, if you delegate to another initializer that causes initialization to fail, the entire initialization process fails immediately, and no further initialization code is executed.

> NOTE
>
> A failable initializer can also delegate to a nonfailable initializer. Use this approach if you need to add a potential failure state to an existing initialization process that does not otherwise fail.

The example below defines a subclass of `Product` called `CartItem`. The `CartItem` class models an item in an online shopping cart. `CartItem` introduces a stored constant property called `quantity` and ensures that this property always has a value of at least `1`:

```
enum TemperatureUnit {
    case kelvin, celsius, fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .kelvin
        case "C":
            self = .celsius
        case "F":
            self = .fahrenheit
        default:
            return nil
        }
    }
}
```

You can use this failable initializer to choose an appropriate enumeration case for the three possible states and to cause initialization to fail if the parameter does not match one of these states:

```
let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This is not a defined temperature unit, so initialization failed.")
}
// Prints "This is not a defined temperature unit, so initialization failed."
```

Failable Initializers for Enumerations with Raw Values

Enumerations with raw values automatically receive a failable initializer, `init?(rawValue:)`, that takes a parameter called `rawValue` of the appropriate raw-value type and selects a matching enumeration case if one is found, or triggers an initialization failure if no matching value exists.

You can rewrite the `TemperatureUnit` example from above to use raw values of type `Character` and to take advantage of the `init?(rawValue:)` initializer:


```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

This version of the mutating `moveBy(x:y:)` method creates a new structure whose `x` and `y` values are set to the target location. The end result of calling this alternative version of the method will be exactly the same as for calling the earlier version.

Mutating methods for enumerations can set the implicit `self` parameter to be a different case from the same enumeration:

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next()
// ovenLight is now equal to .high
ovenLight.next()
// ovenLight is now equal to .off
```

This example defines an enumeration for a three-state switch. The switch cycles between three different power states (`off`, `low` and `high`) every time its `next()` method is called.

Type Methods

Instance methods, as described above, are methods that are called on an instance of a particular type. You can also define methods that are called on the type itself. These kinds of methods are called *type methods*. You indicate type methods by writing the `static` keyword before the method's `func` keyword. Classes may also use the `class` keyword to allow subclasses to override the superclass's implementation of that method.

> NOTE
>
> In Objective-C, you can define type-level methods only for Objective-C classes. In Swift, you can define type-level methods for all classes, structures, and enumerations. Each type method is explicitly scoped to the type it supports.

Type methods are called with dot syntax, like instance methods. However, you call type methods on the type, not on an instance of that type. Here's how you call a type method on a class called `SomeClass`:

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

Within the body of a type method, the implicit `self` property refers to the type itself, rather than an instance of that type. This means that you can use `self` to disambiguate between type properties and type method parameters, just as you do for instance properties and instance method parameters.

More generally, any unqualified method and property names that you use within the body of a type method will refer to other type-level methods and properties. A type method can call another type method with the other method's name, without needing to prefix it with the type name. Similarly, type methods on structures and enumerations can access type properties by using the type property's name without a type name prefix.

The example below defines a structure called `LevelTracker`, which tracks a player's progress through the different levels or stages of a game. It is a single-player game, but can store information for multiple players on a single device.

All of the game's levels (apart from level one) are locked when the game is first played. Every time a player finishes a level, that level is unlocked for all players on the device. The `LevelTracker` structure uses type properties and methods to keep track of which levels of the game have been unlocked. It also tracks the current level for an individual player.


```
let someCreature = Animal(species: "Giraffe")
// someCreature is of type Animal?, not Animal

if let giraffe = someCreature {
    print("An animal was initialized with a species of \
  (giraffe.species)")
}
// Prints "An animal was initialized with a species of Giraffe"
```

If you pass an empty string value to the failable initializer's `species` parameter, the initializer triggers an initialization failure:

```
let anonymousCreature = Animal(species: "")
// anonymousCreature is of type Animal?, not Animal

if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}
// Prints "The anonymous creature could not be initialized"
```

> NOTE
>
> Checking for an empty string value (such as `""` rather than `"Giraffe"`) is not the same as checking for `nil` to indicate the absence of an *optional* `String` value. In the example above, an empty string (`""`) is a valid, nonoptional `String`. However, it is not appropriate for an animal to have an empty string as the value of its `species` property. To model this restriction, the failable initializer triggers an initialization failure if an empty string is found.

Failable Initializers for Enumerations

You can use a failable initializer to select an appropriate enumeration case based on one or more parameters. The initializer can then fail if the provided parameters do not match an appropriate enumeration case.

The example below defines an enumeration called `TemperatureUnit`, with three possible states (`kelvin`, `celsius`, and `fahrenheit`). A failable initializer is used to find an appropriate enumeration case for a `Character` value representing a temperature symbol:

> NOTE
>
> You cannot define a failable and a nonfailable initializer with the same parameter types and names.

A failable initializer creates an *optional* value of the type it initializes. You write `return nil` within a failable initializer to indicate a point at which initialization failure can be triggered.

> NOTE
>
> Strictly speaking, initializers do not return a value. Rather, their role is to ensure that `self` is fully and correctly initialized by the time that initialization ends. Although you write `return nil` to trigger an initialization failure, you do not use the `return` keyword to indicate initialization success.

For instance, failable initializers are implemented for numeric type conversions. To ensure conversion between numeric types maintains the value exactly, use the `init(exactly:)` initializer. If the type conversion cannot maintain the value, the initializer fails.

```
let wholeNumber: Double = 12345.0
let pi = 3.14159

if let valueMaintained = Int(exactly: wholeNumber) {
    print("\(wholeNumber) conversion to Int maintains value of \
  (valueMaintained)")
}
// Prints "12345.0 conversion to Int maintains value of 12345"

let valueChanged = Int(exactly: pi)
// valueChanged is of type Int?, not Int

if valueChanged == nil {
    print("\(pi) conversion to Int does not maintain value")
}
// Prints "3.14159 conversion to Int does not maintain value"
```

The example below defines a structure called `Animal`, with a constant `String` property called `species`. The `Animal` structure also defines a failable initializer with a single parameter called `species`. This initializer checks if the `species` value passed to the initializer is an empty string. If an empty string is found, an initialization failure is triggered. Otherwise, the `species` property's value is set, and initialization succeeds:

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
```

You can use this failable initializer to try to initialize a new `Animal` instance and to check if initialization succeeded:


```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    var currentLevel = 1

    static func unlock(_ level: Int) {
        if level > highestUnlockedLevel { highestUnlockedLevel = level }
    }

    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }

    @discardableResult
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

The `LevelTracker` structure keeps track of the highest level that any player has unlocked. This value is stored in a type property called `highestUnlockedLevel`.

`LevelTracker` also defines two type functions to work with the `highestUnlockedLevel` property. The first is a type function called `unlock(_:)`, which updates the value of `highestUnlockedLevel` whenever a new level is unlocked. The second is a convenience type function called `isUnlocked(_:)`, which returns `true` if a particular level number is already unlocked. (Note that these type methods can access the `highestUnlockedLevel` type property without your needing to write it as `LevelTracker.highestUnlockedLevel`.)

In addition to its type property and type methods, `LevelTracker` tracks an individual player's progress through the game. It uses an instance property called `currentLevel` to track the level that a player is currently playing.

To help manage the `currentLevel` property, `LevelTracker` defines an instance method called `advance(to:)`. Before updating `currentLevel`, this method checks whether the requested new level is already unlocked. The `advance(to:)` method returns a Boolean value to indicate whether or not it was actually able to set `currentLevel`. Because it's not necessarily a mistake for code that calls the `advance(to:)` method to ignore the return value, this function is marked with the `@discardableResult` attribute. For more information about this attribute, see Attributes.

The `LevelTracker` structure is used with the `Player` class, shown below, to track and update the progress of an individual player:

```swift
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

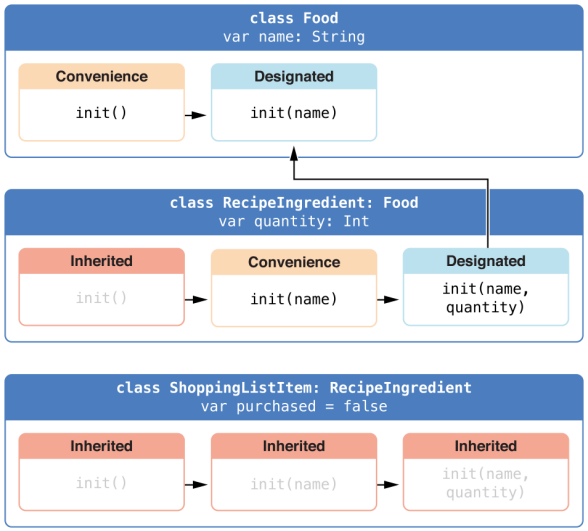
The `Player` class creates a new instance of `LevelTracker` to track that player's progress. It also provides a method called `complete(level:)`, which is called whenever a player completes a particular level. This method unlocks the next level for all players and updates the player's progress to move them to the next level. (The Boolean return value of `advance(to:)` is ignored, because the level is known to have been unlocked by the call to `LevelTracker.unlock(_:)` on the previous line.)

You can create an instance of the `Player` class for a new player, and see what happens when the player completes level one:

```swift
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("highest unlocked level is now \
  (LevelTracker.highestUnlockedLevel)")
// Prints "highest unlocked level is now 2"
```

If you create a second player, whom you try to move to a level that is not yet unlocked by any player in the game, the attempt to set the player's current level fails:

```swift
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("player is now on level 6")
} else {
    print("level 6 has not yet been unlocked")
}
// Prints "level 6 has not yet been unlocked"
```

You can use all three of the inherited initializers to create a new `ShoppingListItem` instance:

```swift
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
// 1 x Orange juice ✔
// 1 x Bacon ✘
// 6 x Eggs ✘
```

Here, a new array called `breakfastList` is created from an array literal containing three new `ShoppingListItem` instances. The type of the array is inferred to be `[ShoppingListItem]`. After the array is created, the name of the `ShoppingListItem` at the start of the array is changed from `"[Unnamed]"` to `"Orange juice"` and it is marked as having been purchased. Printing the description of each item in the array shows that their default states have been set as expected.

Failable Initializers

It is sometimes useful to define a class, structure, or enumeration for which initialization can fail. This failure might be triggered by invalid initialization parameter values, the absence of a required external resource, or some other condition that prevents initialization from succeeding.

To cope with initialization conditions that can fail, define one or more failable initializers as part of a class, structure, or enumeration definition. You write a failable initializer by placing a question mark after the `init` keyword (`init?`).

In this example, the superclass for `RecipeIngredient` is `Food`, which has a single convenience initializer called `init()`. This initializer is therefore inherited by `RecipeIngredient`. The inherited version of `init()` functions in exactly the same way as the `Food` version, except that it delegates to the `RecipeIngredient` version of `init(name: String)` rather than the `Food` version.

All three of these initializers can be used to create new `RecipeIngredient` instances:

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

The third and final class in the hierarchy is a subclass of `RecipeIngredient` called `ShoppingListItem`. The `ShoppingListItem` class models a recipe ingredient as it appears in a shopping list.

Every item in the shopping list starts out as "unpurchased". To represent this fact, `ShoppingListItem` introduces a Boolean property called `purchased`, with a default value of `false`. `ShoppingListItem` also adds a computed `description` property, which provides a textual description of a `ShoppingListItem` instance:

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

> NOTE
>
> `ShoppingListItem` does not define an initializer to provide an initial value for `purchased`, because items in a shopping list (as modeled here) always start out unpurchased.

Because it provides a default value for all of the properties it introduces and does not define any initializers itself, `ShoppingListItem` automatically inherits *all* of the designated and convenience initializers from its superclass.

The figure below shows the overall initializer chain for all three classes:

Subscripts

Classes, structures, and enumerations can define *subscripts*, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. For example, you access elements in an `Array` instance as `someArray[index]` and elements in a `Dictionary` instance as `someDictionary[key]`.

You can define multiple subscripts for a single type, and the appropriate subscript overload to use is selected based on the type of index value you pass to the subscript. Subscripts are not limited to a single dimension, and you can define subscripts with multiple input parameters to suit your custom type's needs.

Subscript Syntax

Subscripts enable you to query instances of a type by writing one or more values in square brackets after the instance name. Their syntax is similar to both instance method syntax and computed property syntax. You write subscript definitions with the `subscript` keyword, and specify one or more input parameters and a return type, in the same way as instance methods. Unlike instance methods, subscripts can be read-write or read-only. This behavior is communicated by a getter and setter in the same way as for computed properties:

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate subscript value here
    }
    set(newValue) {
        // perform a suitable setting action here
    }
}
```

The type of `newValue` is the same as the return value of the subscript. As with computed properties, you can choose not to specify the setter's `(newValue)` parameter. A default parameter called `newValue` is provided to your setter if you do not provide one yourself.

As with read-only computed properties, you can simplify the declaration of a read-only subscript by removing the `get` keyword and its braces:

```
subscript(index: Int) -> Int {
    // return an appropriate subscript value here
}
```

Here's an example of a read-only subscript implementation, which defines a `TimesTable` structure to represent an *n*-times-table of integers:

```swift
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")
// Prints "six times three is 18"
```

In this example, a new instance of `TimesTable` is created to represent the three-times-table. This is indicated by passing a value of `3` to the structure's `initializer` as the value to use for the instance's `multiplier` parameter.

You can query the `threeTimesTable` instance by calling its subscript, as shown in the call to `threeTimesTable[6]`. This requests the sixth entry in the three-times-table, which returns a value of 18, or 3 times 6.

> NOTE
>
> An *n*-times-table is based on a fixed mathematical rule. It is not appropriate to set `threeTimesTable[someIndex]` to a new value, and so the subscript for `TimesTable` is defined as a read-only subscript.

Subscript Usage

The exact meaning of "subscript" depends on the context in which it is used. Subscripts are typically used as a shortcut for accessing the member elements in a collection, list, or sequence. You are free to implement subscripts in the most appropriate way for your particular class or structure's functionality.

For example, Swift's `Dictionary` type implements a subscript to set and retrieve the values stored in a `Dictionary` instance. You can set a value in a dictionary by providing a key of the dictionary's key type within subscript brackets, and assigning a value of the dictionary's value type to the subscript:

```swift
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

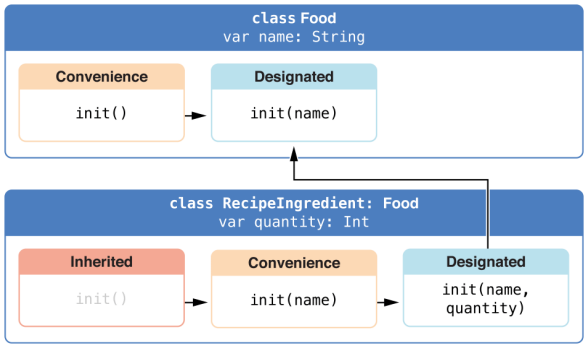
The example above defines a variable called `numberOfLegs` and initializes it with a dictionary literal containing three key-value pairs. The type of the `numberOfLegs` dictionary is inferred to be `[String: Int]`. After creating the dictionary, this example uses subscript assignment to add a `String` key of `"bird"` and an `Int` value of `2` to the dictionary.

For more information about `Dictionary` subscripting, see Accessing and Modifying a Dictionary.

> NOTE
>
> Swift's `Dictionary` type implements its key-value subscripting as a subscript that takes and returns an *optional* type. For the `numberOfLegs` dictionary above, the key-value subscript takes and returns a value of type `Int?`, or "optional int". The `Dictionary` type uses an optional subscript type to model the fact that not every key will have a value, and to give a way to delete a value for a key by assigning a `nil` value for that key.


```swift
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

The figure below shows the initializer chain for the `RecipeIngredient` class:



The `RecipeIngredient` class has a single designated initializer, `init(name: String, quantity: Int)`, which can be used to populate all of the properties of a new `RecipeIngredient` instance. This initializer starts by assigning the passed `quantity` argument to the `quantity` property, which is the only new property introduced by `RecipeIngredient`. After doing so, the initializer delegates up to the `init(name: String)` initializer of the `Food` class. This process satisfies safety check 1 from Two-Phase Initialization above.

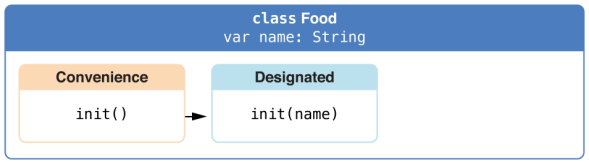
`RecipeIngredient` also defines a convenience initializer, `init(name: String)`, which is used to create a `RecipeIngredient` instance by name alone. This convenience initializer assumes a quantity of 1 for any `RecipeIngredient` instance that is created without an explicit quantity. The definition of this convenience initializer makes `RecipeIngredient` instances quicker and more convenient to create, and avoids code duplication when creating several single-quantity `RecipeIngredient` instances. This convenience initializer simply delegates across to the class's designated initializer, passing in a `quantity` value of 1.

The `init(name: String)` convenience initializer provided by `RecipeIngredient` takes the same parameters as the `init(name: String)` *designated* initializer from `Food`. Because this convenience initializer overrides a designated initializer from its superclass, it must be marked with the `override` modifier (as described in Initializer Inheritance and Overriding).

Even though `RecipeIngredient` provides the `init(name: String)` initializer as a convenience initializer, `RecipeIngredient` has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore, `RecipeIngredient` automatically inherits all of its superclass's convenience initializers too.

```swift
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

The figure below shows the initializer chain for the Food class:



Classes do not have a default memberwise initializer, and so the Food class provides a designated initializer that takes a single argument called name. This initializer can be used to create a new Food instance with a specific name:

```swift
let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"
```

The init(name: String) initializer from the Food class is provided as a *designated* initializer, because it ensures that all stored properties of a new Food instance are fully initialized. The Food class does not have a superclass, and so the init(name: String) initializer does not need to call super.init() to complete its initialization.

The Food class also provides a *convenience* initializer, init(), with no arguments. The init() initializer provides a default placeholder name for a new food by delegating across to the Food class's init(name: String) with a name value of [Unnamed]:

```swift
let mysteryMeat = Food()
// mysteryMeat's name is "[Unnamed]"
```

The second class in the hierarchy is a subclass of Food called RecipeIngredient. The RecipeIngredient class models an ingredient in a cooking recipe. It introduces an Int property called quantity (in addition to the name property it inherits from Food) and defines two initializers for creating RecipeIngredient instances:

Subscript Options

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type. Subscripts can use variadic parameters, but they can't use in-out parameters or provide default parameter values.

A class or structure can provide as many subscript implementations as it needs, and the appropriate subscript to be used will be inferred based on the types of the value or values that are contained within the subscript brackets at the point that the subscript is used. This definition of multiple subscripts is known as *subscript overloading*.

While it is most common for a subscript to take a single parameter, you can also define a subscript with multiple parameters if it is appropriate for your type. The following example defines a Matrix structure, which represents a two-dimensional matrix of Double values. The Matrix structure's subscript takes two integer parameters:

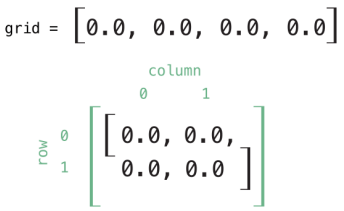
```swift
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of
  range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of
  range")
            grid[(row * columns) + column] = newValue
        }
    }
}
```

Matrix provides an initializer that takes two parameters called rows and columns, and creates an array that is large enough to store rows * columns values of type Double. Each position in the matrix is given an initial value of 0.0. To achieve this, the array's size, and an initial cell value of 0.0, are passed to an array initializer that creates and initializes a new array of the correct size. This initializer is described in more detail in Creating an Array with a Default Value.

You can construct a new Matrix instance by passing an appropriate row and column count to its initializer:

```
var matrix = Matrix(rows: 2, columns: 2)
```

The example above creates a new `Matrix` instance with two rows and two columns. The `grid` array for this `Matrix` instance is effectively a flattened version of the matrix, as read from top left to bottom right:



Values in the matrix can be set by passing row and column values into the subscript, separated by a comma:

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

These two statements call the subscript's setter to set a value of `1.5` in the top right position of the matrix (where `row` is `0` and `column` is `1`), and `3.2` in the bottom left position (where `row` is `1` and `column` is `0`):

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

The `Matrix` subscript's getter and setter both contain an assertion to check that the subscript's `row` and `column` values are valid. To assist with these assertions, `Matrix` includes a convenience method called `indexIsValid(row:column:)`, which checks whether the requested `row` and `column` are inside the bounds of the matrix:

```
func indexIsValid(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}
```

An assertion is triggered if you try to access a subscript that is outside of the matrix bounds:

```
let someValue = matrix[2, 2]
// this triggers an assert, because [2, 2] is outside of the matrix bounds
```


The `init()` initializer for `Bicycle` starts by calling `super.init()`, which calls the default initializer for the `Bicycle` class's superclass, `Vehicle`. This ensures that the `numberOfWheels` inherited property is initialized by `Vehicle` before `Bicycle` has the opportunity to modify the property. After calling `super.init()`, the original value of `numberOfWheels` is replaced with a new value of `2`.

If you create an instance of `Bicycle`, you can call its inherited `description` computed property to see how its `numberOfWheels` property has been updated:

```
let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// Bicycle: 2 wheel(s)
```

> NOTE
>
> Subclasses can modify inherited variable properties during initialization, but can not modify inherited constant properties.

Automatic Initializer Inheritance

As mentioned above, subclasses do not inherit their superclass initializers by default. However, superclass initializers *are* automatically inherited if certain conditions are met. In practice, this means that you do not need to write initializer overrides in many common scenarios, and can inherit your superclass initializers with minimal effort whenever it is safe to do so.

Assuming that you provide default values for any new properties you introduce in a subclass, the following two rules apply:

Rule 1

If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

Rule 2

If your subclass provides an implementation of *all* of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

These rules apply even if your subclass adds further convenience initializers.

> NOTE
>
> A subclass can implement a superclass designated initializer as a subclass convenience initializer as part of satisfying rule 2.

Designated and Convenience Initializers in Action

The following example shows designated initializers, convenience initializers, and automatic initializer inheritance in action. This example defines a hierarchy of three classes called `Food`, `RecipeIngredient`, and `ShoppingListItem`, and demonstrates how their initializers interact.

The base class in the hierarchy is called `Food`, which is a simple class to encapsulate the name of a foodstuff. The `Food` class introduces a single `String` property called `name` and provides two initializers for creating `Food` instances:

As with an overridden property, method or subscript, the presence of the `override` modifier prompts Swift to check that the superclass has a matching designated initializer to be overridden, and validates that the parameters for your overriding initializer have been specified as intended.

> NOTE
>
> You always write the `override` modifier when overriding a superclass designated initializer, even if your subclass's implementation of the initializer is a convenience initializer.

Conversely, if you write a subclass initializer that matches a superclass *convenience* initializer, that superclass convenience initializer can never be called directly by your subclass, as per the rules described above in Initializer Delegation for Class Types. Therefore, your subclass is not (strictly speaking) providing an override of the superclass initializer. As a result, you do not write the `override` modifier when providing a matching implementation of a superclass convenience initializer.

The example below defines a base class called `Vehicle`. This base class declares a stored property called `numberOfWheels`, with a default `Int` value of `0`. The `numberOfWheels` property is used by a computed property called `description` to create a `String` description of the vehicle's characteristics:

```
class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}
```

The `Vehicle` class provides a default value for its only stored property, and does not provide any custom initializers itself. As a result, it automatically receives a default initializer, as described in Default Initializers. The default initializer (when available) is always a designated initializer for a class, and can be used to create a new `Vehicle` instance with a `numberOfWheels` of `0`:

```
let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)
```

The next example defines a subclass of `Vehicle` called `Bicycle`:

```
class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}
```

The `Bicycle` subclass defines a custom designated initializer, `init()`. This designated initializer matches a designated initializer from the superclass of `Bicycle`, and so the `Bicycle` version of this initializer is marked with the `override` modifier.

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 type name 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":

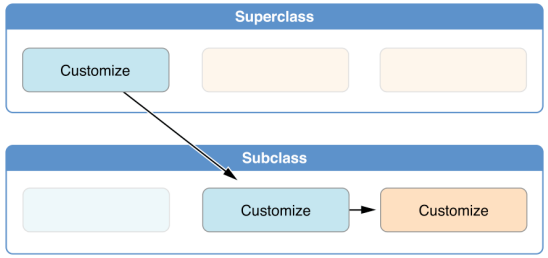
In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer cannot yet modify any properties. It delegates across to a designated initializer from the same class.

The designated initializer makes sure that all of the subclass's properties have a value, as per safety check 1. It then calls a designated initializer on its superclass to continue the initialization up the chain.

The superclass's designated initializer makes sure that all of the superclass properties have a value. There are no further superclasses to initialize, and so no further delegation is needed.

As soon as all properties of the superclass have an initial value, its memory is considered fully initialized, and Phase 1 is complete.

Here's how phase 2 looks for the same initialization call:



The superclass's designated initializer now has an opportunity to customize the instance further (although it does not have to).

Once the superclass's designated initializer is finished, the subclass's designated initializer can perform additional customization (although again, it does not have to).

Finally, once the subclass's designated initializer is finished, the convenience initializer that was originally called can perform additional customization.

Initializer Inheritance and Overriding

Unlike subclasses in Objective-C, Swift subclasses do not inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.

> NOTE
>
> Superclass initializers *are* inherited in certain circumstances, but only when it is safe and appropriate to do so. For more information, see Automatic Initializer Inheritance below.

If you want a custom subclass to present one or more of the same initializers as its superclass, you can provide a custom implementation of those initializers within the subclass.

When you write a subclass initializer that matches a superclass *designated* initializer, you are effectively providing an override of that designated initializer. Therefore, you must write the `override` modifier before the subclass's initializer definition. This is true even if you are overriding an automatically provided default initializer, as described in Default Initializers.

A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

Safety check 3

A convenience initializer must delegate to another initializer before assigning a value to *any* property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

Safety check 4

An initializer cannot call any instance methods, read the values of any instance properties, or refer to `self` as a value until after the first phase of initialization is complete.

The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here's how two-phase initialization plays out, based on the four safety checks above:

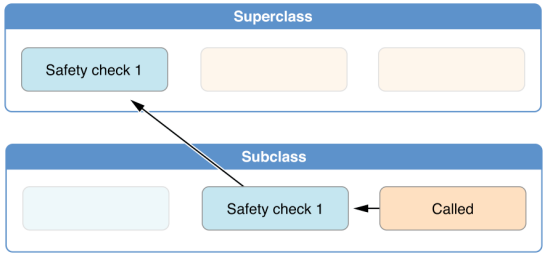
Phase 1

- A designated or convenience initializer is called on a class.
- Memory for a new instance of that class is allocated. The memory is not yet initialized.
- A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized.
- The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties.
- This continues up the class inheritance chain until the top of the chain is reached.
- Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

Phase 2

- Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further. Initializers are now able to access `self` and can modify its properties, call its instance methods, and so on.
- Finally, any convenience initializers in the chain have the option to customize the instance and to work with `self`.

Here's how phase 1 looks for an initialization call for a hypothetical subclass and superclass:




```
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.

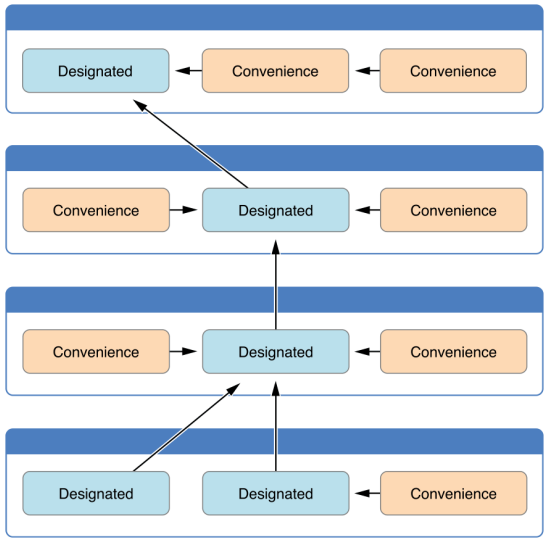
Overriding Property Getters and Setters

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:



Two-Phase Initialization

Class initialization in Swift is a two-phase process. In the first phase, each stored property is assigned an initial value by the class that introduced it. Once the initial state for every stored property has been determined, the second phase begins, and each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use.

The use of a two-phase initialization process makes initialization safe, while still giving complete flexibility to each class in a class hierarchy. Two-phase initialization prevents property values from being accessed before they are initialized, and prevents property values from being set to a different value by another initializer unexpectedly.

> NOTE
>
> Swift's two-phase initialization process is similar to initialization in Objective-C. The main difference is that during phase 1, Objective-C assigns zero or null values (such as `0` or `nil`) to every property. Swift's initialization flow is more flexible in that it lets you set custom initial values, and can cope with types for which `0` or `nil` is not a valid default value.

Swift's compiler performs four helpful safety-checks to make sure that two-phase initialization is completed without error:

Safety check 1

A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.

As mentioned above, the memory for an object is only considered fully initialized once the initial state of all of its stored properties is known. In order for this rule to be satisfied, a designated initializer must make sure that all of its own properties are initialized before it hands off up the chain.

Safety check 2

To simplify the relationships between designated and convenience initializers, Swift applies the following three rules for delegation calls between initializers:

Rule 1

A designated initializer must call a designated initializer from its immediate superclass.

Rule 2

A convenience initializer must call another initializer from the *same* class.

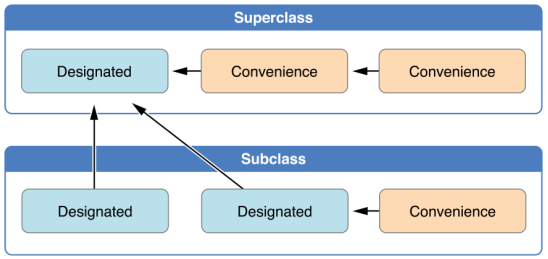
Rule 3

A convenience initializer must ultimately call a designated initializer.

A simple way to remember this is:

- Designated initializers must always delegate *up*.
- Convenience initializers must always delegate *across*.

These rules are illustrated in the figure below:



Here, the superclass has a single designated initializer and two convenience initializers. One convenience initializer calls another convenience initializer, which in turn calls the single designated initializer. This satisfies rules 2 and 3 from above. The superclass does not itself have a further superclass, and so rule 1 does not apply.

The subclass in this figure has two designated initializers and one convenience initializer. The convenience initializer must call one of the two designated initializers, because it can only call another initializer from the same class. This satisfies rules 2 and 3 from above. Both designated initializers must call the single designated initializer from the superclass, to satisfy rule 1 from above.

> NOTE
>
> These rules don't affect how users of your classes *create* instances of each class. Any initializer in the diagram above can be used to create a fully-initialized instance of the class they belong to. The rules only affect how you write the implementation of the class's initializers.

The figure below shows a more complex class hierarchy for four classes. It illustrates how the designated initializers in this hierarchy act as "funnel" points for class initialization, simplifying the interrelationships among classes in the chain:


```
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.

Class Inheritance and Initialization

All of a class's stored properties—including any properties the class inherits from its superclass—*must* be assigned an initial value during initialization.

Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as designated initializers and convenience initializers.

Designated Initializers and Convenience Initializers

Designated initializers are the primary initializers for a class. A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.

Classes tend to have very few designated initializers, and it is quite common for a class to have only one. Designated initializers are "funnel" points through which initialization takes place, and through which the initialization process continues up the superclass chain.

Every class must have at least one designated initializer. In some cases, this requirement is satisfied by inheriting one or more designated initializers from a superclass, as described in Automatic Initializer Inheritance below.

Convenience initializers are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated initializer's parameters set to default values. You can also define a convenience initializer to create an instance of that class for a specific use case or input value type.

You do not have to provide convenience initializers if your class does not require them. Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent.

Syntax for Designated and Convenience Initializers

Designated initializers for classes are written in the same way as simple initializers for value types:

```
init(parameters) {
    statements
}
```

Convenience initializers are written in the same style, but with the `convenience` modifier placed before the `init` keyword, separated by a space:

```
convenience init(parameters) {
    statements
}
```

Initializer Delegation for Class Types

```swift
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

The first `Rect` initializer, `init()`, is functionally the same as the default initializer that the structure would have received if it did not have its own custom initializers. This initializer has an empty body, represented by an empty pair of curly braces `{}`. Calling this initializer returns a `Rect` instance whose `origin` and `size` properties are both initialized with the default values of `Point(x: 0.0, y: 0.0)` and `Size(width: 0.0, height: 0.0)` from their property definitions:

```swift
let basicRect = Rect()
// basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

The second `Rect` initializer, `init(origin:size:)`, is functionally the same as the memberwise initializer that the structure would have received if it did not have its own custom initializers. This initializer simply assigns the `origin` and `size` argument values to the appropriate stored properties:

```swift
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
                      size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

The third `Rect` initializer, `init(center:size:)`, is slightly more complex. It starts by calculating an appropriate origin point based on a `center` point and a `size` value. It then calls (or *delegates*) to the `init(origin:size:)` initializer, which stores the new origin and size values in the appropriate properties:

```swift
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

The `init(center:size:)` initializer could have assigned the new values of `origin` and `size` to the appropriate properties itself. However, it is more convenient (and clearer in intent) for the `init(center:size:)` initializer to take advantage of an existing initializer that already provides exactly that functionality.

> NOTE
>
> For an alternative way to write this example without defining the `init()` and `init(origin:size:)` initializers yourself, see Extensions.

Initialization

Initialization is the process of preparing an instance of a class, structure, or enumeration for use. This process involves setting an initial value for each stored property on that instance and performing any other setup or initialization that is required before the new instance is ready for use.

You implement this initialization process by defining *initializers*, which are like special methods that can be called to create a new instance of a particular type. Unlike Objective-C initializers, Swift initializers do not return a value. Their primary role is to ensure that new instances of a type are correctly initialized before they are used for the first time.

Instances of class types can also implement a *deinitializer*, which performs any custom cleanup just before an instance of that class is deallocated. For more information about deinitializers, see Deinitialization.

Setting Initial Values for Stored Properties

Classes and structures *must* set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created. Stored properties cannot be left in an indeterminate state.

You can set an initial value for a stored property within an initializer, or by assigning a default property value as part of the property's definition. These actions are described in the following sections.

> NOTE
>
> When you assign a default value to a stored property, or set its initial value within an initializer, the value of that property is set directly, without calling any property observers.

Initializers

Initializers are called to create a new instance of a particular type. In its simplest form, an initializer is like an instance method with no parameters, written using the `init` keyword:

```swift
init() {
    // perform some initialization here
}
```

The example below defines a new structure called `Fahrenheit` to store temperatures expressed in the Fahrenheit scale. The `Fahrenheit` structure has one stored property, `temperature`, which is of type `Double`:

```swift
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

The structure defines a single initializer, `init`, with no parameters, which initializes the stored temperature with a value of `32.0` (the freezing point of water in degrees Fahrenheit).

Default Property Values

You can set the initial value of a stored property from within an initializer, as shown above. Alternatively, specify a *default property value* as part of the property's declaration. You specify a default property value by assigning an initial value to the property when it is defined.

> NOTE
>
> If a property always takes the same initial value, provide a default value rather than setting a value within an initializer. The end result is the same, but the default value ties the property's initialization more closely to its declaration. It makes for shorter, clearer initializers and enables you to infer the type of the property from its default value. The default value also makes it easier for you to take advantage of default initializers and initializer inheritance, as described later in this chapter.

You can write the `Fahrenheit` structure from above in a simpler form by providing a default value for its `temperature` property at the point that the property is declared:

```swift
struct Fahrenheit {
    var temperature = 32.0
}
```

Customizing Initialization

You can customize the initialization process with input parameters and optional property types, or by assigning constant properties during initialization, as described in the following sections.

Initialization Parameters

You can provide *initialization parameters* as part of an initializer's definition, to define the types and names of values that customize the initialization process. Initialization parameters have the same capabilities and syntax as function and method parameters.

The following example defines a structure called `Celsius`, which stores temperatures expressed in degrees Celsius. The `Celsius` structure implements two custom initializers called `init(fromFahrenheit:)` and `init(fromKelvin:)`, which initialize a new instance of the structure with a value from a different temperature scale:

The rules for how initializer delegation works, and for what forms of delegation are allowed, are different for value types and class types. Value types (structures and enumerations) do not support inheritance, and so their initializer delegation process is relatively simple, because they can only delegate to another initializer that they provide themselves. Classes, however, can inherit from other classes, as described in Inheritance. This means that classes have additional responsibilities for ensuring that all stored properties they inherit are assigned a suitable value during initialization. These responsibilities are described in Class Inheritance and Initialization below.

For value types, you use `self.init` to refer to other initializers from the same value type when writing your own custom initializers. You can call `self.init` only from within an initializer.

Note that if you define a custom initializer for a value type, you will no longer have access to the default initializer (or the memberwise initializer, if it is a structure) for that type. This constraint prevents a situation in which additional essential setup provided in a more complex initializer is accidentally circumvented by someone using one of the automatic initializers.

> NOTE
>
> If you want your custom value type to be initializable with the default initializer and memberwise initializer, and also with your own custom initializers, write your custom initializers in an extension rather than as part of the value type's original implementation. For more information, see Extensions.

The following example defines a custom `Rect` structure to represent a geometric rectangle. The example requires two supporting structures called `Size` and `Point`, both of which provide default values of `0.0` for all of their properties:

```swift
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

You can initialize the `Rect` structure below in one of three ways—by using its default zero-initialized `origin` and `size` property values, by providing a specific origin point and size, or by providing a specific center point and size. These initialization options are represented by three custom initializers that are part of the `Rect` structure's definition:

Default Initializers

Swift provides a *default initializer* for any structure or class that provides default values for all of its properties and does not provide at least one initializer itself. The default initializer simply creates a new instance with all of its properties set to their default values.

This example defines a class called `ShoppingListItem`, which encapsulates the name, quantity, and purchase state of an item in a shopping list:

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

Because all properties of the `ShoppingListItem` class have default values, and because it is a base class with no superclass, `ShoppingListItem` automatically gains a default initializer implementation that creates a new instance with all of its properties set to their default values. (The `name` property is an optional `String` property, and so it automatically receives a default value of `nil`, even though this value is not written in the code.) The example above uses the default initializer for the `ShoppingListItem` class to create a new instance of the class with initializer syntax, written as `ShoppingListItem()`, and assigns this new instance to a variable called `item`.

Memberwise Initializers for Structure Types

Structure types automatically receive a *memberwise initializer* if they do not define any of their own custom initializers. Unlike a default initializer, the structure receives a memberwise initializer even if it has stored properties that do not have default values.

The memberwise initializer is a shorthand way to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name.

The example below defines a structure called `Size` with two properties called `width` and `height`. Both properties are inferred to be of type `Double` by assigning a default value of `0.0`.

The `Size` structure automatically receives an `init(width:height:)` memberwise initializer, which you can use to initialize a new `Size` instance:

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

Initializer Delegation for Value Types

Initializers can call other initializers to perform part of an instance's initialization. This process, known as *initializer delegation*, avoids duplicating code across multiple initializers.


```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

The first initializer has a single initialization parameter with an argument label of `fromFahrenheit` and a parameter name of `fahrenheit`. The second initializer has a single initialization parameter with an argument label of `fromKelvin` and a parameter name of `kelvin`. Both initializers convert their single argument into the corresponding Celsius value and store this value in a property called `temperatureInCelsius`.

Parameter Names and Argument Labels

As with function and method parameters, initialization parameters can have both a parameter name for use within the initializer's body and an argument label for use when calling the initializer.

However, initializers do not have an identifying function name before their parentheses in the way that functions and methods do. Therefore, the names and types of an initializer's parameters play a particularly important role in identifying which initializer should be called. Because of this, Swift provides an automatic argument label for *every* parameter in an initializer if you don't provide one.

The following example defines a structure called `Color`, with three constant properties called `red`, `green`, and `blue`. These properties store a value between `0.0` and `1.0` to indicate the amount of red, green, and blue in the color.

`Color` provides an initializer with three appropriately named parameters of type `Double` for its red, green, and blue components. `Color` also provides a second initializer with a single `white` parameter, which is used to provide the same value for all three color components.

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}
```

Both initializers can be used to create a new `Color` instance, by providing named values for each initializer parameter:

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)
```

Note that it is not possible to call these initializers without using argument labels. Argument labels must always be used in an initializer if they are defined, and omitting them is a compile-time error:

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - argument labels are required
```

Initializer Parameters Without Argument Labels

If you do not want to use an argument label for an initializer parameter, write an underscore (_) instead of an explicit argument label for that parameter to override the default behavior.

Here's an expanded version of the `Celsius` example from Initialization Parameters above, with an additional initializer to create a new `Celsius` instance from a `Double` value that is already in the Celsius scale:

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}
let bodyTemperature = Celsius(37.0)
// bodyTemperature.temperatureInCelsius is 37.0
```

The initializer call `Celsius(37.0)` is clear in its intent without the need for an argument label. It is therefore appropriate to write this initializer as `init(_ celsius: Double)` so that it can be called by providing an unnamed `Double` value.

Optional Property Types

If your custom type has a stored property that is logically allowed to have "no value"—perhaps because its value cannot be set during initialization, or because it is allowed to have "no value" at some later point—declare the property with an *optional* type. Properties of optional type are automatically initialized with a value of `nil`, indicating that the property is deliberately intended to have "no value yet" during initialization.

The following example defines a class called `SurveyQuestion`, with an optional `String` property called `response`:

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
// Prints "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese."
```

The response to a survey question cannot be known until it is asked, and so the `response` property is declared with a type of `String?`, or "optional `String`". It is automatically assigned a default value of `nil`, meaning "no string yet", when a new instance of `SurveyQuestion` is initialized.

Assigning Constant Properties During Initialization

You can assign a value to a constant property at any point during initialization, as long as it is set to a definite value by the time initialization finishes. Once a constant property is assigned a value, it can't be further modified.

> NOTE
>
> For class instances, a constant property can be modified during initialization only by the class that introduces it. It cannot be modified by a subclass.

You can revise the `SurveyQuestion` example from above to use a constant property rather than a variable property for the `text` property of the question, to indicate that the question does not change once an instance of `SurveyQuestion` is created. Even though the `text` property is now a constant, it can still be set within the class's initializer:

```
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// Prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```