`Rank` also defines a computed property, `values`, which returns an instance of the `Values` structure. This computed property considers the rank of the card and initializes a new `Values` instance with appropriate values based on its rank. It uses special values for `jack`, `queen`, `king`, and `ace`. For the numeric cards, it uses the rank's raw `Int` value.

The `BlackjackCard` structure itself has two properties—`rank` and `suit`. It also defines a computed property called `description`, which uses the values stored in `rank` and `suit` to build a description of the name and value of the card. The `description` property uses optional binding to check whether there is a second value to display, and if so, inserts additional description detail for that second value.

Because `BlackjackCard` is a structure with no custom initializers, it has an implicit memberwise initializer, as described in Memberwise Initializers for Structure Types. You can use this initializer to initialize a new constant called `theAceOfSpades`:

```
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
print("theAceOfSpades: \(theAceOfSpades.description)")
// Prints "theAceOfSpades: suit is ♠, value is 1 or 11"
```

Even though `Rank` and `Suit` are nested within `BlackjackCard`, their type can be inferred from context, and so the initialization of this instance is able to refer to the enumeration cases by their case names (`.ace` and `.spades`) alone. In the example above, the `description` property correctly reports that the Ace of Spades has a value of `1` or `11`.

Referring to Nested Types

To use a nested type outside of its definition context, prefix its name with the name of the type it is nested within:

```
let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
// heartsSymbol is "♡"
```

For the example above, this enables the names of `Suit`, `Rank`, and `Values` to be kept deliberately short, because their names are naturally qualified by the context in which they are defined.

Extensions

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as *retroactive modeling*). Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions do not have names.)

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of. For more details, see Protocol Extensions.

> NOTE
>
> Extensions can add new functionality to a type, but they cannot override existing functionality.

Extension Syntax

Declare extensions with the `extension` keyword:

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

Adding protocol conformance in this way is described in Adding Protocol Conformance with an Extension.

An extension can be used to extend an existing generic type, as described in Extending a Generic Type. You can also extend a generic type to conditionally add functionality, as described in Extensions with a Generic Where Clause.

> NOTE

This operator adds together the `x` values of two vectors, and subtracts the `y` value of the second vector from the first. Because it is in essence an "additive" operator, it has been given the same precedence group as additive infix operators such as + and –. For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations. For more information about precedence groups and to see the syntax for defining your own operators and precedence groups, see Operator Declaration.

> NOTE
>
> You do not specify a precedence when defining a prefix or postfix operator. However, if you apply both a prefix and a postfix operator to the same operand, the postfix operator is applied first.

‹ A Swift Tour

Basic Operators ›

> BETA SOFTWARE
>
> This documentation contains preliminary information about an API or technology in development. This information is subject to change, and software implemented according to this documentation should be tested with final operating system software.
>
> Learn more about using Apple's beta software

> If you define an extension to add new functionality to an existing type, the new functionality will be available on all existing instances of that type, even if they were created before the extension was defined.

Computed Properties

Extensions can add computed instance properties and computed type properties to existing types. This example adds five computed instance properties to Swift's built-in `Double` type, to provide basic support for working with distance units:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

These computed properties express that a `Double` value should be considered as a certain unit of length. Although they are implemented as computed properties, the names of these properties can be appended to a floating-point literal value with dot syntax, as a way to use that literal value to perform distance conversions.

In this example, a `Double` value of `1.0` is considered to represent "one meter". This is why the `m` computed property returns `self`—the expression `1.m` is considered to calculate a `Double` value of `1.0`.

Other units require some conversion to be expressed as a value measured in meters. One kilometer is the same as 1,000 meters, so the `km` computed property multiplies the value by `1_000.00` to convert into a number expressed in meters. Similarly, there are 3.28084 feet in a meter, and so the `ft` computed property divides the underlying `Double` value by `3.28084`, to convert it from feet to meters.

These properties are read-only computed properties, and so they are expressed without the `get` keyword, for brevity. Their return value is of type `Double`, and can be used within mathematical calculations wherever a `Double` is accepted:

```
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// Prints "A marathon is 42195.0 meters long"
```

> NOTE
>
> Extensions can add new computed properties, but they cannot add stored properties, or add property observers to existing properties.

Initializers

Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type's original implementation.

Extensions can add new convenience initializers to a class, but they cannot add new designated initializers or deinitializers to a class. Designated initializers and deinitializers must always be provided by the original class implementation.

If you use an extension to add an initializer to a value type that provides default values for all of its stored properties and does not define any custom initializers, you can call the default initializer and memberwise initializer for that value type from within your extension's initializer. This wouldn't be the case if you had written the initializer as part of the value type's original implementation, as described in Initializer Delegation for Value Types.

If you use an extension to add an initializer to a structure that was declared in another module, the new initializer can't access `self` until it calls an initializer from the defining module.

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

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```

Because the `Rect` structure provides default values for all of its properties, it receives a default initializer and a memberwise initializer automatically, as described in Default Initializers. These initializers can be used to create new `Rect` instances:

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
   size: Size(width: 5.0, height: 5.0))
```

You can extend the `Rect` structure to provide an additional initializer that takes a specific center point and size:

You can declare and implement your own *custom operators* in addition to the standard operators provided by Swift. For a list of characters that can be used to define custom operators, see Operators.

New operators are declared at a global level using the `operator` keyword, and are marked with the `prefix`, `infix` or `postfix` modifiers:

```
prefix operator +++
```

The example above defines a new prefix operator called `+++`. This operator does not have an existing meaning in Swift, and so it is given its own custom meaning below in the specific context of working with `Vector2D` instances. For the purposes of this example, `+++` is treated as a new "prefix doubling" operator. It doubles the `x` and `y` values of a `Vector2D` instance, by adding the vector to itself with the addition assignment operator defined earlier. To implement the `+++` operator, you add a type method called `+++` to `Vector2D` as follows:

```
extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
        return vector
    }
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

Precedence for Custom Infix Operators

Custom infix operators each belong to a precedence group. A precedence group specifies an operator's precedence relative to other infix operators, as well as the operator's associativity. See Precedence and Associativity for an explanation of how these characteristics affect an infix operator's interaction with other infix operators.

A custom infix operator that is not explicitly placed into a precedence group is given a default precedence group with a precedence immediately higher than the precedence of the ternary conditional operator.

The following example defines a new custom infix operator called `+-`, which belongs to the precedence group `AdditionPrecedence`:

```
infix operator +-: AdditionPrecedence
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y - right.y)
    }
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

```
extension Vector2D: Equatable {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) && (left.y == right.y)
    }
}
```

The above example implements an "equal to" operator (==) to check whether two `Vector2D` instances have equivalent values. In the context of `Vector2D`, it makes sense to consider "equal" as meaning "both instances have the same `x` values and `y` values", and so this is the logic used by the operator implementation. If you've implemented an "equal to" operator, you usually don't need to implement a "not equal to" operator (`!=`) yourself. The standard library provides a default implementation of the "not equal to" operator, which simply negates the result of the "equal to" operator that you implemented.

You can now use these operators to check whether two `Vector2D` instances are equivalent:

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
// Prints "These two vectors are equivalent."
```

In many simple cases, you can ask Swift to provide synthesized implementations of the equivalence operators for you. Swift provides synthesized implementations for the following kinds of custom types:

- Structures that have only stored properties that conform to the `Equatable` protocol
- Enumerations that have only associated types that conform to the `Equatable` protocol
- Enumerations that have no associated types

Declare `Equatable` conformance in the file that contains the original declaration to receive these implementations.

The example below defines a `Vector3D` structure for a three-dimensional position vector `(x, y, z)`, similar to the `Vector2D` structure. Because the `x`, `y`, and `z` properties are all of an `Equatable` type, `Vector3D` receives default implementations of the equivalence operators.

```
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}

let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("These two vectors are also equivalent.")
}
// Prints "These two vectors are also equivalent."
```

Custom Operators


```
extension Rect {
    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)
    }
}
```

This new initializer starts by calculating an appropriate origin point based on the provided `center` point and `size` value. The initializer then calls the structure's automatic memberwise initializer `init(origin:size:)`, which stores the new origin and size values in the appropriate properties:

```
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)
```

> NOTE
>
> If you provide a new initializer with an extension, you are still responsible for making sure that each instance is fully initialized once the initializer completes.

Methods

Extensions can add new instance methods and type methods to existing types. The following example adds a new instance method called `repetitions` to the `Int` type:

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}
```

The `repetitions(task:)` method takes a single argument of type `() -> Void`, which indicates a function that has no parameters and does not return a value.

After defining this extension, you can call the `repetitions(task:)` method on any integer to perform a task that many number of times:

```
3.repetitions {
    print("Hello!")
}
// Hello!
// Hello!
// Hello!
```

Mutating Instance Methods

Instance methods added with an extension can also modify (or *mutate*) the instance itself. Structure and enumeration methods that modify `self` or its properties must mark the instance method as `mutating`, just like mutating methods from an original implementation.

The example below adds a new mutating method called `square` to Swift's `Int` type, which squares the original value:

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt is now 9
```

Subscripts

Extensions can add new subscripts to an existing type. This example adds an integer subscript to Swift's built-in `Int` type. This subscript `[n]` returns the decimal digit `n` places in from the right of the number:

- `123456789[0]` returns 9
- `123456789[1]` returns 8

...and so on:

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

If the `Int` value does not have enough digits for the requested index, the subscript implementation returns `0`, as if the number had been padded with zeros to the left:

```
746381295[9]
// returns 0, as if you had requested:
0746381295[9]
```


For simple numeric values, the unary minus operator converts positive numbers into their negative equivalent and vice versa. The corresponding implementation for `Vector2D` instances performs this operation on both the `x` and `y` properties:

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

Compound Assignment Operators

Compound assignment operators combine assignment (=) with another operation. For example, the addition assignment operator (+=) combines addition and assignment into a single operation. You mark a compound assignment operator's left input parameter type as `inout`, because the parameter's value will be modified directly from within the operator method.

The example below implements an addition assignment operator method for `Vector2D` instances:

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}
```

Because an addition operator was defined earlier, you don't need to reimplement the addition process here. Instead, the addition assignment operator method takes advantage of the existing addition operator method, and uses it to set the left value to be the left value plus the right value:

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

> NOTE
>
> It is not possible to overload the default assignment operator (=). Only the compound assignment operators can be overloaded. Similarly, the ternary conditional operator (`a ? b : c`) cannot be overloaded.

Equivalence Operators

By default, custom classes and structures don't receive a default implementation of the *equivalence operators*, known as the *equal to* operator (==) and *not equal to* operator (!=).

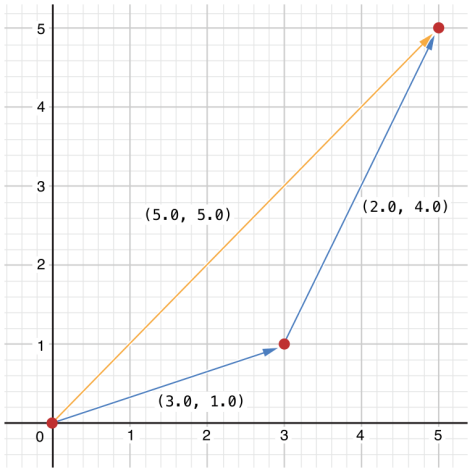
To use the equivalence operators to check for equivalence of your own custom type, provide an implementation of the "equal to" operator in the same way as for other infix operators, and add conformance to the standard library's `Equatable` protocol:

In this implementation, the input parameters are named `left` and `right` to represent the `Vector2D` instances that will be on the left side and right side of the `+` operator. The method returns a new `Vector2D` instance, whose `x` and `y` properties are initialized with the sum of the `x` and `y` properties from the two `Vector2D` instances that are added together.

The type method can be used as an infix operator between existing `Vector2D` instances:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

This example adds together the vectors `(3.0, 1.0)` and `(2.0, 4.0)` to make the vector `(5.0, 5.0)`, as illustrated below.



Prefix and Postfix Operators

The example shown above demonstrates a custom implementation of a binary infix operator. Classes and structures can also provide implementations of the standard *unary operators*. Unary operators operate on a single target. They are *prefix* if they precede their target (such as `–a`) and *postfix* operators if they follow their target (such as `b!`).

You implement a prefix or postfix unary operator by writing the `prefix` or `postfix` modifier before the `func` keyword when declaring the operator method:

```
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
}
```

The example above implements the unary minus operator (`–a`) for `Vector2D` instances. The unary minus operator is a prefix operator, and so this method has to be qualified with the `prefix` modifier.

Nested Types

Extensions can add new nested types to existing classes, structures, and enumerations:

```
extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}
```

This example adds a new nested enumeration to `Int`. This enumeration, called `Kind`, expresses the kind of number that a particular integer represents. Specifically, it expresses whether the number is negative, zero, or positive.

This example also adds a new computed instance property to `Int`, called `kind`, which returns the appropriate `Kind` enumeration case for that integer.

The nested enumeration can now be used with any `Int` value:

```
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Prints "+ + - 0 - 0 + "
```

This function, `printIntegerKinds(_:)`, takes an input array of `Int` values and iterates over those values in turn. For each integer in the array, the function considers the `kind` computed property for that integer, and prints an appropriate description.

NOTE

`number.kind` is already known to be of type `Int.Kind`. Because of this, all of the `Int.Kind` case values can be written in shorthand form inside the `switch` statement, such as `.negative` rather than `Int.Kind.negative`.

`(3 % 4)` is 3, so this is equivalent to:

```
2 + (3 * 5)
```

`(3 * 5)` is 15, so this is equivalent to:

```
2 + 15
```

This calculation yields the final answer of 17.

For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations.

> NOTE
>
> Swift's operator precedences and associativity rules are simpler and more predictable than those found in C and Objective-C. However, this means that they are not exactly the same as in C-based languages. Be careful to ensure that operator interactions still behave in the way you intend when porting existing code to Swift.

Operator Methods

Classes and structures can provide their own implementations of existing operators. This is known as *overloading* the existing operators.

The example below shows how to implement the arithmetic addition operator (+) for a custom structure. The arithmetic addition operator is a *binary operator* because it operates on two targets and is said to be *infix* because it appears in between those two targets.

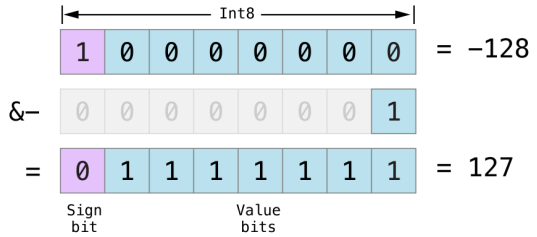
The example defines a `Vector2D` structure for a two-dimensional position vector `(x, y)`, followed by a definition of an *operator method* to add together instances of the `Vector2D` structure:

```
struct Vector2D {
    var x = 0.0, y = 0.0
}

extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y + right.y)
    }
}
```

The operator method is defined as a type method on `Vector2D`, with a method name that matches the operator to be overloaded (+). Because addition isn't part of the essential behavior for a vector, the type method is defined in an extension of `Vector2D` rather than in the main structure declaration of `Vector2D`. Because the arithmetic addition operator is a binary operator, this operator method takes two input parameters of type `Vector2D` and returns a single output value, also of type `Vector2D`.

The minimum value that an `Int8` can hold is `-128`, or `10000000` in binary. Subtracting `1` from this binary number with the overflow operator gives a binary value of `01111111`, which toggles the sign bit and gives positive `127`, the maximum positive value that an `Int8` can hold.



For both signed and unsigned integers, overflow in the positive direction wraps around from the maximum valid integer value back to the minimum, and overflow in the negative direction wraps around from the minimum value to the maximum.

Precedence and Associativity

Operator *precedence* gives some operators higher priority than others; these operators are applied first.

Operator *associativity* defines how operators of the same precedence are grouped together—either grouped from the left, or grouped from the right. Think of it as meaning "they associate with the expression to their left," or "they associate with the expression to their right."

It is important to consider each operator's precedence and associativity when working out the order in which a compound expression will be calculated. For example, operator precedence explains why the following expression equals 17.

```
2 + 3 % 4 * 5
// this equals 17
```

If you read strictly from left to right, you might expect the expression to be calculated as follows:

- 2 plus 3 equals 5
- 5 remainder 4 equals 1
- 1 times 5 equals 5

However, the actual answer is 17, not 5. Higher-precedence operators are evaluated before lower-precedence ones. In Swift, as in C, the remainder operator (`%`) and the multiplication operator (`*`) have a higher precedence than the addition operator (`+`). As a result, they are both evaluated before the addition is considered.

However, remainder and multiplication have the *same* precedence as each other. To work out the exact evaluation order to use, you also need to consider their associativity. Remainder and multiplication both associate with the expression to their left. Think of this as adding implicit parentheses around these parts of the expression, starting from their left:

```
2 + ((3 % 4) * 5)
```


Protocols

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

Protocol Syntax

You define protocols in a very similar way to classes, structures, and enumerations:

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

Property Requirements

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn't specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable *and* settable.

If a protocol requires a property to be gettable and settable, that property requirement can't be fulfilled by a constant stored property or a read-only computed property. If the protocol only requires a property to be gettable, the requirement can be satisfied by any kind of property, and it's valid for the property to be also settable if this is useful for your own code.

Property requirements are always declared as variable properties, prefixed with the `var` keyword. Gettable and settable properties are indicated by writing `{ get set }` after their type declaration, and gettable properties are indicated by writing `{ get }`.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Always prefix type property requirements with the `static` keyword when you define them in a protocol. This rule pertains even though type property requirements can be prefixed with the `class` or `static` keyword when implemented by a class:

```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

Here's an example of a protocol with a single instance property requirement:

```
protocol FullyNamed {
    var fullName: String { get }
}
```

The `FullyNamed` protocol requires a conforming type to provide a fully-qualified name. The protocol doesn't specify anything else about the nature of the conforming type—it only specifies that the type must be able to provide a full name for itself. The protocol states that any `FullyNamed` type must have a gettable instance property called `fullName`, which is of type `String`.

Here's an example of a simple structure that adopts and conforms to the `FullyNamed` protocol:

```
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

This example defines a structure called `Person`, which represents a specific named person. It states that it adopts the `FullyNamed` protocol as part of the first line of its definition.

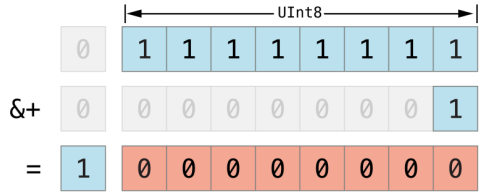
Each instance of `Person` has a single stored property called `fullName`, which is of type `String`. This matches the single requirement of the `FullyNamed` protocol, and means that `Person` has correctly conformed to the protocol. (Swift reports an error at compile-time if a protocol requirement is not fulfilled.)

Here's a more complex class, which also adopts and conforms to the `FullyNamed` protocol:

Here's an example of what happens when an unsigned integer is allowed to overflow in the positive direction, using the overflow addition operator (&+):

```
var unsignedOverflow = UInt8.max
// unsignedOverflow equals 255, which is the maximum value a UInt8 can
  hold
unsignedOverflow = unsignedOverflow &+ 1
// unsignedOverflow is now equal to 0
```

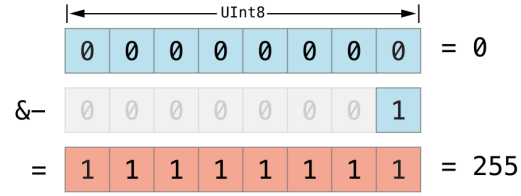
The variable `unsignedOverflow` is initialized with the maximum value a `UInt8` can hold (`255`, or `11111111` in binary). It is then incremented by `1` using the overflow addition operator (&+). This pushes its binary representation just over the size that a `UInt8` can hold, causing it to overflow beyond its bounds, as shown in the diagram below. The value that remains within the bounds of the `UInt8` after the overflow addition is `00000000`, or zero.



Something similar happens when an unsigned integer is allowed to overflow in the negative direction. Here's an example using the overflow subtraction operator (&–):

```
var unsignedOverflow = UInt8.min
// unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1
// unsignedOverflow is now equal to 255
```

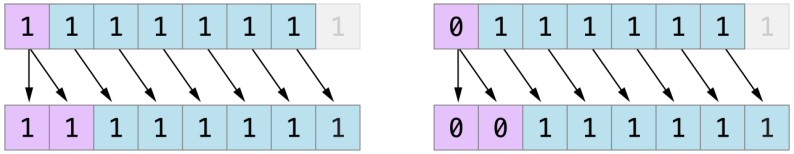
The minimum value that a `UInt8` can hold is zero, or `00000000` in binary. If you subtract `1` from `00000000` using the overflow subtraction operator (&–), the number will overflow and wrap around to `11111111`, or `255` in decimal.



Overflow also occurs for signed integers. All addition and subtraction for signed integers is performed in bitwise fashion, with the sign bit included as part of the numbers being added or subtracted, as described in Bitwise Left and Right Shift Operators.

```
var signedOverflow = Int8.min
// signedOverflow equals -128, which is the minimum value an Int8 can hold
signedOverflow = signedOverflow &- 1
// signedOverflow is now equal to 127
```

Second, the two's complement representation also lets you shift the bits of negative numbers to the left and right like positive numbers, and still end up doubling them for every shift you make to the left, or halving them for every shift you make to the right. To achieve this, an extra rule is used when signed integers are shifted to the right: When you shift signed integers to the right, apply the same rules as for unsigned integers, but fill any empty bits on the left with the *sign bit*, rather than with a zero.



This action ensures that signed integers have the same sign after they are shifted to the right, and is known as an *arithmetic shift*.

Because of the special way that positive and negative numbers are stored, shifting either of them to the right moves them closer to zero. Keeping the sign bit the same during this shift means that negative integers remain negative as their value moves closer to zero.

Overflow Operators

If you try to insert a number into an integer constant or variable that cannot hold that value, by default Swift reports an error rather than allowing an invalid value to be created. This behavior gives extra safety when you work with numbers that are too large or too small.

For example, the `Int16` integer type can hold any signed integer between `–32768` and `32767`. Trying to set an `Int16` constant or variable to a number outside of this range causes an error:

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the maximum value an Int16 can
  hold
potentialOverflow += 1
// this causes an error
```

Providing error handling when values get too large or too small gives you much more flexibility when coding for boundary value conditions.

However, when you specifically want an overflow condition to truncate the number of available bits, you can opt in to this behavior rather than triggering an error. Swift provides three arithmetic *overflow operators* that opt in to the overflow behavior for integer calculations. These operators all begin with an ampersand (`&`):

- Overflow addition (`&+`)
- Overflow subtraction (`&–`)
- Overflow multiplication (`&*`)

Value Overflow

Numbers can overflow in both the positive and negative direction.


```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

This class implements the `fullName` property requirement as a computed read-only property for a starship. Each `Starship` class instance stores a mandatory `name` and an optional `prefix`. The `fullName` property uses the `prefix` value if it exists, and prepends it to the beginning of `name` to create a full name for the starship.

Method Requirements

Protocols can require specific instance methods and type methods to be implemented by conforming types. These methods are written as part of the protocol's definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods. Default values, however, can't be specified for method parameters within a protocol's definition.

As with type property requirements, you always prefix type method requirements with the `static` keyword when they're defined in a protocol. This is true even though type method requirements are prefixed with the `class` or `static` keyword when implemented by a class:

```
protocol SomeProtocol {
    static func someTypeMethod()
}
```

The following example defines a protocol with a single instance method requirement:

```
protocol RandomNumberGenerator {
    func random() -> Double
}
```

This protocol, `RandomNumberGenerator`, requires any conforming type to have an instance method called `random`, which returns a `Double` value whenever it's called. Although it's not specified as part of the protocol, it's assumed that this value will be a number from `0.0` up to (but not including) `1.0`.

The `RandomNumberGenerator` protocol doesn't make any assumptions about how each random number will be generated—it simply requires the generator to provide a standard way to generate a new random number.

Here's an implementation of a class that adopts and conforms to the `RandomNumberGenerator` protocol. This class implements a pseudorandom number generator algorithm known as a *linear congruential generator*:

```
class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a +
  c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.3746499199817101"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283"
```

Mutating Method Requirements

It's sometimes necessary for a method to modify (or *mutate*) the instance it belongs to. For instance methods on value types (that is, structures and enumerations) you place the `mutating` keyword before a method's `func` keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance. This process is described in Modifying Value Types from Within Instance Methods.

If you define a protocol instance method requirement that is intended to mutate instances of any type that adopts the protocol, mark the method with the `mutating` keyword as part of the protocol's definition. This enables structures and enumerations to adopt the protocol and satisfy that method requirement.

> NOTE
>
> If you mark a protocol instance method requirement as `mutating`, you don't need to write the `mutating` keyword when writing an implementation of that method for a class. The `mutating` keyword is only used by structures and enumerations.

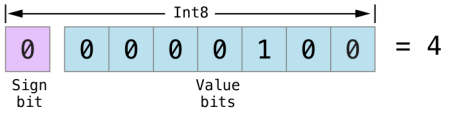
The example below defines a protocol called `Togglable`, which defines a single instance method requirement called `toggle`. As its name suggests, the `toggle()` method is intended to toggle or invert the state of any conforming type, typically by modifying a property of that type.

The `toggle()` method is marked with the `mutating` keyword as part of the `Togglable` protocol definition, to indicate that the method is expected to mutate the state of a conforming instance when it's called:

```
protocol Togglable {
    mutating func toggle()
}
```


Signed integers use their first bit (known as the *sign bit*) to indicate whether the integer is positive or negative. A sign bit of `0` means positive, and a sign bit of `1` means negative.

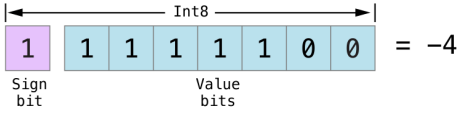
The remaining bits (known as the *value bits*) store the actual value. Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from `0`. Here's how the bits inside an `Int8` look for the number `4`:



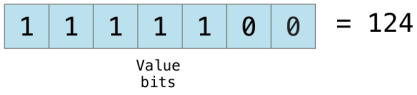
The sign bit is `0` (meaning "positive"), and the seven value bits are just the number `4`, written in binary notation.

Negative numbers, however, are stored differently. They are stored by subtracting their absolute value from `2` to the power of `n`, where `n` is the number of value bits. An eight-bit number has seven value bits, so this means `2` to the power of `7`, or `128`.

Here's how the bits inside an `Int8` look for the number `-4`:

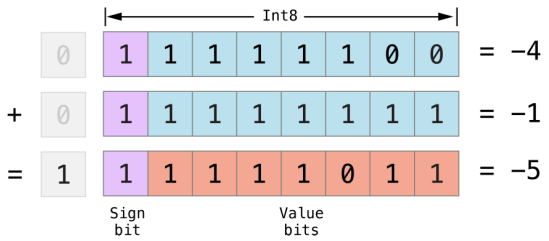


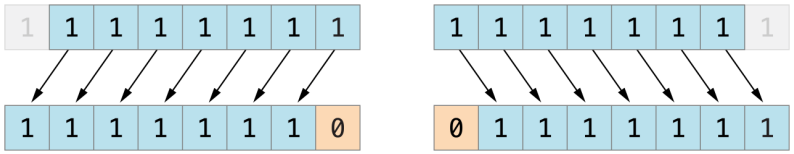
This time, the sign bit is `1` (meaning "negative"), and the seven value bits have a binary value of `124` (which is `128 - 4`):



This encoding for negative numbers is known as a *two's complement* representation. It may seem an unusual way to represent negative numbers, but it has several advantages.

First, you can add `-1` to `-4`, simply by performing a standard binary addition of all eight bits (including the sign bit), and discarding anything that doesn't fit in the eight bits once you're done:





Here's how bit shifting looks in Swift code:

```
let shiftBits: UInt8 = 4   // 00000100 in binary
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

You can use bit shifting to encode and decode values within other data types:

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent is 0xCC, or 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent is 0x66, or 102
let blueComponent = pink & 0x0000FF           // blueComponent is 0x99, or 153
```

This example uses a `UInt32` constant called `pink` to store a Cascading Style Sheets color value for the color pink. The CSS color value `#CC6699` is written as `0xCC6699` in Swift's hexadecimal number representation. This color is then decomposed into its red (`CC`), green (`66`), and blue (`99`) components by the bitwise AND operator (`&`) and the bitwise right shift operator (`>>`).

The red component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0xFF0000`. The zeros in `0xFF0000` effectively "mask" the second and third bytes of `0xCC6699`, causing the `6699` to be ignored and leaving `0xCC0000` as the result.

This number is then shifted 16 places to the right (`>> 16`). Each pair of characters in a hexadecimal number uses 8 bits, so a move 16 places to the right will convert `0xCC0000` into `0x0000CC`. This is the same as `0xCC`, which has a decimal value of `204`.

Similarly, the green component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0x00FF00`, which gives an output value of `0x006600`. This output value is then shifted eight places to the right, giving a value of `0x66`, which has a decimal value of `102`.

Finally, the blue component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0x0000FF`, which gives an output value of `0x000099`. There's no need to shift this to the right, as `0x000099` already equals `0x99`, which has a decimal value of `153`.

Shifting Behavior for Signed Integers

The shifting behavior is more complex for signed integers than for unsigned integers, because of the way signed integers are represented in binary. (The examples below are based on 8-bit signed integers for simplicity, but the same principles apply for signed integers of any size.)

If you implement the `Togglable` protocol for a structure or enumeration, that structure or enumeration can conform to the protocol by providing an implementation of the `toggle()` method that is also marked as `mutating`.

The example below defines an enumeration called `OnOffSwitch`. This enumeration toggles between two states, indicated by the enumeration cases `on` and `off`. The enumeration's `toggle` implementation is marked as `mutating`, to match the `Togglable` protocol's requirements:

```
enum OnOffSwitch: Togglable {
    case off, on
    mutating func toggle() {
        switch self {
        case .off:
            self = .on
        case .on:
            self = .off
        }
    }
}
var lightSwitch = OnOffSwitch.off
lightSwitch.toggle()
// lightSwitch is now equal to .on
```

Initializer Requirements

Protocols can require specific initializers to be implemented by conforming types. You write these initializers as part of the protocol's definition in exactly the same way as for normal initializers, but without curly braces or an initializer body:

```
protocol SomeProtocol {
    init(someParameter: Int)
}
```

Class Implementations of Protocol Initializer Requirements

You can implement a protocol initializer requirement on a conforming class as either a designated initializer or a convenience initializer. In both cases, you must mark the initializer implementation with the `required` modifier:

```
class SomeClass: SomeProtocol {
    required init(someParameter: Int) {
        // initializer implementation goes here
    }
}
```

The use of the `required` modifier ensures that you provide an explicit or inherited implementation of the initializer requirement on all subclasses of the conforming class, such that they also conform to the protocol.

For more information on required initializers, see Required Initializers.

NOTE

You don't need to mark protocol initializer implementations with the `required` modifier on classes that are marked with the `final` modifier, because final classes can't subclassed. For more about the `final` modifier, see Preventing Overrides.

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the `required` and `override` modifiers:

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from
  SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```

Failable Initializer Requirements

Protocols can define failable initializer requirements for conforming types, as defined in Failable Initializers.

A failable initializer requirement can be satisfied by a failable or nonfailable initializer on a conforming type. A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer.

Protocols as Types

Protocols don't actually implement any functionality themselves. Nonetheless, any protocol you create will become a fully-fledged type for use in your code.

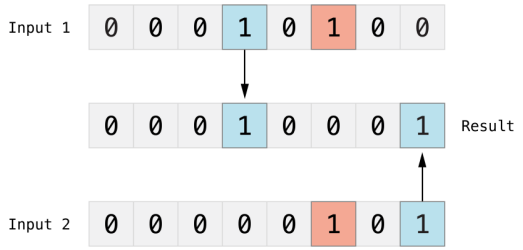
Because it's a type, you can use a protocol in many places where other types are allowed, including:

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

NOTE

Bitwise XOR Operator

The *bitwise XOR operator*, or "exclusive OR operator" (^), compares the bits of two numbers. The operator returns a new number whose bits are set to `1` where the input bits are different and are set to `0` where the input bits are the same:



In the example below, the values of `firstBits` and `otherBits` each have a bit set to `1` in a location that the other does not. The bitwise XOR operator sets both of these bits to `1` in its output value. All of the other bits in `firstBits` and `otherBits` match and are set to `0` in the output value:

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

Bitwise Left and Right Shift Operators

The *bitwise left shift operator* (<<) and *bitwise right shift operator* (>>) move all bits in a number to the left or the right by a certain number of places, according to the rules defined below.

Bitwise left and right shifts have the effect of multiplying or dividing an integer by a factor of two. Shifting an integer's bits to the left by one position doubles its value, whereas shifting it to the right by one position halves its value.

Shifting Behavior for Unsigned Integers

The bit-shifting behavior for unsigned integers is as follows:

1. Existing bits are moved to the left or right by the requested number of places.
2. Any bits that are moved beyond the bounds of the integer's storage are discarded.
3. Zeros are inserted in the spaces left behind after the original bits are moved to the left or right.

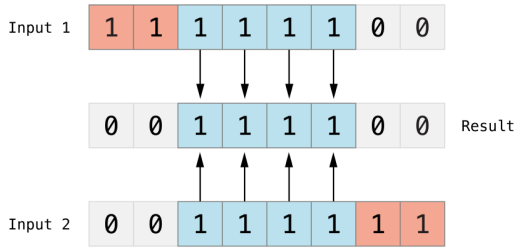
This approach is known as a *logical shift*.

The illustration below shows the results of `11111111 << 1` (which is `11111111` shifted to the left by `1` place), and `11111111 >> 1` (which is `11111111` shifted to the right by `1` place). Blue numbers are shifted, gray numbers are discarded, and orange zeros are inserted:

The bitwise NOT operator is then used to create a new constant called `invertedBits`, which is equal to `initialBits`, but with all of the bits inverted. Zeros become ones, and ones become zeros. The value of `invertedBits` is `11110000`, which is equal to an unsigned decimal value of `240`.

Bitwise AND Operator

The *bitwise AND operator* (`&`) combines the bits of two numbers. It returns a new number whose bits are set to `1` only if the bits were equal to `1` in *both* input numbers:

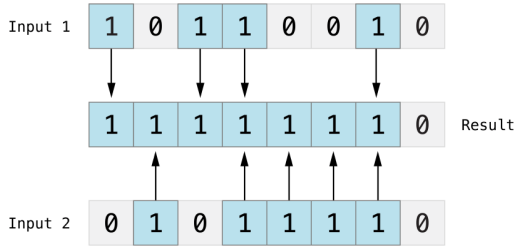


In the example below, the values of `firstSixBits` and `lastSixBits` both have four middle bits equal to `1`. The bitwise AND operator combines them to make the number `00111100`, which is equal to an unsigned decimal value of `60`:

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100
```

Bitwise OR Operator

The *bitwise OR operator* (`|`) compares the bits of two numbers. The operator returns a new number whose bits are set to `1` if the bits are equal to `1` in *either* input number:



In the example below, the values of `someBits` and `moreBits` have different bits set to `1`. The bitwise OR operator combines them to make the number `11111110`, which equals an unsigned decimal of `254`:

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110
```


> Because protocols are types, begin their names with a capital letter (such as `FullyNamed` and `RandomNumberGenerator`) to match the names of other types in Swift (such as `Int`, `String`, and `Double`).

Here's an example of a protocol used as a type:

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

This example defines a new class called `Dice`, which represents an *n*-sided dice for use in a board game. `Dice` instances have an integer property called `sides`, which represents how many sides they have, and a property called `generator`, which provides a random number generator from which to create dice roll values.

The `generator` property is of type `RandomNumberGenerator`. Therefore, you can set it to an instance of *any* type that adopts the `RandomNumberGenerator` protocol. Nothing else is required of the instance you assign to this property, except that the instance must adopt the `RandomNumberGenerator` protocol.

`Dice` also has an initializer, to set up its initial state. This initializer has a parameter called `generator`, which is also of type `RandomNumberGenerator`. You can pass a value of any conforming type in to this parameter when initializing a new `Dice` instance.

`Dice` provides one instance method, `roll`, which returns an integer value between 1 and the number of sides on the dice. This method calls the generator's `random()` method to create a new random number between `0.0` and `1.0`, and uses this random number to create a dice roll value within the correct range. Because `generator` is known to adopt `RandomNumberGenerator`, it's guaranteed to have a `random()` method to call.

Here's how the `Dice` class can be used to create a six-sided dice with a `LinearCongruentialGenerator` instance as its random number generator:

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

Delegation

Delegation is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type. This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated. Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

The example below defines two protocols for use with dice-based board games:

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

The `DiceGame` protocol is a protocol that can be adopted by any game that involves dice.

The `DiceGameDelegate` protocol can be adopted to track the progress of a `DiceGame`. To prevent strong reference cycles, delegates are declared as weak references. For information about weak references, see Strong Reference Cycles Between Class Instances. Marking the protocol as class-only lets the `SnakesAndLadders` class later in this chapter declare that its delegate must use a weak reference. A class-only protocol is marked by its inheritance from `AnyObject` as discussed in Class-Only Protocols.

Here's a version of the *Snakes and Ladders* game originally introduced in Control Flow. This version is adapted to use a `Dice` instance for its dice-rolls; to adopt the `DiceGame` protocol; and to notify a `DiceGameDelegate` about its progress:

Advanced Operators

In addition to the operators described in Basic Operators, Swift provides several advanced operators that perform more complex value manipulation. These include all of the bitwise and bit shifting operators you will be familiar with from C and Objective-C.

Unlike arithmetic operators in C, arithmetic operators in Swift do not overflow by default. Overflow behavior is trapped and reported as an error. To opt in to overflow behavior, use Swift's second set of arithmetic operators that overflow by default, such as the overflow addition operator (`&+`). All of these overflow operators begin with an ampersand (`&`).

When you define your own structures, classes, and enumerations, it can be useful to provide your own implementations of the standard Swift operators for these custom types. Swift makes it easy to provide tailored implementations of these operators and to determine exactly what their behavior should be for each type you create.

You're not limited to the predefined operators. Swift gives you the freedom to define your own custom infix, prefix, postfix, and assignment operators, with custom precedence and associativity values. These operators can be used and adopted in your code like any of the predefined operators, and you can even extend existing types to support the custom operators you define.

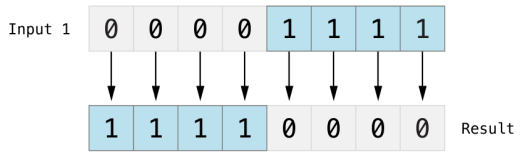
Bitwise Operators

Bitwise operators enable you to manipulate the individual raw data bits within a data structure. They are often used in low-level programming, such as graphics programming and device driver creation. Bitwise operators can also be useful when you work with raw data from external sources, such as encoding and decoding data for communication over a custom protocol.

Swift supports all of the bitwise operators found in C, as described below.

Bitwise NOT Operator

The *bitwise NOT operator* (`~`) inverts all bits in a number:



The bitwise NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space:

```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

`UInt8` integers have eight bits and can store any value between `0` and `255`. This example initializes a `UInt8` integer with the binary value `00001111`, which has its first four bits set to `0`, and its second four bits set to `1`. This is equivalent to a decimal value of `15`.

```
protocol SomeProtocol {
    func doSomething()
}
```

You can use an extension to add protocol conformance, like this:

```
struct SomeStruct {
    private var privateVariable = 12
}

extension SomeStruct: SomeProtocol {
    func doSomething() {
        print(privateVariable)
    }
}
```

Generics

The access level for a generic type or generic function is the minimum of the access level of the generic type or function itself and the access level of any type constraints on its type parameters.

Type Aliases

Any type aliases you define are treated as distinct types for the purposes of access control. A type alias can have an access level less than or equal to the access level of the type it aliases. For example, a private type alias can alias a private, file-private, internal, public, or open type, but a public type alias can't alias an internal, file-private, or private type.

> NOTE
>
> This rule also applies to type aliases for associated types used to satisfy protocol conformances.


```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    weak var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

For a description of the *Snakes and Ladders* gameplay, see Break.

This version of the game is wrapped up as a class called `SnakesAndLadders`, which adopts the `DiceGame` protocol. It provides a gettable `dice` property and a `play()` method in order to conform to the protocol. (The `dice` property is declared as a constant property because it doesn't need to change after initialization, and the protocol only requires that it must be gettable.)

The *Snakes and Ladders* game board setup takes place within the class's `init()` initializer. All game logic is moved into the protocol's `play` method, which uses the protocol's required `dice` property to provide its dice roll values.

Note that the `delegate` property is defined as an *optional* `DiceGameDelegate`, because a delegate isn't required in order to play the game. Because it's of an optional type, the `delegate` property is automatically set to an initial value of `nil`. Thereafter, the game instantiator has the option to set the property to a suitable delegate. Because the `DiceGameDelegate` protocol is class-only, you can declare the delegate to be `weak` to prevent reference cycles.

`DiceGameDelegate` provides three methods for tracking the progress of a game. These three methods have been incorporated into the game logic within the `play()` method above, and are called when a new game starts, a new turn begins, or the game ends.

Because the `delegate` property is an *optional* `DiceGameDelegate`, the `play()` method uses optional chaining each time it calls a method on the delegate. If the `delegate` property is nil, these delegate calls fail gracefully and without error. If the `delegate` property is non-nil, the delegate methods are called, and are passed the `SnakesAndLadders` instance as a parameter.

This next example shows a class called `DiceGameTracker`, which adopts the `DiceGameDelegate` protocol:

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
  {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

`DiceGameTracker` implements all three methods required by `DiceGameDelegate`. It uses these methods to keep track of the number of turns a game has taken. It resets a `numberOfTurns` property to zero when the game starts, increments it each time a new turn begins, and prints out the total number of turns once the game has ended.

The implementation of `gameDidStart(_:)` shown above uses the `game` parameter to print some introductory information about the game that is about to be played. The `game` parameter has a type of `DiceGame`, not `SnakesAndLadders`, and so `gameDidStart(_:)` can access and use only methods and properties that are implemented as part of the `DiceGame` protocol. However, the method is still able to use type casting to query the type of the underlying instance. In this example, it checks whether `game` is actually an instance of `SnakesAndLadders` behind the scenes, and prints an appropriate message if so.

The `gameDidStart(_:)` method also accesses the `dice` property of the passed `game` parameter. Because `game` is known to conform to the `DiceGame` protocol, it's guaranteed to have a `dice` property, and so the `gameDidStart(_:)` method is able to access and print the dice's `sides` property, regardless of what kind of game is being played.

Here's how `DiceGameTracker` looks in action:

The context in which a type conforms to a particular protocol is the minimum of the type's access level and the protocol's access level. If a type is public, but a protocol it conforms to is internal, the type's conformance to that protocol is also internal.

When you write or extend a type to conform to a protocol, you must ensure that the type's implementation of each protocol requirement has at least the same access level as the type's conformance to that protocol. For example, if a public type conforms to an internal protocol, the type's implementation of each protocol requirement must be at least "internal".

> NOTE
>
> In Swift, as in Objective-C, protocol conformance is global—it isn't possible for a type to conform to a protocol in two different ways within the same program.

Extensions

You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available. Any type members added in an extension have the same default access level as type members declared in the original type being extended. If you extend a public or internal type, any new type members you add have a default access level of internal. If you extend a file-private type, any new type members you add have a default access level of file private. If you extend a private type, any new type members you add have a default access level of private.

Alternatively, you can mark an extension with an explicit access-level modifier (for example, `private extension`) to set a new default access level for all members defined within the extension. This new default can still be overridden within the extension for individual type members.

You can't provide an explicit access-level modifier for an extension if you're using that extension to add protocol conformance. Instead, the protocol's own access level is used to provide the default access level for each protocol requirement implementation within the extension.

Private Members in Extensions

Extensions that are in the same file as the class, structure, or enumeration that they extend behave as if the code in the extension had been written as part of the original type's declaration. As a result, you can:

- Declare a private member in the original declaration, and access that member from extensions in the same file.
- Declare a private member in one extension, and access that member from another extension in the same file.
- Declare a private member in an extension, and access that member from the original declaration in the same file.

This behavior means you can use extensions in the same way to organize your code, whether or not your types have private entities. For example, given the following simple protocol:

Default Initializers

As described in Default Initializers, Swift automatically provides a *default initializer* without any arguments for any structure or base class that provides default values for all of its properties and doesn't provide at least one initializer itself.

A default initializer has the same access level as the type it initializes, unless that type is defined as `public`. For a type that is defined as `public`, the default initializer is considered internal. If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type's definition.

Default Memberwise Initializers for Structure Types

The default memberwise initializer for a structure type is considered private if any of the structure's stored properties are private. Likewise, if any of the structure's stored properties are file private, the initializer is file private. Otherwise, the initializer has an access level of internal.

As with the default initializer above, if you want a public structure type to be initializable with a memberwise initializer when used in another module, you must provide a public memberwise initializer yourself as part of the type's definition.

Protocols

If you want to assign an explicit access level to a protocol type, do so at the point that you define the protocol. This enables you to create protocols that can only be adopted within a certain access context.

The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol. You can't set a protocol requirement to a different access level than the protocol it supports. This ensures that all of the protocol's requirements will be visible on any type that adopts the protocol.

> NOTE
>
> If you define a public protocol, the protocol's requirements require a public access level for those requirements when they're implemented. This behavior is different from other types, where a public type definition implies an access level of internal for the type's members.

Protocol Inheritance

If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. You can't write a public protocol that inherits from an internal protocol, for example.

Protocol Conformance

A type can conform to a protocol with a lower access level than the type itself. For example, you can define a public type that can be used in other modules, but whose conformance to an internal protocol can only be used within the internal protocol's defining module.


```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
// Rolled a 5
// The game lasted for 4 turns
```

Adding Protocol Conformance with an Extension

You can extend an existing type to adopt and conform to a new protocol, even if you don't have access to the source code for the existing type. Extensions can add new properties, methods, and subscripts to an existing type, and are therefore able to add any requirements that a protocol may demand. For more about extensions, see Extensions.

> NOTE
>
> Existing instances of a type automatically adopt and conform to a protocol when that conformance is added to the instance's type in an extension.

For example, this protocol, called `TextRepresentable`, can be implemented by any type that has a way to be represented as text. This might be a description of itself, or a text version of its current state:

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

The `Dice` class from above can be extended to adopt and conform to `TextRepresentable`:

```
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
```

This extension adopts the new protocol in exactly the same way as if `Dice` had provided it in its original implementation. The protocol name is provided after the type name, separated by a colon, and an implementation of all requirements of the protocol is provided within the extension's curly braces.

Any `Dice` instance can now be treated as `TextRepresentable`:

```
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
print(d12.textualDescription)
// Prints "A 12-sided dice"
```

Similarly, the `SnakesAndLadders` game class can be extended to adopt and conform to the `TextRepresentable` protocol:

```
extension SnakesAndLadders: TextRepresentable {
    var textualDescription: String {
        return "A game of Snakes and Ladders with \(finalSquare) squares"
    }
}
print(game.textualDescription)
// Prints "A game of Snakes and Ladders with 25 squares"
```

Conditionally Conforming to a Protocol

A generic type may be able to satisfy the requirements of a protocol only under certain conditions, such as when the type's generic parameter conforms to the protocol. You can make a generic type conditionally conform to a protocol by listing constraints when extending the type. Write these constraints after the name of the protocol you're adopting by writing a generic `where` clause. For more about generic `where` clauses, see Generic Where Clauses.

The following extension makes `Array` instances conform to the `TextRepresentable` protocol whenever they store elements of a type that conforms to `TextRepresentable`.

```
extension Array: TextRepresentable where Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
let myDice = [d6, d12]
print(myDice.textualDescription)
// Prints "[A 6-sided dice, A 12-sided dice]"
```

Declaring Protocol Adoption with an Extension

If a type already conforms to all of the requirements of a protocol, but has not yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Instances of `Hamster` can now be used wherever `TextRepresentable` is the required type:

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Prints "A hamster named Simon"
```


The `TrackedString` structure and the `value` property don't provide an explicit access-level modifier, and so they both receive the default access level of internal. However, the access level for the `numberOfEdits` property is marked with a `private(set)` modifier to indicate that the property's getter still has the default access level of internal, but the property is settable only from within code that's part of the `TrackedString` structure. This enables `TrackedString` to modify the `numberOfEdits` property internally, but to present the property as a read-only property when it's used outside the structure's definition.

If you create a `TrackedString` instance and modify its string value a few times, you can see the `numberOfEdits` property value update to match the number of modifications:

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfEdits."
stringToEdit.value += " So will this one."
print("The number of edits is \(stringToEdit.numberOfEdits)")
// Prints "The number of edits is 3"
```

Although you can query the current value of the `numberOfEdits` property from within another source file, you can't *modify* the property from another source file. This restriction protects the implementation details of the `TrackedString` edit-tracking functionality, while still providing convenient access to an aspect of that functionality.

Note that you can assign an explicit access level for both a getter and a setter if required. The example below shows a version of the `TrackedString` structure in which the structure is defined with an explicit access level of public. The structure's members (including the `numberOfEdits` property) therefore have an internal access level by default. You can make the structure's `numberOfEdits` property getter public, and its property setter private, by combining the `public` and `private(set)` access-level modifiers:

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

Initializers

Custom initializers can be assigned an access level less than or equal to the type that they initialize. The only exception is for required initializers (as defined in Required Initializers). A required initializer must have the same access level as the class it belongs to.

As with function and method parameters, the types of an initializer's parameters can't be more private than the initializer's own access level.

Constants, Variables, Properties, and Subscripts

A constant, variable, or property can't be more public than its type. It's not valid to write a public property with a private type, for example. Similarly, a subscript can't be more public than either its index type or return type.

If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as `private`:

```
private var privateInstance = SomePrivateClass()
```

Getters and Setters

Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.

You can give a setter a *lower* access level than its corresponding getter, to restrict the read-write scope of that variable, property, or subscript. You assign a lower access level by writing `fileprivate(set)`, `private(set)`, or `internal(set)` before the `var` or `subscript` introducer.

> NOTE
>
> This rule applies to stored properties as well as computed properties. Even though you don't write an explicit getter and setter for a stored property, Swift still synthesizes an implicit getter and setter for you to provide access to the stored property's backing storage. Use `fileprivate(set)`, `private(set)`, and `internal(set)` to change the access level of this synthesized setter in exactly the same way as for an explicit setter in a computed property.

The example below defines a structure called `TrackedString`, which keeps track of the number of times a string property is modified:

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
}
```

The `TrackedString` structure defines a stored string property called `value`, with an initial value of `""` (an empty string). The structure also defines a stored integer property called `numberOfEdits`, which is used to track the number of times that `value` is modified. This modification tracking is implemented with a `didSet` property observer on the `value` property, which increments `numberOfEdits` every time the `value` property is set to a new value.

> NOTE
>
> Types don't automatically adopt a protocol just by satisfying its requirements. They must always explicitly declare their adoption of the protocol.

Collections of Protocol Types

A protocol can be used as the type to be stored in a collection such as an array or a dictionary, as mentioned in Protocols as Types. This example creates an array of `TextRepresentable` things:

```
let things: [TextRepresentable] = [game, d12, simonTheHamster]
```

It's now possible to iterate over the items in the array, and print each item's textual description:

```
for thing in things {
    print(thing.textualDescription)
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

Note that the `thing` constant is of type `TextRepresentable`. It's not of type `Dice`, or `DiceGame`, or `Hamster`, even if the actual instance behind the scenes is of one of those types. Nonetheless, because it's of type `TextRepresentable`, and anything that is `TextRepresentable` is known to have a `textualDescription` property, it's safe to access `thing.textualDescription` each time through the loop.

Protocol Inheritance

A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

Here's an example of a protocol that inherits the `TextRepresentable` protocol from above:

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

This example defines a new protocol, `PrettyTextRepresentable`, which inherits from `TextRepresentable`. Anything that adopts `PrettyTextRepresentable` must satisfy all of the requirements enforced by `TextRepresentable`, *plus* the additional requirements enforced by `PrettyTextRepresentable`. In this example, `PrettyTextRepresentable` adds a single requirement to provide a gettable property called `prettyTextualDescription` that returns a `String`.

The `SnakesAndLadders` class can be extended to adopt and conform to `PrettyTextRepresentable`:

```
extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
            case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

This extension states that it adopts the `PrettyTextRepresentable` protocol and provides an implementation of the `prettyTextualDescription` property for the `SnakesAndLadders` type. Anything that is `PrettyTextRepresentable` must also be `TextRepresentable`, and so the implementation of `prettyTextualDescription` starts by accessing the `textualDescription` property from the `TextRepresentable` protocol to begin an output string. It appends a colon and a line break, and uses this as the start of its pretty text representation. It then iterates through the array of board squares, and appends a geometric shape to represent the contents of each square:

- If the square's value is greater than `0`, it's the base of a ladder, and is represented by ▲.
- If the square's value is less than `0`, it's the head of a snake, and is represented by ▼.
- Otherwise, the square's value is `0`, and it's a "free" square, represented by ○.

The `prettyTextualDescription` property can now be used to print a pretty text description of any `SnakesAndLadders` instance:

```
print(game.prettyTextualDescription)
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the `AnyObject` protocol to a protocol's inheritance list.

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```


Nested Types

Nested types defined within a private type have an automatic access level of private. Nested types defined within a file-private type have an automatic access level of file private. Nested types defined within a public type or an internal type have an automatic access level of internal. If you want a nested type within a public type to be publicly available, you must explicitly declare the nested type as public.

Subclassing

You can subclass any class that can be accessed in the current access context. A subclass can't have a higher access level than its superclass—for example, you can't write a public subclass of an internal superclass.

In addition, you can override any class member (method, property, initializer, or subscript) that is visible in a certain access context.

An override can make an inherited class member more accessible than its superclass version. In the example below, class `A` is a public class with a file-private method called `someMethod()`. Class `B` is a subclass of `A`, with a reduced access level of "internal". Nonetheless, class `B` provides an override of `someMethod()` with an access level of "internal", which is *higher* than the original implementation of `someMethod()`:

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {}
}
```

It's even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass's member takes place within an allowed access level context (that is, within the same source file as the superclass for a file-private member call, or within the same module as the superclass for an internal member call):

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {
        super.someMethod()
    }
}
```

Because superclass `A` and subclass `B` are defined in the same source file, it's valid for the `B` implementation of `someMethod()` to call `super.someMethod()`.

The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type. You must specify the access level explicitly as part of the function's definition if the function's calculated access level doesn't match the contextual default.

The example below defines a global function called `someFunction()`, without providing a specific access-level modifier for the function itself. You might expect this function to have the default access level of "internal", but this isn't the case. In fact, `someFunction()` won't compile as written below:

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

The function's return type is a tuple type composed from two of the custom classes defined above in Custom Types. One of these classes is defined as internal, and the other is defined as private. Therefore, the overall access level of the compound tuple type is private (the minimum access level of the tuple's constituent types).

Because the function's return type is private, you must mark the function's overall access level with the `private` modifier for the function declaration to be valid:

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

It's not valid to mark the definition of `someFunction()` with the `public` or `internal` modifiers, or to use the default setting of internal, because public or internal users of the function might not have appropriate access to the private class used in the function's return type.

Enumeration Types

The individual cases of an enumeration automatically receive the same access level as the enumeration they belong to. You can't specify a different access level for individual enumeration cases.

In the example below, the `CompassPoint` enumeration has an explicit access level of public. The enumeration cases `north`, `south`, `east`, and `west` therefore also have an access level of public:

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

Raw Values and Associated Values

The types used for any raw values or associated values in an enumeration definition must have an access level at least as high as the enumeration's access level. You can't use a private type as the raw-value type of an enumeration with an internal access level, for example.

In the example above, `SomeClassOnlyProtocol` can only be adopted by class types. It's a compile-time error to write a structure or enumeration definition that tries to adopt `SomeClassOnlyProtocol`.

> NOTE
>
> Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics. For more about reference and value semantics, see Structures and Enumerations Are Value Types and Classes Are Reference Types.

Protocol Composition

It can be useful to require a type to conform to multiple protocols at the same time. You can combine multiple protocols into a single requirement with a *protocol composition*. Protocol compositions behave as if you defined a temporary local protocol that has the combined requirements of all protocols in the composition. Protocol compositions don't define any new protocol types.

Protocol compositions have the form `SomeProtocol & AnotherProtocol`. You can list as many protocols as you need, separating them with ampersands (`&`). In addition to its list of protocols, a protocol composition can also contain one class type, which you can use to specify a required superclass.

Here's an example that combines two protocols called `Named` and `Aged` into a single protocol composition requirement on a function parameter:

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

In this example, the `Named` protocol has a single requirement for a gettable `String` property called `name`. The `Aged` protocol has a single requirement for a gettable `Int` property called `age`. Both protocols are adopted by a structure called `Person`.

The example also defines a `wishHappyBirthday(to:)` function. The type of the `celebrator` parameter is `Named & Aged`, which means "any type that conforms to both the `Named` and `Aged` protocols." It doesn't matter which specific type is passed to the function, as long as it conforms to both of the required protocols.

The example then creates a new `Person` instance called `birthdayPerson` and passes this new instance to the `wishHappyBirthday(to:)` function. Because `Person` conforms to both protocols, this call is valid, and the `wishHappyBirthday(to:)` function can print its birthday greeting.

Here's an example that combines the `Named` protocol from the previous example with a `Location` class:

```
class Location {
    var latitude: Double
    var longitude: Double
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }
}
class City: Location, Named {
    var name: String
    init(name: String, latitude: Double, longitude: Double) {
        self.name = name
        super.init(latitude: latitude, longitude: longitude)
    }
}
func beginConcert(in location: Location & Named) {
    print("Hello, \(location.name)!")
}

let seattle = City(name: "Seattle", latitude: 47.6, longitude: -122.3)
beginConcert(in: seattle)
// Prints "Hello, Seattle!"
```

The `beginConcert(in:)` function takes a parameter of type `Location & Named`, which means "any type that's a subclass of `Location` and that conforms to the `Named` protocol." In this case, `City` satisfies both requirements.

Passing `birthdayPerson` to the `beginConcert(in:)` function is invalid because `Person` isn't a subclass of `Location`. Likewise, if you made a subclass of `Location` that didn't conform to the `Named` protocol, calling `beginConcert(in:)` with an instance of that type is also invalid.

Checking for Protocol Conformance

You can use the `is` and `as` operators described in Type Casting to check for protocol conformance, and to cast to a specific protocol. Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a type:

- The `is` operator returns `true` if an instance conforms to a protocol and returns `false` if it doesn't.


```
public class SomePublicClass {                  // explicitly public class
    public var somePublicProperty = 0            // explicitly public
  class member
    var someInternalProperty = 0                 // implicitly internal
  class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-
  private class member
    private func somePrivateMethod() {}          // explicitly private
  class member
}

class SomeInternalClass {                       // implicitly internal
  class
    var someInternalProperty = 0                 // implicitly internal
  class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-
  private class member
    private func somePrivateMethod() {}          // explicitly private
  class member
}

fileprivate class SomeFilePrivateClass {        // explicitly file-private
  class
    func someFilePrivateMethod() {}              // implicitly file-
  private class member
    private func somePrivateMethod() {}          // explicitly private
  class member
}

private class SomePrivateClass {                // explicitly private
  class
    func somePrivateMethod() {}                  // implicitly private
  class member
}
```

Tuple Types

The access level for a tuple type is the most restrictive access level of all types used in that tuple. For example, if you compose a tuple from two different types, one with internal access and one with private access, the access level for that compound tuple type will be private.

> NOTE
>
> Tuple types don't have a standalone definition in the way that classes, structures, enumerations, and functions do. A tuple type's access level is deduced automatically when the tuple type is used, and can't be specified explicitly.

Function Types

Custom Types

If you want to specify an explicit access level for a custom type, do so at the point that you define the type. The new type can then be used wherever its access level permits. For example, if you define a file-private class, that class can only be used as the type of a property, or as a function parameter or return type, in the source file in which the file-private class is defined.

The access control level of a type also affects the default access level of that type's *members* (its properties, methods, initializers, and subscripts). If you define a type's access level as private or file private, the default access level of its members will also be private or file private. If you define a type's access level as internal or public (or use the default access level of internal without specifying an access level explicitly), the default access level of the type's members will be internal.

> IMPORTANT
>
> A public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such. This requirement ensures that the public-facing API for a type is something you opt in to publishing, and avoids presenting the internal workings of a type as public API by mistake.

- The `as?` version of the downcast operator returns an optional value of the protocol's type, and this value is `nil` if the instance doesn't conform to that protocol.
- The `as!` version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn't succeed.

This example defines a protocol called `HasArea`, with a single property requirement of a gettable `Double` property called `area`:

```
protocol HasArea {
    var area: Double { get }
}
```

Here are two classes, `Circle` and `Country`, both of which conform to the `HasArea` protocol:

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

The `Circle` class implements the `area` property requirement as a computed property, based on a stored `radius` property. The `Country` class implements the `area` requirement directly as a stored property. Both classes correctly conform to the `HasArea` protocol.

Here's a class called `Animal`, which doesn't conform to the `HasArea` protocol:

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

The `Circle`, `Country` and `Animal` classes don't have a shared base class. Nonetheless, they're all classes, and so instances of all three types can be used to initialize an array that stores values of type `AnyObject`:

```
let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

The `objects` array is initialized with an array literal containing a `Circle` instance with a radius of 2 units; a `Country` instance initialized with the surface area of the United Kingdom in square kilometers; and an `Animal` instance with four legs.

The `objects` array can now be iterated, and each object in the array can be checked to see if it conforms to the `HasArea` protocol:

```swift
for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area
```

Whenever an object in the array conforms to the `HasArea` protocol, the optional value returned by the `as?` operator is unwrapped with optional binding into a constant called `objectWithArea`. The `objectWithArea` constant is known to be of type `HasArea`, and so its `area` property can be accessed and printed in a type-safe way.

Note that the underlying objects aren't changed by the casting process. They continue to be a `Circle`, a `Country` and an `Animal`. However, at the point that they're stored in the `objectWithArea` constant, they're only known to be of type `HasArea`, and so only their `area` property can be accessed.

Optional Protocol Requirements

You can define *optional requirements* for protocols, These requirements don't have to be implemented by types that conform to the protocol. Optional requirements are prefixed by the `optional` modifier as part of the protocol's definition. Optional requirements are available so that you can write code that interoperates with Objective-C. Both the protocol and the optional requirement must be marked with the `@objc` attribute. Note that `@objc` protocols can be adopted only by classes that inherit from Objective-C classes or other `@objc` classes. They can't be adopted by structures or enumerations.

When you use a method or property in an optional requirement, its type automatically becomes an optional. For example, a method of type `(Int) -> String` becomes `((Int) -> String)?`. Note that the entire function type is wrapped in the optional, not the method's return value.

An optional protocol requirement can be called with optional chaining, to account for the possibility that the requirement was not implemented by a type that conforms to the protocol. You check for an implementation of an optional method by writing a question mark after the name of the method when it's called, such as `someOptionalMethod?(someArgument)`. For information on optional chaining, see Optional Chaining.

The following example defines an integer-counting class called `Counter`, which uses an external data source to provide its increment amount. This data source is defined by the `CounterDataSource` protocol, which has two optional requirements:

```swift
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
```


When you write a simple single-target app, the code in your app is typically self-contained within the app and doesn't need to be made available outside of the app's module. The default access level of internal already matches this requirement. Therefore, you don't need to specify a custom access level. You may, however, want to mark some parts of your code as file private or private in order to hide their implementation details from other code within the app's module.

Access Levels for Frameworks

When you develop a framework, mark the public-facing interface to that framework as open or public so that it can be viewed and accessed by other modules, such as an app that imports the framework. This public-facing interface is the application programming interface (or API) for the framework.

> NOTE
>
> Any internal implementation details of your framework can still use the default access level of internal, or can be marked as private or file private if you want to hide them from other parts of the framework's internal code. You need to mark an entity as open or public only if you want it to become part of your framework's API.

Access Levels for Unit Test Targets

When you write an app with a unit test target, the code in your app needs to be made available to that module in order to be tested. By default, only entities marked as open or public are accessible to other modules. However, a unit test target can access any internal entity, if you mark the import declaration for a product module with the `@testable` attribute and compile that product module with testing enabled.

Access Control Syntax

Define the access level for an entity by placing one of the `open`, `public`, `internal`, `fileprivate`, or `private` modifiers before the entity's introducer:

```swift
public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
fileprivate func someFilePrivateFunction() {}
private func somePrivateFunction() {}
```

Unless otherwise specified, the default access level is internal, as described in Default Access Levels. This means that `SomeInternalClass` and `someInternalConstant` can be written without an explicit access-level modifier, and will still have an access level of internal:

```swift
class SomeInternalClass {}              // implicitly internal
let someInternalConstant = 0            // implicitly internal
```

defining an app's or a framework's internal structure.
- *File-private access* restricts the use of an entity to its own defining source file. Use file-private access to hide the implementation details of a specific piece of functionality when those details are used within an entire file.
- *Private access* restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file. Use private access to hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.

Open access is the highest (least restrictive) access level and private access is the lowest (most restrictive) access level.

Open access applies only to classes and class members, and it differs from public access as follows:

- Classes with public access, or any more restrictive access level, can be subclassed only within the module where they're defined.
- Class members with public access, or any more restrictive access level, can be overridden by subclasses only within the module where they're defined.
- Open classes can be subclassed within the module where they're defined, and within any module that imports the module where they're defined.
- Open class members can be overridden by subclasses within the module where they're defined, and within any module that imports the module where they're defined.

Marking a class as open explicitly indicates that you've considered the impact of code from other modules using that class as a superclass, and that you've designed your class's code accordingly.

Guiding Principle of Access Levels

Access levels in Swift follow an overall guiding principle: *No entity can be defined in terms of another entity that has a lower (more restrictive) access level.*

For example:

- A public variable can't be defined as having an internal, file-private, or private type, because the type might not be available everywhere that the public variable is used.
- A function can't have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are unavailable to the surrounding code.

The specific implications of this guiding principle for different aspects of the language are covered in detail below.

Default Access Levels

All entities in your code (with a few specific exceptions, as described later in this chapter) have a default access level of internal if you don't specify an explicit access level yourself. As a result, in many cases you don't need to specify an explicit access level in your code.

Access Levels for Single-Target Apps

The `CounterDataSource` protocol defines an optional method requirement called `increment(forCount:)` and an optional property requirement called `fixedIncrement`. These requirements define two different ways for data sources to provide an appropriate increment amount for a `Counter` instance.

> NOTE
>
> Strictly speaking, you can write a custom class that conforms to `CounterDataSource` without implementing *either* protocol requirement. They're both optional, after all. Although technically allowed, this wouldn't make for a very good data source.

The `Counter` class, defined below, has an optional `dataSource` property of type `CounterDataSource?`:

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount: count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

The `Counter` class stores its current value in a variable property called `count`. The `Counter` class also defines a method called `increment`, which increments the `count` property every time the method is called.

The `increment()` method first tries to retrieve an increment amount by looking for an implementation of the `increment(forCount:)` method on its data source. The `increment()` method uses optional chaining to try to call `increment(forCount:)`, and passes the current `count` value as the method's single argument.

Note that *two* levels of optional chaining are at play here. First, it's possible that `dataSource` may be `nil`, and so `dataSource` has a question mark after its name to indicate that `increment(forCount:)` should be called only if `dataSource` isn't `nil`. Second, even if `dataSource` *does* exist, there's no guarantee that it implements `increment(forCount:)`, because it's an optional requirement. Here, the possibility that `increment(forCount:)` might not be implemented is also handled by optional chaining. The call to `increment(forCount:)` happens only if `increment(forCount:)` exists—that is, if it isn't `nil`. This is why `increment(forCount:)` is also written with a question mark after its name.

Because the call to `increment(forCount:)` can fail for either of these two reasons, the call returns an *optional* `Int` value. This is true even though `increment(forCount:)` is defined as returning a nonoptional `Int` value in the definition of `CounterDataSource`. Even though there are two optional chaining operations, one after another, the result is still wrapped in a single optional. For more information about using multiple optional chaining operations, see Linking Multiple Levels of Chaining.

After calling `increment(forCount:)`, the optional `Int` that it returns is unwrapped into a constant called `amount`, using optional binding. If the optional `Int` does contain a value—that is, if the delegate and method both exist, and the method returned a value—the unwrapped `amount` is added onto the stored `count` property, and incrementation is complete.

If it's *not* possible to retrieve a value from the `increment(forCount:)` method—either because `dataSource` is nil, or because the data source doesn't implement `increment(forCount:)`—then the `increment()` method tries to retrieve a value from the data source's `fixedIncrement` property instead. The `fixedIncrement` property is also an optional requirement, so its value is an optional `Int` value, even though `fixedIncrement` is defined as a nonoptional `Int` property as part of the `CounterDataSource` protocol definition.

Here's a simple `CounterDataSource` implementation where the data source returns a constant value of `3` every time it's queried. It does this by implementing the optional `fixedIncrement` property requirement:

```
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
```

You can use an instance of `ThreeSource` as the data source for a new `Counter` instance:

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    print(counter.count)
}
// 3
// 6
// 9
// 12
```

The code above creates a new `Counter` instance; sets its data source to be a new `ThreeSource` instance; and calls the counter's `increment()` method four times. As expected, the counter's `count` property increases by three each time `increment()` is called.

Here's a more complex data source called `TowardsZeroSource`, which makes a `Counter` instance count up or down towards zero from its current `count` value:

```
class TowardsZeroSource: NSObject, CounterDataSource {
    func increment(forCount count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```


Access Control

Access control restricts access to parts of your code from code in other source files and modules. This feature enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

You can assign specific access levels to individual types (classes, structures, and enumerations), as well as to properties, methods, initializers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.

In addition to offering various levels of access control, Swift reduces the need to specify explicit access control levels by providing default access levels for typical scenarios. Indeed, if you are writing a single-target app, you may not need to specify explicit access control levels at all.

> NOTE
>
> The various aspects of your code that can have access control applied to them (properties, types, functions, and so on) are referred to as "entities" in the sections below, for brevity.

Modules and Source Files

Swift's access control model is based on the concept of modules and source files.

A *module* is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift's `import` keyword.

Each build target (such as an app bundle or framework) in Xcode is treated as a separate module in Swift. If you group together aspects of your app's code as a stand-alone framework—perhaps to encapsulate and reuse that code across multiple applications—then everything you define within that framework will be part of a separate module when it's imported and used within an app, or when it's used within another framework.

A *source file* is a single Swift source code file within a module (in effect, a single file within an app or framework). Although it's common to define individual types in separate source files, a single source file can contain definitions for multiple types, functions, and so on.

Access Levels

Swift provides five different *access levels* for entities within your code. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

- *Open access* and *public access* enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use open or public access when specifying the public interface to a framework. The difference between open and public access is described below.
- *Internal access* enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when

```
var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health, &playerInformation.energy)
// Error: conflicting access to properties of playerInformation
```

In the example above, calling `balance(_:_:)` on the elements of a tuple produces a conflict because there are overlapping write accesses to `playerInformation`. Both `playerInformation.health` and `playerInformation.energy` are passed as in-out parameters, which means `balance(_:_:)` needs write access to them for the duration of the function call. In both cases, a write access to the tuple element requires a write access to the entire tuple. This means there are two write accesses to `playerInformation` with durations that overlap, causing a conflict.

The code below shows that the same error appears for overlapping write accesses to the properties of a structure that's stored in a global variable.

```
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy)  // Error
```

In practice, most access to the properties of a structure can overlap safely. For example, if the variable `holly` in the example above is changed to a local variable instead of a global variable, the compiler can prove that overlapping access to stored properties of the structure is safe:

```
func someFunction() {
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
    balance(&oscar.health, &oscar.energy)  // OK
}
```

In the example above, Oscar's health and energy are passed as the two in-out parameters to `balance(_:_:)`. The compiler can prove that memory safety is preserved because the two stored properties don't interact in any way.

The restriction against overlapping access to properties of a structure isn't always necessary to preserve memory safety. Memory safety is the desired guarantee, but exclusive access is a stricter requirement than memory safety—which means some code preserves memory safety, even though it violates exclusive access to memory. Swift allows this memory-safe code if the compiler can prove that the nonexclusive access to memory is still safe. Specifically, it can prove that overlapping access to properties of a structure is safe if the following conditions apply:

- You're accessing only stored properties of an instance, not computed properties or class properties.
- The structure is the value of a local variable, not a global variable.
- The structure is either not captured by any closures, or it's captured only by nonescaping closures.

If the compiler can't prove the access is safe, it doesn't allow the access.

The `TowardsZeroSource` class implements the optional `increment(forCount:)` method from the `CounterDataSource` protocol and uses the `count` argument value to work out which direction to count in. If `count` is already zero, the method returns `0` to indicate that no further counting should take place.

You can use an instance of `TowardsZeroSource` with the existing `Counter` instance to count from `-4` to zero. Once the counter reaches zero, no more counting takes place:

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    print(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

Protocol Extensions

Protocols can be extended to provide method, initializer, subscript, and computed property implementations to conforming types. This allows you to define behavior on protocols themselves, rather than in each type's individual conformance or in a global function.

For example, the `RandomNumberGenerator` protocol can be extended to provide a `randomBool()` method, which uses the result of the required `random()` method to return a random `Bool` value:

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

By creating an extension on the protocol, all conforming types automatically gain this method implementation without any additional modification.

```
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.3746499199817101"
print("And here's a random Boolean: \(generator.randomBool())")
// Prints "And here's a random Boolean: true"
```

Protocol extensions can add implementations to conforming types but can't make a protocol extend or inherit from another protocol. Protocol inheritance is always specified in the protocol declaration itself.

Providing Default Implementations

You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol. If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

> NOTE
>
> Protocol requirements with default implementations provided by extensions are distinct from optional protocol requirements. Although conforming types don't have to provide their own implementation of either, requirements with default implementations can be called without optional chaining.

For example, the `PrettyTextRepresentable` protocol, which inherits the `TextRepresentable` protocol can provide a default implementation of its required `prettyTextualDescription` property to simply return the result of accessing the `textualDescription` property:

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

Adding Constraints to Protocol Extensions

When you define a protocol extension, you can specify constraints that conforming types must satisfy before the methods and properties of the extension are available. You write these constraints after the name of the protocol you're extending by writing a generic `where` clause. For more about generic `where` clauses, see Generic Where Clauses.

For example, you can define an extension to the `Collection` protocol that applies to any collection whose elements conform to the `Equatable` protocol. By constraining a collection's elements to the `Equatable` protocol, a part of the standard library, you can use the `==` and `!=` operators to check for equality and inequality between two elements.

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

The `allEqual()` method returns `true` only if all the elements in the collection are equal.

Consider two arrays of integers, one where all the elements are the same, and one where they aren't:

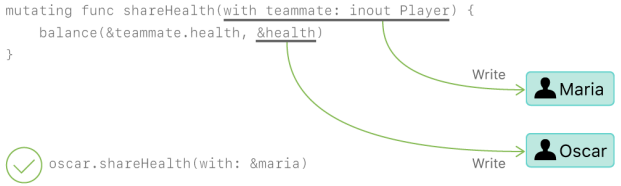
```
let equalNumbers = [100, 100, 100, 100, 100]
let differentNumbers = [100, 100, 200, 100, 200]
```



```
extension Player {
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria)  // OK
```

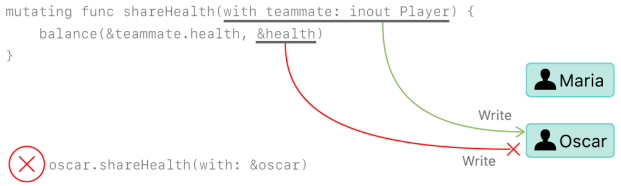
In the example above, calling the `shareHealth(with:)` method for Oscar's player to share health with Maria's player doesn't cause a conflict. There's a write access to `oscar` during the method call because `oscar` is the value of `self` in a mutating method, and there's a write access to `maria` for the same duration because `maria` was passed as an in-out parameter. As shown in the figure below, they access different locations in memory. Even though the two write accesses overlap in time, they don't conflict.



However, if you pass `oscar` as the argument to `shareHealth(with:)`, there's a conflict:

```
oscar.shareHealth(with: &oscar)
// Error: conflicting accesses to oscar
```

The mutating method needs write access to `self` for the duration of the method, and the in-out parameter needs write access to `teammate` for the same duration. Within the method, both `self` and `teammate` refer to the same location in memory—as shown in the figure below. The two write accesses refer to the same memory and they overlap, producing a conflict.



Conflicting Access to Properties

Types like structures, tuples, and enumerations are made up of individual constituent values, such as the properties of a structure or the elements of a tuple. Because these are value types, mutating any piece of the value mutates the whole value, meaning read or write access to one of the properties requires read or write access to the whole value. For example, overlapping write accesses to the elements of a tuple produces a conflict:

Another consequence of long-term write access to in-out parameters is that passing a single variable as the argument for multiple in-out parameters of the same function produces a conflict. For example:

```
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
}
var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore)  // OK
balance(&playerOneScore, &playerOneScore)
// Error: conflicting accesses to playerOneScore
```

The `balance(_:_:)` function above modifies its two parameters to divide the total value evenly between them. Calling it with `playerOneScore` and `playerTwoScore` as arguments doesn't produce a conflict—there are two write accesses that overlap in time, but they access different locations in memory. In contrast, passing `playerOneScore` as the value for both parameters produces a conflict because it tries to perform two write accesses to the same location in memory at the same time.

> NOTE
>
> Because operators are functions, they can also have long-term accesses to their in-out parameters. For example, if `balance(_:_:)` was an operator function named `<^>`, writing `playerOneScore <^> playerOneScore` would result in the same conflict as `balance(&playerOneScore, &playerOneScore)`.

Conflicting Access to self in Methods

A mutating method on a structure has write access to `self` for the duration of the method call. For example, consider a game where each player has a health amount, which decreases when taking damage, and an energy amount, which decreases when using special abilities.

```
struct Player {
    var name: String
    var health: Int
    var energy: Int

    static let maxHealth = 10
    mutating func restoreHealth() {
        health = Player.maxHealth
    }
}
```

In the `restoreHealth()` method above, a write access to `self` starts at the beginning of the method and lasts until the method returns. In this case, there's no other code inside `restoreHealth()` that could have an overlapping access to the properties of a `Player` instance. The `shareHealth(with:)` method below takes another `Player` instance as an in-out parameter, creating the possibility of overlapping accesses.

Because arrays conform to `Collection` and integers conform to `Equatable`, `equalNumbers` and `differentNumbers` can use the `allEqual()` method:

```
print(equalNumbers.allEqual())
// Prints "true"
print(differentNumbers.allEqual())
// Prints "false"
```

> NOTE
>
> If a conforming type satisfies the requirements for multiple constrained extensions that provide implementations for the same method or property, Swift uses the implementation corresponding to the most specialized constraints.

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you've been using generics throughout the *Language Guide*, even if you didn't realize it. For example, Swift's `Array` and `Dictionary` types are both generic collections. You can create an array that holds `Int` values, or an array that holds `String` values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

The Problem That Generics Solve

Here's a standard, nongeneric function called `swapTwoInts(_:_:)`, which swaps two `Int` values:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

This function makes use of in-out parameters to swap the values of `a` and `b`, as described in In-Out Parameters.

The `swapTwoInts(_:_:)` function swaps the original value of `b` into `a`, and the original value of `a` into `b`. You can call this function to swap the values in two `Int` variables:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The `swapTwoInts(_:_:)` function is useful, but it can only be used with `Int` values. If you want to swap two `String` values, or two `Double` values, you have to write more functions, such as the `swapTwoStrings(_:_:)` and `swapTwoDoubles(_:_:)` functions shown below:

Overlapping accesses appear primarily in code that uses in-out parameters in functions and methods or mutating methods of a structure. The specific kinds of Swift code that use long-term accesses are discussed in the sections below.

Conflicting Access to In-Out Parameters

A function has long-term write access to all of its in-out parameters. The write access for an in-out parameter starts after all of the non-in-out parameters have been evaluated and lasts for the entire duration of that function call. If there are multiple in-out parameters, the write accesses start in the same order as the parameters appear.

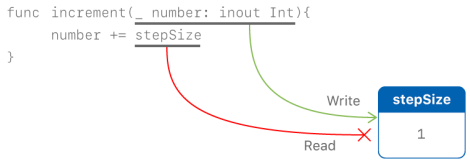
One consequence of this long-term write access is that you can't access the original variable that was passed as in-out, even if scoping rules and access control would otherwise permit it—any access to the original creates a conflict. For example:

```
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

increment(&stepSize)
// Error: conflicting accesses to stepSize
```

In the code above, `stepSize` is a global variable, and it is normally accessible from within `increment(_:)`. However, the read access to `stepSize` overlaps with the write access to `number`. As shown in the figure below, both `number` and `stepSize` refer to the same location in memory. The read and write accesses refer to the same memory and they overlap, producing a conflict.



One way to solve this conflict is to make an explicit copy of `stepSize`:

```
// Make an explicit copy.
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
// stepSize is now 2
```

When you make a copy of `stepSize` before calling `increment(_:)`, it's clear that the value of `copyOfStepSize` is incremented by the current step size. The read access ends before the write access starts, so there isn't a conflict.

This example also demonstrates a challenge you may encounter when fixing conflicting access to memory: There are sometimes multiple ways to fix the conflict that produce different answers, and it's not always obvious which answer is correct. In this example, depending on whether you wanted the original total amount or the updated total amount, either $5 or $320 could be the correct answer. Before you can fix the conflicting access, you have to determine what it was intended to do.

> NOTE
>
> If you've written concurrent or multithreaded code, conflicting access to memory might be a familiar problem. However, the conflicting access discussed here can happen on a single thread and *doesn't* involve concurrent or multithreaded code.
>
> If you have conflicting access to memory from within a single thread, Swift guarantees that you'll get an error at either compile time or runtime. For multithreaded code, use Thread Sanitizer to help detect conflicting access across threads.

Characteristics of Memory Access

There are three characteristics of memory access to consider in the context of conflicting access: whether the access is a read or a write, the duration of the access, and the location in memory being accessed. Specifically, a conflict occurs if you have two accesses that meet all of the following conditions:

- At least one is a write access.
- They access the same location in memory.
- Their durations overlap.

The difference between a read and write access is usually obvious: a write access changes the location in memory, but a read access doesn't. The location in memory refers to what is being accessed—for example, a variable, constant, or property. The duration of a memory access is either instantaneous or long-term.

An access is *instantaneous* if it's not possible for other code to run after that access starts but before it ends. By their nature, two instantaneous accesses can't happen at the same time. Most memory access is instantaneous. For example, all the read and write accesses in the code listing below are instantaneous:

```
func oneMore(than number: Int) -> Int {
    return number + 1
}

var myNumber = 1
myNumber = oneMore(than: myNumber)
print(myNumber)
// Prints "2"
```

However, there are several ways to access memory, called *long-term* accesses, that span the execution of other code. The difference between instantaneous access and long-term access is that it's possible for other code to run after a long-term access starts but before it ends, which is called *overlap*. A long-term access can overlap with other long-term accesses and instantaneous accesses.


```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

You may have noticed that the bodies of the `swapTwoInts(_:_:)`, `swapTwoStrings(_:_:)`, and `swapTwoDoubles(_:_:)` functions are identical. The only difference is the type of the values that they accept (`Int`, `String`, and `Double`).

It's more useful, and considerably more flexible, to write a single function that swaps two values of *any* type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

> NOTE
>
> In all three functions, the types of `a` and `b` must be the same. If `a` and `b` aren't of the same type, it isn't possible to swap their values. Swift is a type-safe language, and doesn't allow (for example) a variable of type `String` and a variable of type `Double` to swap values with each other. Attempting to do so results in a compile-time error.

Generic Functions

Generic functions can work with any type. Here's a generic version of the `swapTwoInts(_:_:)` function from above, called `swapTwoValues(_:_:)`:

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

The body of the `swapTwoValues(_:_:)` function is identical to the body of the `swapTwoInts(_:_:)` function. However, the first line of `swapTwoValues(_:_:)` is slightly different from `swapTwoInts(_:_:)`. Here's how the first lines compare:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

The generic version of the function uses a *placeholder* type name (called `T`, in this case) instead of an *actual* type name (such as `Int`, `String`, or `Double`). The placeholder type name doesn't say anything about what `T` must be, but it *does* say that both `a` and `b` must be of the same type `T`, whatever `T` represents. The actual type to use in place of `T` is determined each time the `swapTwoValues(_:_:)` function is called.

The other difference between a generic function and a nongeneric function is that the generic function's name (`swapTwoValues(_:_:)`) is followed by the placeholder type name (`T`) inside angle brackets (`<T>`). The brackets tell Swift that `T` is a placeholder type name within the `swapTwoValues(_:_:)` function definition. Because `T` is a placeholder, Swift doesn't look for an actual type called `T`.

The `swapTwoValues(_:_:)` function can now be called in the same way as `swapTwoInts`, except that it can be passed two values of *any* type, as long as both of those values are of the same type as each other. Each time `swapTwoValues(_:_:)` is called, the type to use for `T` is inferred from the types of values passed to the function.

In the two examples below, `T` is inferred to be `Int` and `String` respectively:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

> NOTE
>
> The `swapTwoValues(_:_:)` function defined above is inspired by a generic function called `swap`, which is part of the Swift standard library, and is automatically made available for you to use in your apps. If you need the behavior of the `swapTwoValues(_:_:)` function in your own code, you can use Swift's existing `swap(_:_:)` function rather than providing your own implementation.

Type Parameters

In the `swapTwoValues(_:_:)` example above, the placeholder type `T` is an example of a *type parameter*. Type parameters specify and name a placeholder type, and are written immediately after the function's name, between a pair of matching angle brackets (such as `<T>`).

Once you specify a type parameter, you can use it to define the type of a function's parameters (such as the `a` and `b` parameters of the `swapTwoValues(_:_:)` function), or as the function's return type, or as a type annotation within the body of the function. In each case, the type parameter is replaced with an *actual* type whenever the function is called. (In the `swapTwoValues(_:_:)` example above, `T` was replaced with `Int` the first time the function was called, and was replaced with `String` the second time it was called.)

You can provide more than one type parameter by writing multiple type parameter names within the angle brackets, separated by commas.

Naming Type Parameters

Memory Safety

By default, Swift prevents unsafe behavior from happening in your code. For example, Swift ensures that variables are initialized before they're used, memory isn't accessed after it's been deallocated, and array indices are checked for out-of-bounds errors.

Swift also makes sure that multiple accesses to the same area of memory don't conflict, by requiring code that modifies a location in memory to have exclusive access to that memory. Because Swift manages memory automatically, most of the time you don't have to think about accessing memory at all. However, it's important to understand where potential conflicts can occur, so you can avoid writing code that has conflicting access to memory. If your code does contain conflicts, you'll get a compile-time or runtime error.

Understanding Conflicting Access to Memory

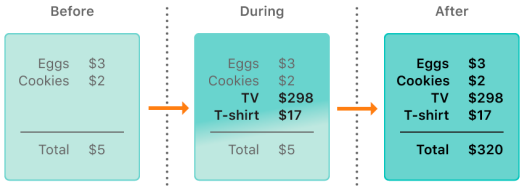
Access to memory happens in your code when you do things like set the value of a variable or pass an argument to a function. For example, the following code contains both a read access and a write access:

```
// A write access to the memory where one is stored.
var one = 1

// A read access from the memory where one is stored.
print("We're number \(one)!")
```

A conflicting access to memory can occur when different parts of your code are trying to access the same location in memory at the same time. Multiple accesses to a location in memory at the same time can produce unpredictable or inconsistent behavior. In Swift, there are ways to modify a value that span several lines of code, making it possible to attempt to access a value in the middle of its own modification.

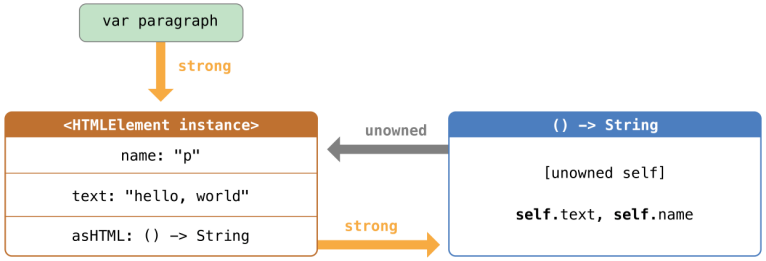
You can see a similar problem by thinking about how you update a budget that's written on a piece of paper. Updating the budget is a two-step process: First you add the items' names and prices, and then you change the total amount to reflect the items currently on the list. Before and after the update, you can read any information from the budget and get a correct answer, as shown in the figure below.



While you're adding items to the budget, it's in a temporary, invalid state because the total amount hasn't been updated to reflect the newly added items. Reading the total amount during the process of adding an item gives you incorrect information.

```swift
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

Here's how the references look with the capture list in place:



This time, the capture of `self` by the closure is an unowned reference, and does not keep a strong hold on the `HTMLElement` instance it has captured. If you set the strong reference from the `paragraph` variable to `nil`, the `HTMLElement` instance is deallocated, as can be seen from the printing of its deinitializer message in the example below:

```swift
paragraph = nil
// Prints "p is being deinitialized"
```

For more information about capture lists, see Capture Lists.

In most cases, type parameters have descriptive names, such as `Key` and `Value` in `Dictionary<Key, Value>` and `Element` in `Array<Element>`, which tells the reader about the relationship between the type parameter and the generic type or function it's used in. However, when there isn't a meaningful relationship between them, it's traditional to name them using single letters such as `T`, `U`, and `V`, such as `T` in the `swapTwoValues(_:_:)` function above.

> NOTE
>
> Always give type parameters upper camel case names (such as `T` and `MyTypeParameter`) to indicate that they're a placeholder for a *type*, not a value.

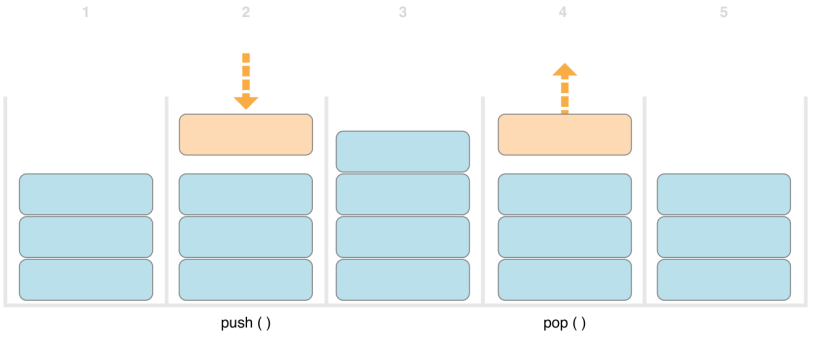
Generic Types

In addition to generic functions, Swift enables you to define your own *generic types*. These are custom classes, structures, and enumerations that can work with *any* type, in a similar way to `Array` and `Dictionary`.

This section shows you how to write a generic collection type called `Stack`. A stack is an ordered set of values, similar to an array, but with a more restricted set of operations than Swift's `Array` type. An array allows new items to be inserted and removed at any location in the array. A stack, however, allows new items to be appended only to the end of the collection (known as *pushing* a new value on to the stack). Similarly, a stack allows items to be removed only from the end of the collection (known as *popping* a value off the stack).

> NOTE
>
> The concept of a stack is used by the `UINavigationController` class to model the view controllers in its navigation hierarchy. You call the `UINavigationController` class `pushViewController(_:animated:)` method to add (or push) a view controller on to the navigation stack, and its `popViewControllerAnimated(_:)` method to remove (or pop) a view controller from the navigation stack. A stack is a useful collection model whenever you need a strict "last in, first out" approach to managing a collection.

The illustration below shows the push and pop behavior for a stack:



1. There are currently three values on the stack.
2. A fourth value is pushed onto the top of the stack.
3. The stack now holds four values, with the most recent one at the top.
4. The top item in the stack is popped.

5. After popping a value, the stack once again holds three values.

Here's how to write a nongeneric version of a stack, in this case for a stack of `Int` values:

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

This structure uses an `Array` property called `items` to store the values in the stack. `Stack` provides two methods, `push` and `pop`, to push and pop values on and off the stack. These methods are marked as `mutating`, because they need to modify (or *mutate*) the structure's `items` array.

The `IntStack` type shown above can only be used with `Int` values, however. It would be much more useful to define a *generic* `Stack` class, that can manage a stack of *any* type of value.

Here's a generic version of the same code:

```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

Note how the generic version of `Stack` is essentially the same as the nongeneric version, but with a type parameter called `Element` instead of an actual type of `Int`. This type parameter is written within a pair of angle brackets (`<Element>`) immediately after the structure's name.

`Element` defines a placeholder name for a type to be provided later. This future type can be referred to as `Element` anywhere within the structure's definition. In this case, `Element` is used as a placeholder in three places:

- To create a property called `items`, which is initialized with an empty array of values of type `Element`
- To specify that the `push(_:)` method has a single parameter called `item`, which must be of type `Element`
- To specify that the value returned by the `pop()` method will be a value of type `Element`

Because it's a generic type, `Stack` can be used to create a stack of *any* valid type in Swift, in a similar manner to `Array` and `Dictionary`.

You create a new `Stack` instance by writing the type to be stored in the stack within angle brackets. For example, to create a new stack of strings, you write `Stack<String>()`:

Weak and Unowned References

Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.

Conversely, define a capture as a weak reference when the captured reference may become `nil` at some point in the future. Weak references are always of an optional type, and automatically become `nil` when the instance they reference is deallocated. This enables you to check for their existence within the closure's body.

> NOTE
>
> If the captured reference will never become `nil`, it should always be captured as an unowned reference, rather than a weak reference.

An unowned reference is the appropriate capture method to use to resolve the strong reference cycle in the `HTMLElement` example from Strong Reference Cycles for Closures above. Here's how you write the `HTMLElement` class to avoid the cycle:

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

This implementation of `HTMLElement` is identical to the previous implementation, apart from the addition of a capture list within the `asHTML` closure. In this case, the capture list is `[unowned self]`, which means "capture self as an unowned reference rather than a strong reference".

You can create and print an `HTMLElement` instance as before:

> Even though the closure refers to `self` multiple times, it only captures one strong reference to the `HTMLElement` instance.

If you set the `paragraph` variable to `nil` and break its strong reference to the `HTMLElement` instance, neither the `HTMLElement` instance nor its closure are deallocated, because of the strong reference cycle:

```
paragraph = nil
```

Note that the message in the `HTMLElement` deinitializer is not printed, which shows that the `HTMLElement` instance is not deallocated.

Resolving Strong Reference Cycles for Closures

You resolve a strong reference cycle between a closure and a class instance by defining a *capture list* as part of the closure's definition. A capture list defines the rules to use when capturing one or more reference types within the closure's body. As with strong reference cycles between two class instances, you declare each captured reference to be a weak or unowned reference rather than a strong reference. The appropriate choice of weak or unowned depends on the relationships between the different parts of your code.

> NOTE
>
> Swift requires you to write `self.someProperty` or `self.someMethod()` (rather than just `someProperty` or `someMethod()`) whenever you refer to a member of `self` within a closure. This helps you remember that it's possible to capture `self` by accident.

Defining a Capture List

Each item in a capture list is a pairing of the `weak` or `unowned` keyword with a reference to a class instance (such as `self`) or a variable initialized with some value (such as `delegate = self.delegate!`). These pairings are written within a pair of square braces, separated by commas.

Place the capture list before a closure's parameter list and return type if they are provided:

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!] (index: Int,
  stringToProcess: String) -> String in
    // closure body goes here
}
```

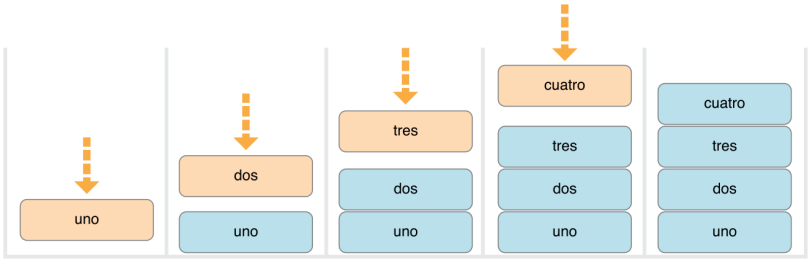
If a closure does not specify a parameter list or return type because they can be inferred from context, place the capture list at the very start of the closure, followed by the `in` keyword:

```
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```



```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

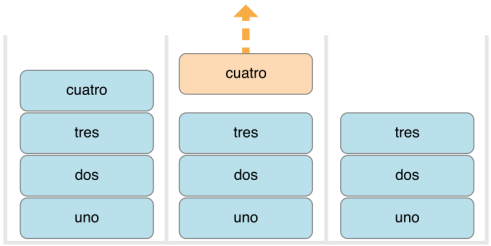
Here's how `stackOfStrings` looks after pushing these four values on to the stack:



Popping a value from the stack removes and returns the top value, `"cuatro"`:

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now contains 3 strings
```

Here's how the stack looks after popping its top value:



Extending a Generic Type

When you extend a generic type, you don't provide a type parameter list as part of the extension's definition. Instead, the type parameter list from the *original* type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

The following example extends the generic `Stack` type to add a read-only computed property called `topItem`, which returns the top item on the stack without popping it from the stack:

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

The `topItem` property returns an optional value of type `Element`. If the stack is empty, `topItem` returns `nil`; if the stack isn't empty, `topItem` returns the final item in the `items` array.

Note that this extension doesn't define a type parameter list. Instead, the `Stack` type's existing type parameter name, `Element`, is used within the extension to indicate the optional type of the `topItem` computed property.

The `topItem` computed property can now be used with any `Stack` instance to access and query its top item without removing it.

```
if let topItem = stackOfStrings.topItem {
    print("The top item on the stack is \(topItem).")
}
// Prints "The top item on the stack is tres."
```

Extensions of a generic type can also include requirements that instances of the extended type must satisfy in order to gain the new functionality, as discussed in Extensions with a Generic Where Clause below.

Type Constraints

The `swapTwoValues(_:_:)` function and the `Stack` type can work with any type. However, it's sometimes useful to enforce certain *type constraints* on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

For example, Swift's `Dictionary` type places a limitation on the types that can be used as keys for a dictionary. As described in Dictionaries, the type of a dictionary's keys must be *hashable*. That is, it must provide a way to make itself uniquely representable. `Dictionary` needs its keys to be hashable so that it can check whether it already contains a value for a particular key. Without this requirement, `Dictionary` could not tell whether it should insert or replace a value for a particular key, nor would it be able to find a value for a given key that is already in the dictionary.

This requirement is enforced by a type constraint on the key type for `Dictionary`, which specifies that the key type must conform to the `Hashable` protocol, a special protocol defined in the Swift standard library. All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default.

You can define your own type constraints when creating custom generic types, and these constraints provide much of the power of generic programming. Abstract concepts like `Hashable` characterize types in terms of their conceptual characteristics, rather than their concrete type.

Type Constraint Syntax

You write type constraints by placing a single class or protocol constraint after a type parameter's name, separated by a colon, as part of the type parameter list. The basic syntax for type constraints on a generic function is shown below (although the syntax is the same for generic types):


```
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\
  (heading.name)>"
}
print(heading.asHTML())
// Prints "<h1>some default text</h1>"
```

> NOTE
>
> The `asHTML` property is declared as a lazy property, because it's only needed if and when the element actually needs to be rendered as a string value for some HTML output target. The fact that `asHTML` is a lazy property means that you can refer to `self` within the default closure, because the lazy property will not be accessed until after initialization has been completed and `self` is known to exist.

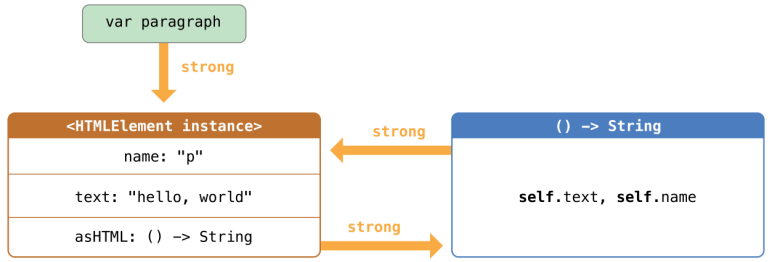
The `HTMLElement` class provides a single initializer, which takes a `name` argument and (if desired) a `text` argument to initialize a new element. The class also defines a deinitializer, which prints a message to show when an `HTMLElement` instance is deallocated.

Here's how you use the `HTMLElement` class to create and print a new instance:

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

> NOTE
>
> The `paragraph` variable above is defined as an *optional* `HTMLElement`, so that it can be set to `nil` below to demonstrate the presence of a strong reference cycle.

Unfortunately, the `HTMLElement` class, as written above, creates a strong reference cycle between an `HTMLElement` instance and the closure used for its default `asHTML` value. Here's how the cycle looks:



The instance's `asHTML` property holds a strong reference to its closure. However, because the closure refers to `self` within its body (as a way to reference `self.name` and `self.text`), the closure *captures* self, which means that it holds a strong reference back to the `HTMLElement` instance. A strong reference cycle is created between the two. (For more information about capturing values in a closure, see Capturing Values.)

> NOTE

The example below shows how you can create a strong reference cycle when using a closure that references `self`. This example defines a class called `HTMLElement`, which provides a simple model for an individual element within an HTML document:

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

The `HTMLElement` class defines a `name` property, which indicates the name of the element, such as `"h1"` for a heading element, `"p"` for a paragraph element, or `"br"` for a line break element. `HTMLElement` also defines an optional `text` property, which you can set to a string that represents the text to be rendered within that HTML element.

In addition to these two simple properties, the `HTMLElement` class defines a lazy property called `asHTML`. This property references a closure that combines `name` and `text` into an HTML string fragment. The `asHTML` property is of type `() -> String`, or "a function that takes no parameters, and returns a `String` value".

By default, the `asHTML` property is assigned a closure that returns a string representation of an HTML tag. This tag contains the optional `text` value if it exists, or no text content if `text` does not exist. For a paragraph element, the closure would return `"<p>some text</p>"` or `"<p />"`, depending on whether the `text` property equals `"some text"` or `nil`.

The `asHTML` property is named and used somewhat like an instance method. However, because `asHTML` is a closure property rather than an instance method, you can replace the default value of the `asHTML` property with a custom closure, if you want to change the HTML rendering for a particular HTML element.

For example, the `asHTML` property could be set to a closure that defaults to some text if the `text` property is `nil`, in order to prevent the representation from returning an empty HTML tag:


```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

The hypothetical function above has two type parameters. The first type parameter, `T`, has a type constraint that requires `T` to be a subclass of `SomeClass`. The second type parameter, `U`, has a type constraint that requires `U` to conform to the protocol `SomeProtocol`.

Type Constraints in Action

Here's a nongeneric function called `findIndex(ofString:in:)`, which is given a `String` value to find and an array of `String` values within which to find it. The `findIndex(ofString:in:)` function returns an optional `Int` value, which will be the index of the first matching string in the array if it's found, or `nil` if the string can't be found:

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The `findIndex(ofString:in:)` function can be used to find a string value in an array of strings:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}
// Prints "The index of llama is 2"
```

The principle of finding the index of a value in an array isn't useful only for strings, however. You can write the same functionality as a generic function by replacing any mention of strings with values of some type `T` instead.

Here's how you might expect a generic version of `findIndex(ofString:in:)`, called `findIndex(of:in:)`, to be written. Note that the return type of this function is still `Int?`, because the function returns an optional index number, not an optional value from the array. Be warned, though—this function doesn't compile, for reasons explained after the example:

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

This function doesn't compile as written above. The problem lies with the equality check, "`if value == valueToFind`". Not every type in Swift can be compared with the equal to operator (==). If you create your own class or structure to represent a complex data model, for example, then the meaning of "equal to" for that class or structure isn't something that Swift can guess for you. Because of this, it isn't possible to guarantee that this code will work for *every* possible type `T`, and an appropriate error is reported when you try to compile the code.

All is not lost, however. The Swift standard library defines a protocol called `Equatable`, which requires any conforming type to implement the equal to operator (==) and the not equal to operator (`!=`) to compare any two values of that type. All of Swift's standard types automatically support the `Equatable` protocol.

Any type that is `Equatable` can be used safely with the `findIndex(of:in:)` function, because it's guaranteed to support the equal to operator. To express this fact, you write a type constraint of `Equatable` as part of the type parameter's definition when you define the function:

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The single type parameter for `findIndex(of:in:)` is written as `T: Equatable`, which means "any type `T` that conforms to the `Equatable` protocol."

The `findIndex(of:in:)` function now compiles successfully and can be used with any type that is `Equatable`, such as `Double` or `String`:

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
// doubleIndex is an optional Int with no value, because 9.3 isn't in the
  array
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm",
  "Andrea"])
// stringIndex is an optional Int containing a value of 2
```

Associated Types

When defining a protocol, it's sometimes useful to declare one or more associated types as part of the protocol's definition. An *associated type* gives a placeholder name to a type that is used as part of the protocol. The actual type to use for that associated type isn't specified until the protocol is adopted. Associated types are specified with the `associatedtype` keyword.

Associated Types in Action

Here's an example of a protocol called `Container`, which declares an associated type called `Item`:

To cope with this requirement, you declare the `capitalCity` property of `Country` as an implicitly unwrapped optional property, indicated by the exclamation mark at the end of its type annotation (`City!`). This means that the `capitalCity` property has a default value of `nil`, like any other optional, but can be accessed without the need to unwrap its value as described in Implicitly Unwrapped Optionals.

Because `capitalCity` has a default `nil` value, a new `Country` instance is considered fully initialized as soon as the `Country` instance sets its `name` property within its initializer. This means that the `Country` initializer can start to reference and pass around the implicit `self` property as soon as the `name` property is set. The `Country` initializer can therefore pass `self` as one of the parameters for the `City` initializer when the `Country` initializer is setting its own `capitalCity` property.

All of this means that you can create the `Country` and `City` instances in a single statement, without creating a strong reference cycle, and the `capitalCity` property can be accessed directly, without needing to use an exclamation mark to unwrap its optional value:

```
var country = Country(name: "Canada", capitalName: "Ottawa")
print("\(country.name)'s capital city is called \
  (country.capitalCity.name)")
// Prints "Canada's capital city is called Ottawa"
```

In the example above, the use of an implicitly unwrapped optional means that all of the two-phase class initializer requirements are satisfied. The `capitalCity` property can be used and accessed like a nonoptional value once initialization is complete, while still avoiding a strong reference cycle.

Strong Reference Cycles for Closures

You saw above how a strong reference cycle can be created when two class instance properties hold a strong reference to each other. You also saw how to use weak and unowned references to break these strong reference cycles.

A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance. This capture might occur because the closure's body accesses a property of the instance, such as `self.someProperty`, or because the closure calls a method on the instance, such as `self.someMethod()`. In either case, these accesses cause the closure to "capture" `self`, creating a strong reference cycle.

This strong reference cycle occurs because closures, like classes, are *reference types*. When you assign a closure to a property, you are assigning a *reference* to that closure. In essence, it's the same problem as above—two strong references are keeping each other alive. However, rather than two class instances, this time it's a class instance and a closure that are keeping each other alive.

Swift provides an elegant solution to this problem, known as a *closure capture list*. However, before you learn how to break a strong reference cycle with a closure capture list, it's useful to understand how such a cycle can be caused.

Unowned References and Implicitly Unwrapped Optional Properties

The examples for weak and unowned references above cover two of the more common scenarios in which it's necessary to break a strong reference cycle.

The `Person` and `Apartment` example shows a situation where two properties, both of which are allowed to be `nil`, have the potential to cause a strong reference cycle. This scenario is best resolved with a weak reference.

The `Customer` and `CreditCard` example shows a situation where one property that is allowed to be `nil` and another property that cannot be `nil` have the potential to cause a strong reference cycle. This scenario is best resolved with an unowned reference.

However, there is a third scenario, in which *both* properties should always have a value, and neither property should ever be `nil` once initialization is complete. In this scenario, it's useful to combine an unowned property on one class with an implicitly unwrapped optional property on the other class.

This enables both properties to be accessed directly (without optional unwrapping) once initialization is complete, while still avoiding a reference cycle. This section shows you how to set up such a relationship.

The example below defines two classes, `Country` and `City`, each of which stores an instance of the other class as a property. In this data model, every country must always have a capital city, and every city must always belong to a country. To represent this, the `Country` class has a `capitalCity` property, and the `City` class has a `country` property:

```
class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

To set up the interdependency between the two classes, the initializer for `City` takes a `Country` instance, and stores this instance in its `country` property.

The initializer for `City` is called from within the initializer for `Country`. However, the initializer for `Country` cannot pass `self` to the `City` initializer until a new `Country` instance is fully initialized, as described in Two-Phase Initialization.


```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

The `Container` protocol defines three required capabilities that any container must provide:

- It must be possible to add a new item to the container with an `append(_:)` method.
- It must be possible to access a count of the items in the container through a `count` property that returns an `Int` value.
- It must be possible to retrieve each item in the container with a subscript that takes an `Int` index value.

This protocol doesn't specify how the items in the container should be stored or what type they're allowed to be. The protocol only specifies the three bits of functionality that any type must provide in order to be considered a `Container`. A conforming type can provide additional functionality, as long as it satisfies these three requirements.

Any type that conforms to the `Container` protocol must be able to specify the type of values it stores. Specifically, it must ensure that only items of the right type are added to the container, and it must be clear about the type of the items returned by its subscript.

To define these requirements, the `Container` protocol needs a way to refer to the type of the elements that a container will hold, without knowing what that type is for a specific container. The `Container` protocol needs to specify that any value passed to the `append(_:)` method must have the same type as the container's element type, and that the value returned by the container's subscript will be of the same type as the container's element type.

To achieve this, the `Container` protocol declares an associated type called `Item`, written as `associatedtype Item`. The protocol doesn't define what `Item` is—that information is left for any conforming type to provide. Nonetheless, the `Item` alias provides a way to refer to the type of the items in a `Container`, and to define a type for use with the `append(_:)` method and subscript, to ensure that the expected behavior of any `Container` is enforced.

Here's a version of the nongeneric `IntStack` type from Generic Types above, adapted to conform to the `Container` protocol:

```swift
struct IntStack: Container {
    // original IntStack implementation
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias Item = Int
    mutating func append(_ item: Int) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

The `IntStack` type implements all three of the `Container` protocol's requirements, and in each case wraps part of the `IntStack` type's existing functionality to satisfy these requirements.

Moreover, `IntStack` specifies that for this implementation of `Container`, the appropriate `Item` to use is a type of `Int`. The definition of `typealias Item = Int` turns the abstract type of `Item` into a concrete type of `Int` for this implementation of the `Container` protocol.

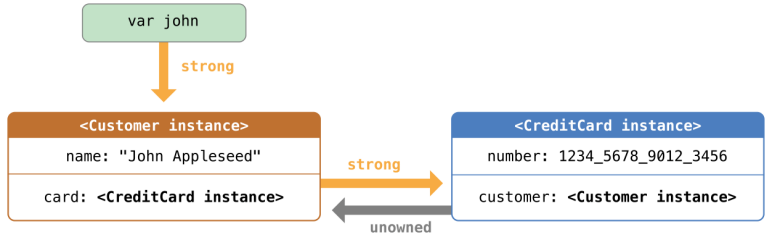
Thanks to Swift's type inference, you don't actually need to declare a concrete `Item` of `Int` as part of the definition of `IntStack`. Because `IntStack` conforms to all of the requirements of the `Container` protocol, Swift can infer the appropriate `Item` to use, simply by looking at the type of the `append(_:)` method's `item` parameter and the return type of the subscript. Indeed, if you delete the `typealias Item = Int` line from the code above, everything still works, because it's clear what type should be used for `Item`.

You can also make the generic `Stack` type conform to the `Container` protocol:

You can now create a `Customer` instance, and use it to initialize and assign a new `CreditCard` instance as that customer's `card` property:

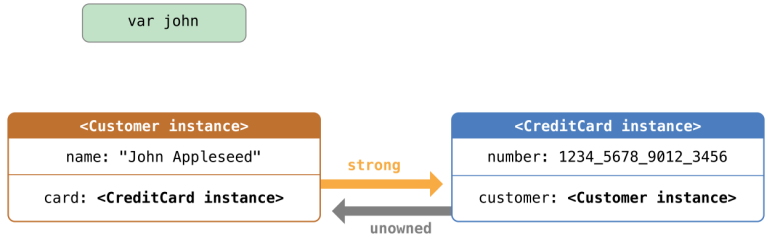
```swift
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

Here's how the references look, now that you've linked the two instances:



The `Customer` instance now has a strong reference to the `CreditCard` instance, and the `CreditCard` instance has an unowned reference to the `Customer` instance.

Because of the unowned `customer` reference, when you break the strong reference held by the `john` variable, there are no more strong references to the `Customer` instance:



Because there are no more strong references to the `Customer` instance, it's deallocated. After this happens, there are no more strong references to the `CreditCard` instance, and it too is deallocated:

```swift
john = nil
// Prints "John Appleseed is being deinitialized"
// Prints "Card #1234567890123456 is being deinitialized"
```

The final code snippet above shows that the deinitializers for the `Customer` instance and `CreditCard` instance both print their "deinitialized" messages after the `john` variable is set to `nil`.

> NOTE
>
> The examples above show how to use *safe* unowned references. Swift also provides *unsafe* unowned references for cases where you need to disable runtime safety checks—for example, for performance reasons. As with all unsafe operations, you take on the responsibility for checking that code for safety.
>
> You indicate an unsafe unowned reference by writing `unowned(unsafe)`. If you try to access an unsafe unowned reference after the instance that it refers to is deallocated, your program will try to access the memory location where the instance used to be, which is an unsafe operation.

> If you try to access the value of an unowned reference after that instance has been deallocated, you'll get a runtime error.

The following example defines two classes, `Customer` and `CreditCard`, which model a bank customer and a possible credit card for that customer. These two classes each store an instance of the other class as a property. This relationship has the potential to create a strong reference cycle.

The relationship between `Customer` and `CreditCard` is slightly different from the relationship between `Apartment` and `Person` seen in the weak reference example above. In this data model, a customer may or may not have a credit card, but a credit card will *always* be associated with a customer. A `CreditCard` instance never outlives the `Customer` that it refers to. To represent this, the `Customer` class has an optional `card` property, but the `CreditCard` class has an unowned (and nonoptional) `customer` property.

Furthermore, a new `CreditCard` instance can *only* be created by passing a `number` value and a `customer` instance to a custom `CreditCard` initializer. This ensures that a `CreditCard` instance always has a `customer` instance associated with it when the `CreditCard` instance is created.

Because a credit card will always have a customer, you define its `customer` property as an unowned reference, to avoid a strong reference cycle:

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
```

> NOTE
>
> The `number` property of the `CreditCard` class is defined with a type of `UInt64` rather than `Int`, to ensure that the `number` property's capacity is large enough to store a 16-digit card number on both 32-bit and 64-bit systems.

This next code snippet defines an optional `Customer` variable called `john`, which will be used to store a reference to a specific customer. This variable has an initial value of nil, by virtue of being optional:

```
var john: Customer?
```



```
struct Stack<Element>: Container {
    // original Stack<Element> implementation
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}
```

This time, the type parameter `Element` is used as the type of the `append(_:)` method's `item` parameter and the return type of the subscript. Swift can therefore infer that `Element` is the appropriate type to use as the `Item` for this particular container.

Extending an Existing Type to Specify an Associated Type

You can extend an existing type to add conformance to a protocol, as described in Adding Protocol Conformance with an Extension. This includes a protocol with an associated type.

Swift's `Array` type already provides an `append(_:)` method, a `count` property, and a subscript with an `Int` index to retrieve its elements. These three capabilities match the requirements of the `Container` protocol. This means that you can extend `Array` to conform to the `Container` protocol simply by declaring that `Array` adopts the protocol. You do this with an empty extension, as described in Declaring Protocol Adoption with an Extension:

```
extension Array: Container {}
```

Array's existing `append(_:)` method and subscript enable Swift to infer the appropriate type to use for `Item`, just as for the generic `Stack` type above. After defining this extension, you can use any `Array` as a `Container`.

Adding Constraints to an Associated Type

You can add type constraints to an associated type in a protocol to require that conforming types satisfy those constraints. For example, the following code defines a version of `Container` that requires the items in the container to be equatable.

```swift
protocol Container {
    associatedtype Item: Equatable
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

To conform to this version of `Container`, the container's `Item` type has to conform to the `Equatable` protocol.

Using a Protocol in Its Associated Type's Constraints

A protocol can appear as part of its own requirements. For example, here's a protocol that refines the `Container` protocol, adding the requirement of a `suffix(_:)` method. The `suffix(_:)` method returns a given number of elements from the end of the container, storing them in an instance of the `Suffix` type.

```swift
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where Suffix.Item == Item
    func suffix(_ size: Int) -> Suffix
}
```

In this protocol, `Suffix` is an associated type, like the `Item` type in the `Container` example above. `Suffix` has two constraints: It must conform to the `SuffixableContainer` protocol (the protocol currently being defined), and its `Item` type must be the same as the container's `Item` type. The constraint on `Item` is a generic `where` clause, which is discussed in Associated Types with a Generic Where Clause below.

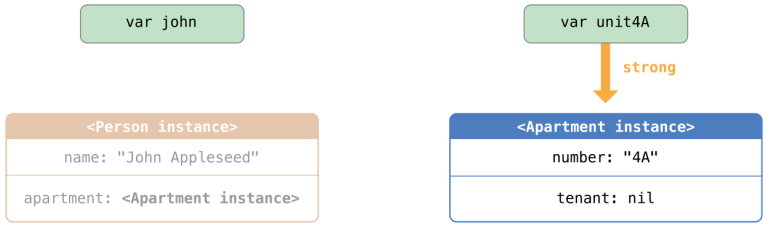
Here's an extension of the `Stack` type from Strong Reference Cycles for Closures above that adds conformance to the `SuffixableContainer` protocol:

```swift
extension Stack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack {
        var result = Stack()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack.
}
var stackOfInts = Stack<Int>()
stackOfInts.append(10)
stackOfInts.append(20)
stackOfInts.append(30)
let suffix = stackOfInts.suffix(2)
// suffix contains 20 and 30
```



```swift
john = nil
// Prints "John Appleseed is being deinitialized"
```

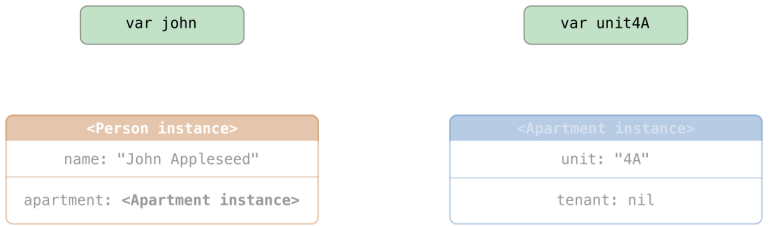
Because there are no more strong references to the `Person` instance, it's deallocated and the `tenant` property is set to `nil`:



The only remaining strong reference to the `Apartment` instance is from the `unit4A` variable. If you break *that* strong reference, there are no more strong references to the `Apartment` instance:

```swift
unit4A = nil
// Prints "Apartment 4A is being deinitialized"
```

Because there are no more strong references to the `Apartment` instance, it too is deallocated:



NOTE

In systems that use garbage collection, weak pointers are sometimes used to implement a simple caching mechanism because objects with no strong references are deallocated only when memory pressure triggers garbage collection. However, with ARC, values are deallocated as soon as their last strong reference is removed, making weak references unsuitable for such a purpose.

Unowned References

Like a weak reference, an *unowned reference* does not keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is used when the other instance has the same lifetime or a longer lifetime. You indicate an unowned reference by placing the `unowned` keyword before a property or variable declaration.

An unowned reference is expected to always have a value. As a result, ARC never sets an unowned reference's value to `nil`, which means that unowned references are defined using nonoptional types.

IMPORTANT

Use an unowned reference only when you are sure that the reference *always* refers to an instance that has not been deallocated.

You can check for the existence of a value in the weak reference, just like any other optional value, and you will never end up with a reference to an invalid instance that no longer exists.

> NOTE
>
> Property observers aren't called when ARC sets a weak reference to `nil`.

The example below is identical to the `Person` and `Apartment` example from above, with one important difference. This time around, the `Apartment` type's `tenant` property is declared as a weak reference:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

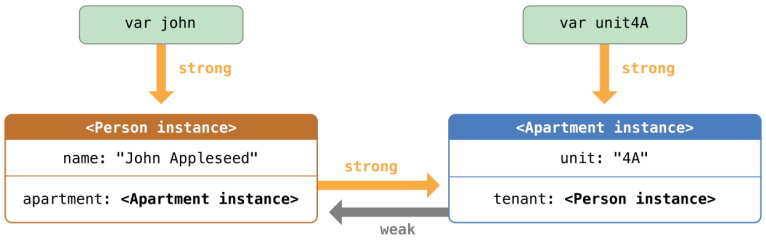
The strong references from the two variables (`john` and `unit4A`) and the links between the two instances are created as before:

```
var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```

Here's how the references look now that you've linked the two instances together:



The `Person` instance still has a strong reference to the `Apartment` instance, but the `Apartment` instance now has a *weak* reference to the `Person` instance. This means that when you break the strong reference held by the `john` variable by setting it to `nil`, there are no more strong references to the `Person` instance:

In the example above, the `Suffix` associated type for `Stack` is also `Stack`, so the suffix operation on `Stack` returns another `Stack`. Alternatively, a type that conforms to `SuffixableContainer` can have a `Suffix` type that's different from itself—meaning the suffix operation can return a different type. For example, here's an extension to the nongeneric `IntStack` type that adds `SuffixableContainer` conformance, using `Stack<Int>` as its suffix type instead of `IntStack`:

```
extension IntStack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack<Int> {
        var result = Stack<Int>()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack<Int>.
}
```

Generic Where Clauses

Type constraints, as described in Type Constraints, enable you to define requirements on the type parameters associated with a generic function, subscript, or type.

It can also be useful to define requirements for associated types. You do this by defining a *generic where clause*. A generic `where` clause enables you to require that an associated type must conform to a certain protocol, or that certain type parameters and associated types must be the same. A generic `where` clause starts with the `where` keyword, followed by constraints for associated types or equality relationships between types and associated types. You write a generic `where` clause right before the opening curly brace of a type or function's body.

The example below defines a generic function called `allItemsMatch`, which checks to see if two `Container` instances contain the same items in the same order. The function returns a Boolean value of `true` if all items match and a value of `false` if they don't.

The two containers to be checked don't have to be the same type of container (although they can be), but they do have to hold the same type of items. This requirement is expressed through a combination of type constraints and a generic `where` clause:

```swift
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {

        // Check that both containers contain the same number of items.
        if someContainer.count != anotherContainer.count {
            return false
        }

        // Check each pair of items to see if they're equivalent.
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // All items match, so return true.
        return true
}
```

This function takes two arguments called `someContainer` and `anotherContainer`. The `someContainer` argument is of type `C1`, and the `anotherContainer` argument is of type `C2`. Both `C1` and `C2` are type parameters for two container types to be determined when the function is called.

The following requirements are placed on the function's two type parameters:

- `C1` must conform to the `Container` protocol (written as `C1: Container`).
- `C2` must also conform to the `Container` protocol (written as `C2: Container`).
- The `Item` for `C1` must be the same as the `Item` for `C2` (written as `C1.Item == C2.Item`).
- The `Item` for `C1` must conform to the `Equatable` protocol (written as `C1.Item: Equatable`).

The first and second requirements are defined in the function's type parameter list, and the third and fourth requirements are defined in the function's generic `where` clause.

These requirements mean:

- `someContainer` is a container of type `C1`.
- `anotherContainer` is a container of type `C2`.
- `someContainer` and `anotherContainer` contain the same type of items.
- The items in `someContainer` can be checked with the not equal operator (`!=`) to see if they're different from each other.

The third and fourth requirements combine to mean that the items in `anotherContainer` can *also* be checked with the `!=` operator, because they're exactly the same type as the items in `someContainer`.

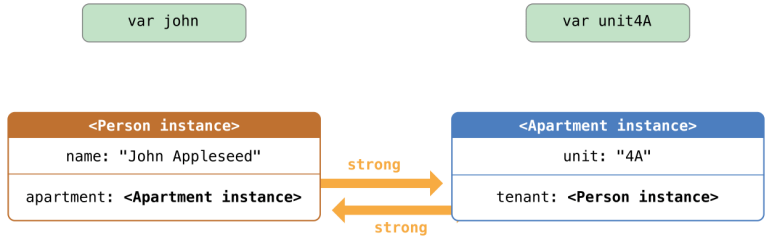
These requirements enable the `allItemsMatch(_:_:)` function to compare the two containers, even if they're of a different container type.

The `allItemsMatch(_:_:)` function starts by checking that both containers contain the same number of items. If they contain a different number of items, there's no way that they can match, and the function returns `false`.


```swift
john = nil
unit4A = nil
```

Note that neither deinitializer was called when you set these two variables to `nil`. The strong reference cycle prevents the `Person` and `Apartment` instances from ever being deallocated, causing a memory leak in your app.

Here's how the strong references look after you set the `john` and `unit4A` variables to `nil`:



The strong references between the `Person` instance and the `Apartment` instance remain and cannot be broken.

Resolving Strong Reference Cycles Between Class Instances

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: weak references and unowned references.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance *without* keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.

Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first. In the `Apartment` example above, it's appropriate for an apartment to be able to have no tenant at some point in its lifetime, and so a weak reference is an appropriate way to break the reference cycle in this case. In contrast, use an unowned reference when the other instance has the same lifetime or a longer lifetime.

Weak References

A *weak reference* is a reference that does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance. This behavior prevents the reference from becoming part of a strong reference cycle. You indicate a weak reference by placing the `weak` keyword before a property or variable declaration.

Because a weak reference does not keep a strong hold on the instance it refers to, it's possible for that instance to be deallocated while the weak reference is still referring to it. Therefore, ARC automatically sets a weak reference to `nil` when the instance that it refers to is deallocated. And, because weak references need to allow their value to be changed to `nil` at runtime, they are always declared as variables, rather than constants, of an optional type.

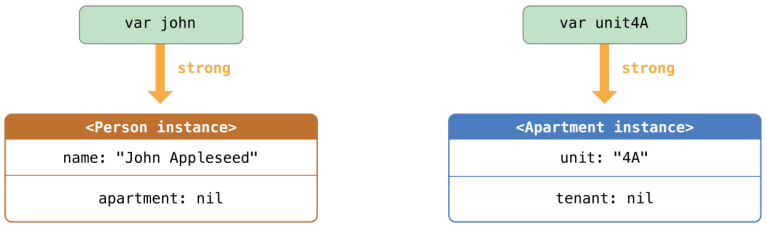
This next code snippet defines two variables of optional type called `john` and `unit4A`, which will be set to a specific `Apartment` and `Person` instance below. Both of these variables have an initial value of `nil`, by virtue of being optional:

```
var john: Person?
var unit4A: Apartment?
```

You can now create a specific `Person` instance and `Apartment` instance and assign these new instances to the `john` and `unit4A` variables:

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

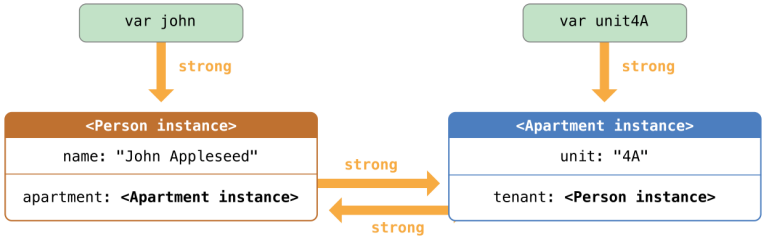
Here's how the strong references look after creating and assigning these two instances. The `john` variable now has a strong reference to the new `Person` instance, and the `unit4A` variable has a strong reference to the new `Apartment` instance:



You can now link the two instances together so that the person has an apartment, and the apartment has a tenant. Note that an exclamation mark (`!`) is used to unwrap and access the instances stored inside the `john` and `unit4A` optional variables, so that the properties of those instances can be set:

```
john!.apartment = unit4A
unit4A!.tenant = john
```

Here's how the strong references look after you link the two instances together:



Unfortunately, linking these two instances creates a strong reference cycle between them. The `Person` instance now has a strong reference to the `Apartment` instance, and the `Apartment` instance has a strong reference to the `Person` instance. Therefore, when you break the strong references held by the `john` and `unit4A` variables, the reference counts do not drop to zero, and the instances are not deallocated by ARC:

After making this check, the function iterates over all of the items in `someContainer` with a `for-in` loop and the half-open range operator (`..<`). For each item, the function checks whether the item from `someContainer` isn't equal to the corresponding item in `anotherContainer`. If the two items aren't equal, then the two containers don't match, and the function returns `false`.

If the loop finishes without finding a mismatch, the two containers match, and the function returns `true`.

Here's how the `allItemsMatch(_:_:)` function looks in action:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    print("All items match.")
} else {
    print("Not all items match.")
}
// Prints "All items match."
```

The example above creates a `Stack` instance to store `String` values, and pushes three strings onto the stack. The example also creates an `Array` instance initialized with an array literal containing the same three strings as the stack. Even though the stack and the array are of a different type, they both conform to the `Container` protocol, and both contain the same type of values. You can therefore call the `allItemsMatch(_:_:)` function with these two containers as its arguments. In the example above, the `allItemsMatch(_:_:)` function correctly reports that all of the items in the two containers match.

Extensions with a Generic Where Clause

You can also use a generic `where` clause as part of an extension. The example below extends the generic `Stack` structure from the previous examples to add an `isTop(_:)` method.

```
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        return topItem == item
    }
}
```

This new `isTop(_:)` method first checks that the stack isn't empty, and then compares the given item against the stack's topmost item. If you tried to do this without a generic `where` clause, you would have a problem: The implementation of `isTop(_:)` uses the == operator, but the definition of `Stack` doesn't require its items to be equatable, so using the == operator results in a compile-time error. Using a generic `where` clause lets you add a new requirement to the extension, so that the extension adds the `isTop(_:)` method only when the items in the stack are equatable.

Here's how the `isTop(_:)` method looks in action:

```
if stackOfStrings.isTop("tres") {
    print("Top element is tres.")
} else {
    print("Top element is something else.")
}
// Prints "Top element is tres."
```

If you try to call the `isTop(_:)` method on a stack whose elements aren't equatable, you'll get a compile-time error.

```
struct NotEquatable { }
var notEquatableStack = Stack<NotEquatable>()
let notEquatableValue = NotEquatable()
notEquatableStack.push(notEquatableValue)
notEquatableStack.isTop(notEquatableValue)  // Error
```

You can use a generic `where` clause with extensions to a protocol. The example below extends the `Container` protocol from the previous examples to add a `startsWith(_:)` method.

```
extension Container where Item: Equatable {
    func startsWith(_ item: Item) -> Bool {
        return count >= 1 && self[0] == item
    }
}
```

The `startsWith(_:)` method first makes sure that the container has at least one item, and then it checks whether the first item in the container matches the given item. This new `startsWith(_:)` method can be used with any type that conforms to the `Container` protocol, including the stacks and arrays used above, as long as the container's items are equatable.

```
if [9, 9, 9].startsWith(42) {
    print("Starts with 42.")
} else {
    print("Starts with something else.")
}
// Prints "Starts with something else."
```

The generic `where` clause in the example above requires `Item` to conform to a protocol, but you can also write a generic `where` clauses that require `Item` to be a specific type. For example:

ARC does not deallocate the `Person` instance until the third and final strong reference is broken, at which point it's clear that you are no longer using the `Person` instance:

```
reference3 = nil
// Prints "John Appleseed is being deinitialized"
```

Strong Reference Cycles Between Class Instances

In the examples above, ARC is able to track the number of references to the new `Person` instance you create and to deallocate that `Person` instance when it's no longer needed.

However, it's possible to write code in which an instance of a class *never* gets to a point where it has zero strong references. This can happen if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a *strong reference cycle*.

You resolve strong reference cycles by defining some of the relationships between classes as weak or unowned references instead of as strong references. This process is described in Resolving Strong Reference Cycles Between Class Instances. However, before you learn how to resolve a strong reference cycle, it's useful to understand how such a cycle is caused.

Here's an example of how a strong reference cycle can be created by accident. This example defines two classes called `Person` and `Apartment`, which model a block of apartments and its residents:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

Every `Person` instance has a `name` property of type `String` and an optional `apartment` property that is initially `nil`. The `apartment` property is optional, because a person may not always have an apartment.

Similarly, every `Apartment` instance has a `unit` property of type `String` and has an optional `tenant` property that is initially `nil`. The tenant property is optional because an apartment may not always have a tenant.

Both of these classes also define a deinitializer, which prints the fact that an instance of that class is being deinitialized. This enables you to see whether instances of `Person` and `Apartment` are being deallocated as expected.

```swift
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

The `Person` class has an initializer that sets the instance's `name` property and prints a message to indicate that initialization is underway. The `Person` class also has a deinitializer that prints a message when an instance of the class is deallocated.

The next code snippet defines three variables of type `Person?`, which are used to set up multiple references to a new `Person` instance in subsequent code snippets. Because these variables are of an optional type (`Person?`, not `Person`), they are automatically initialized with a value of `nil`, and do not currently reference a `Person` instance.

```swift
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

You can now create a new `Person` instance and assign it to one of these three variables:

```swift
reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"
```

Note that the message `"John Appleseed is being initialized"` is printed at the point that you call the `Person` class's initializer. This confirms that initialization has taken place.

Because the new `Person` instance has been assigned to the `reference1` variable, there is now a strong reference from `reference1` to the new `Person` instance. Because there is at least one strong reference, ARC makes sure that this `Person` is kept in memory and is not deallocated.

If you assign the same `Person` instance to two more variables, two more strong references to that instance are established:

```swift
reference2 = reference1
reference3 = reference1
```

There are now *three* strong references to this single `Person` instance.

If you break two of these strong references (including the original reference) by assigning `nil` to two of the variables, a single strong reference remains, and the `Person` instance is not deallocated:

```swift
reference1 = nil
reference2 = nil
```



```swift
extension Container where Item == Double {
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += self[index]
        }
        return sum / Double(count)
    }
}
print([1260.0, 1200.0, 98.6, 37.0].average())
// Prints "648.9"
```

This example adds an `average()` method to containers whose `Item` type is `Double`. It iterates over the items in the container to add them up, and divides by the container's count to compute the average. It explicitly converts the count from `Int` to `Double` to be able to do floating-point division.

You can include multiple requirements in a generic `where` clause that is part of an extension, just like you can for a generic `where` clause that you write elsewhere. Separate each requirement in the list with a comma.

Associated Types with a Generic Where Clause

You can include a generic `where` clause on an associated type. For example, suppose you want to make a version of `Container` that includes an iterator, like what the `Sequence` protocol uses in the standard library. Here's how you write that:

```swift
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }

    associatedtype Iterator: IteratorProtocol where Iterator.Element ==
  Item
    func makeIterator() -> Iterator
}
```

The generic `where` clause on `Iterator` requires that the iterator must traverse over elements of the same item type as the container's items, regardless of the iterator's type. The `makeIterator()` function provides access to a container's iterator.

For a protocol that inherits from another protocol, you add a constraint to an inherited associated type by including the generic `where` clause in the protocol declaration. For example, the following code declares a `ComparableContainer` protocol that requires `Item` to conform to `Comparable`:

```swift
protocol ComparableContainer: Container where Item: Comparable { }
```

Generic Subscripts

Subscripts can be generic, and they can include generic `where` clauses. You write the placeholder type name inside angle brackets after `subscript`, and you write a generic `where` clause right before the opening curly brace of the subscript's body. For example:

```
extension Container {
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result = [Item]()
            for index in indices {
                result.append(self[index])
            }
            return result
    }
}
```

This extension to the `Container` protocol adds a subscript that takes a sequence of indices and returns an array containing the items at each given index. This generic subscript is constrained as follows:

- The generic parameter `Indices` in angle brackets has to be a type that conforms to the `Sequence` protocol from the standard library.
- The subscript takes a single parameter, `indices`, which is an instance of that `Indices` type.
- The generic `where` clause requires that the iterator for the sequence must traverse over elements of type `Int`. This ensures that the indices in the sequence are the same type as the indices used for a container.

Taken together, these constraints mean that the value passed for the `indices` parameter is a sequence of integers.

Automatic Reference Counting

Swift uses *Automatic Reference Counting* (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you do not need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

However, in a few cases ARC requires more information about the relationships between parts of your code in order to manage memory for you. This chapter describes those situations and shows how you enable ARC to manage all of your app's memory. Using ARC in Swift is very similar to the approach described in Transitioning to ARC Release Notes for using ARC with Objective-C.

Reference counting applies only to instances of classes. Structures and enumerations are value types, not reference types, and are not stored and passed by reference.

How ARC Works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances do not take up space in memory when they are no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. The reference is called a "strong" reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains.

ARC in Action

Here's an example of how Automatic Reference Counting works. This example starts with a simple class called `Person`, which defines a stored constant property called `name`: