The Basics | Documentation

Work with common kinds of data and write basic syntax.

Swift is a programming language for iOS, macOS, watchOS, and tvOS app development. If you have experience developing in C or Objective-C, many parts of Swift will be familiar to you.

Swift provides its own versions of all fundamental C and Objective-C types, including `Int` for integers, `Double` and `Float` for floating-point values, `Bool` for Boolean values, and `String` for textual data. Swift also provides powerful versions of the three primary collection types, `Array`, `Set`, and `Dictionary`, as described in Collection Types.

Like C, Swift uses variables to store and refer to values by an identifying name. Swift also makes extensive use of variables whose values can't be changed. These are known as constants, and are much more powerful than constants in C. Constants are used throughout Swift to make code safer and clearer in intent when you work with values that don't need to change.

In addition to familiar types, Swift introduces advanced types not found in Objective-C, such as tuples. Tuples enable you to create and pass around groupings of values. You can use a tuple to return multiple values from a function as a single compound value.

Swift also introduces optional types, which handle the absence of a value. Optionals say either "there *is* a value, and it equals *x*" or "there *isn't* a value at all". Using optionals is similar to using `nil` with pointers in Objective-C, but they work for any type, not just classes. Not only are optionals safer and more expressive than `nil` pointers in Objective-C, they're at the heart of many of Swift's most powerful features.

Swift is a *type-safe* language, which means the language helps you to be clear about the types of values your code can work with. If part of your code requires a `String`, type safety prevents you from passing it an `Int` by mistake. Likewise, type safety prevents you from accidentally passing an optional `String` to a piece of code that requires a non-optional `String`. Type safety helps you catch and fix errors as early as possible in the development process.

Constants and Variables

Constants and variables associate a name (such as `maximumNumberOfLoginAttempts` or `welcomeMessage`) with a value of a particular type (such as the number `10` or the string `"Hello"`). The value of a *constant* can't be changed once it's set, whereas a *variable* can be set to a different value in the future.

Declaring Constants and Variables

Constants and variables must be declared before they're used. You declare constants with the `let` keyword and variables with the `var` keyword. Here's an example of how constants and variables can be used to track the number of login attempts a user has made:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

This code can be read as:

"Declare a new constant called `maximumNumberOfLoginAttempts`, and give it a value of `10`. Then, declare a new variable called `currentLoginAttempt`, and give it an initial value of `0`."

In this example, the maximum number of allowed login attempts is declared as a constant, because the maximum value never changes. The current login attempt counter is declared as a variable, because this value must be incremented after each failed login attempt.

You can declare multiple constants or multiple variables on a single line, separated by commas:

```
var x = 0.0, y = 0.0, z = 0.0
```

Type Annotations

You can provide a *type annotation* when you declare a constant or variable, to be clear about the kind of values the constant or variable can store. Write a type annotation by placing a colon after the constant or variable name, followed by a space, followed by the name of the type to use.

This example provides a type annotation for a variable called `welcomeMessage`, to indicate that the variable can store `String` values:

```
var welcomeMessage: String
```

The colon in the declaration means "…of type…," so the code above can be read as:

"Declare a variable called `welcomeMessage` that's of type `String`."

The phrase “of type `String`” means “can store any `String` value.” Think of it as meaning “the type of thing” (or “the kind of thing”) that can be stored.

The `welcomeMessage` variable can now be set to any string value without error:

```
welcomeMessage = "Hello"
```

You can define multiple related variables of the same type on a single line, separated by commas, with a single type annotation after the final variable name:

```
var red, green, blue: Double
```

Naming Constants and Variables

Constant and variable names can contain almost any character, including Unicode characters:

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Constant and variable names can’t contain whitespace characters, mathematical symbols, arrows, private-use Unicode scalar values, or line- and box-drawing characters. Nor can they begin with a number, although numbers may be included elsewhere within the name.

Once you’ve declared a constant or variable of a certain type, you can’t declare it again with the same name, or change it to store values of a different type. Nor can you change a constant into a variable or a variable into a constant.

You can change the value of an existing variable to another value of a compatible type. In this example, the value of `friendlyWelcome` is changed from `"Hello!"` to `"Bonjour!"`:

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"
```

Unlike a variable, the value of a constant can't be changed after it's set. Attempting to do so is reported as an error when your code is compiled:

```
let languageName = "Swift"
languageName = "Swift++"
// This is a compile-time error: languageName cannot be changed.
```

Printing Constants and Variables

You can print the current value of a constant or variable with the `print(_:separator:terminator:)` function:

```
print(friendlyWelcome)
// Prints "Bonjour!"
```

The `print(_:separator:terminator:)` function is a global function that prints one or more values to an appropriate output. In Xcode, for example, the `print(_:separator:terminator:)` function prints its output in Xcode's "console" pane. The `separator` and `terminator` parameter have default values, so you can omit them when you call this function. By default, the function terminates the line it prints by adding a line break. To print a value without a line break after it, pass an empty string as the terminator — for example, `print(someValue, terminator: "")`. For information about parameters with default values, see Default Parameter Values.

Swift uses *string interpolation* to include the name of a constant or variable as a placeholder in a longer string, and to prompt Swift to replace it with the current value of that constant or variable. Wrap the name in parentheses and escape it with a backslash before the opening parenthesis:

```
print("The current value of friendlyWelcome is \(friendlyWelcome)")
// Prints "The current value of friendlyWelcome is Bonjour!"
```

Use comments to include nonexecutable text in your code, as a note or reminder to yourself. Comments are ignored by the Swift compiler when your code is compiled.

Comments in Swift are very similar to comments in C. Single-line comments begin with two forward-slashes (`//`):

```
// This is a comment.
```

Multiline comments start with a forward-slash followed by an asterisk (`/*`) and end with an asterisk followed by a forward-slash (`*/`):

```
/* This is also a comment
but is written over multiple lines. */
```

Unlike multiline comments in C, multiline comments in Swift can be nested inside other multiline comments. You write nested comments by starting a multiline comment block and then starting a second multiline comment within the first block. The second block is then closed, followed by the first block:

```
/* This is the start of the first multiline comment.
    /* This is the second, nested multiline comment. */
This is the end of the first multiline comment. */
```

Nested multiline comments enable you to comment out large blocks of code quickly and easily, even if the code already contains multiline comments.

Semicolons

Unlike many other languages, Swift doesn't require you to write a semicolon (`;`) after each statement in your code, although you can do so if you wish. However, semicolons *are* required if you want to write multiple separate statements on a single line:

```
let cat = "🐱"; print(cat)
// Prints "🐱"
```

Integers

Integers are whole numbers with no fractional component, such as `42` and `-23`. Integers are either *signed* (positive, zero, or negative) or *unsigned* (positive or zero).

Swift provides signed and unsigned integers in 8, 16, 32, and 64 bit forms. These integers follow a naming convention similar to C, in that an 8-bit unsigned integer is of type `UInt8`, and a 32-bit signed integer is of type `Int32`. Like all types in Swift, these integer types have capitalized names.

Integer Bounds

You can access the minimum and maximum values of each integer type with its `min` and `max` properties:

```
let minValue = UInt8.min  // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max  // maxValue is equal to 255, and is of type UInt8
```

The values of these properties are of the appropriate-sized number type (such as `UInt8` in the example above) and can therefore be used in expressions alongside other values of the same type.

Int

In most cases, you don't need to pick a specific size of integer to use in your code. Swift provides an additional integer type, `Int`, which has the same size as the current platform's native word size:

- On a 32-bit platform, `Int` is the same size as `Int32`.
- On a 64-bit platform, `Int` is the same size as `Int64`.

Unless you need to work with a specific size of integer, always use `Int` for integer values in your code. This aids code consistency and interoperability. Even on 32-bit platforms, `Int` can store any value between `-2,147,483,648` and `2,147,483,647`, and is large enough for many integer ranges.

UInt

Swift also provides an unsigned integer type, `UInt`, which has the same size as the current platform's native word size:

- On a 32-bit platform, `UInt` is the same size as `UInt32`.
- On a 64-bit platform, `UInt` is the same size as `UInt64`.

Floating-Point Numbers

Floating-point numbers are numbers with a fractional component, such as `3.14159`, `0.1`, and `-273.15`.

Floating-point types can represent a much wider range of values than integer types, and can store numbers that are much larger or smaller than can be stored in an `Int`. Swift provides two signed floating-point number types:

- `Double` represents a 64-bit floating-point number.
- `Float` represents a 32-bit floating-point number.

Type Safety and Type Inference

Swift is a *type-safe* language. A type safe language encourages you to be clear about the types of values your code can work with. If part of your code requires a `String`, you can't pass it an `Int` by mistake.

Because Swift is type safe, it performs *type checks* when compiling your code and flags any mismatched types as errors. This enables you to catch and fix errors as early as possible in the development process.

Type-checking helps you avoid errors when you're working with different types of values. However, this doesn't mean that you have to specify the type of every constant and variable that you declare. If you don't specify the type of value you need, Swift uses *type inference* to work out the appropriate type. Type inference enables a compiler to deduce the type of a particular expression automatically when it compiles your code, simply by examining the values you provide.

Because of type inference, Swift requires far fewer type declarations than languages such as C or Objective-C. Constants and variables are still explicitly typed, but much of the work of specifying their type is done for you.

Type inference is particularly useful when you declare a constant or variable with an initial value. This is often done by assigning a *literal value* (or *literal*) to the constant or variable at the point that you declare it. (A literal value is a value that appears directly in your source code, such as `42` and `3.14159` in the examples below.)

For example, if you assign a literal value of `42` to a new constant without saying what type it is, Swift infers that you want the constant to be an `Int`, because you have initialized it with a number that looks like an integer:

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int
```

Likewise, if you don't specify a type for a floating-point literal, Swift infers that you want to create a `Double`:

```
let pi = 3.14159
// pi is inferred to be of type Double
```

Swift always chooses `Double` (rather than `Float`) when inferring the type of floating-point numbers.

If you combine integer and floating-point literals in an expression, a type of `Double` will be inferred from the context:

```
let anotherPi = 3 + 0.14159
// anotherPi is also inferred to be of type Double
```

The literal value of `3` has no explicit type in and of itself, and so an appropriate output type of `Double` is inferred from the presence of a floating-point literal as part of the addition.

Numeric Literals

Integer literals can be written as:

- A *decimal* number, with no prefix
- A *binary* number, with a `0b` prefix
- An *octal* number, with a `0o` prefix
- A *hexadecimal* number, with a `0x` prefix

All of these integer literals have a decimal value of `17`:

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 17 in binary notation
let octalInteger = 0o21           // 17 in octal notation
let hexadecimalInteger = 0x11     // 17 in hexadecimal notation
```

Floating-point literals can be decimal (with no prefix), or hexadecimal (with a `0x` prefix). They must always have a number (or hexadecimal number) on both sides of the decimal point. Decimal floats can also have an optional *exponent*, indicated by an uppercase or lowercase `e`; hexadecimal floats must have an exponent, indicated by an uppercase or lowercase `p`.

For decimal numbers with an exponent of `x`, the base number is multiplied by 10^x:

- `1.25e2` means 1.25×10^2, or `125.0`.
- `1.25e-2` means 1.25×10^{-2}, or `0.0125`.

For hexadecimal numbers with an exponent of `x`, the base number is multiplied by 2^x:

- `0xFp2` means 15×2^2, or `60.0`.
- `0xFp-2` means 15×2^{-2}, or `3.75`.

All of these floating-point literals have a decimal value of `12.1875`:

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

Numeric literals can contain extra formatting to make them easier to read. Both integers and floats can be padded with extra zeros and can contain underscores to help with readability. Neither type of formatting affects the underlying value of the literal:

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

Numeric Type Conversion

Use the `Int` type for all general-purpose integer constants and variables in your code, even if they’re known to be nonnegative. Using the default integer type in everyday situations means that integer constants and variables are immediately interoperable in your code and will match the inferred type for integer literal values.

Use other integer types only when they’re specifically needed for the task at hand, because of explicitly sized data from an external source, or for performance, memory usage, or other necessary optimization.

Using explicitly sized types in these situations helps to catch any accidental value overflows and implicitly documents the nature of the data being used.

Integer Conversion

The range of numbers that can be stored in an integer constant or variable is different for each numeric type. An `Int8` constant or variable can store numbers between `-128` and `127`, whereas a `UInt8` constant or variable can store numbers between `0` and `255`. A number that won't fit into a constant or variable of a sized integer type is reported as an error when your code is compiled:

```
let cannotBeNegative: UInt8 = -1
// UInt8 can't store negative numbers, and so this will report an error
let tooBig: Int8 = Int8.max + 1
// Int8 can't store a number larger than its maximum value,
// and so this will also report an error
```

Because each numeric type can store a different range of values, you must opt in to numeric type conversion on a case-by-case basis. This opt-in approach prevents hidden conversion errors and helps make type conversion intentions explicit in your code.

To convert one specific number type to another, you initialize a new number of the desired type with the existing value. In the example below, the constant `twoThousand` is of type `UInt16`, whereas the constant `one` is of type `UInt8`. They can't be added together directly, because they're not of the same type. Instead, this example calls `UInt16(one)` to create a new `UInt16` initialized with the value of `one`, and uses this value in place of the original:

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

Because both sides of the addition are now of type `UInt16`, the addition is allowed. The output constant (`twoThousandAndOne`) is inferred to be of type `UInt16`, because it's the sum of two `UInt16` values.

`SomeType(ofInitialValue)` is the default way to call the initializer of a Swift type and pass in an initial value. Behind the scenes, `UInt16` has an initializer that accepts a `UInt8` value, and so this initializer is used to make a new `UInt16` from an existing `UInt8`. You can't pass in *any* type here, however — it has to be a type for which `UInt16` provides an initializer. Extending existing types to provide initializers that accept new types (including your own type definitions) is covered in Extensions.

Integer and Floating-Point Conversion

Conversions between integer and floating-point numeric types must be made explicit:

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be of type Double
```

Here, the value of the constant `three` is used to create a new value of type `Double`, so that both sides of the addition are of the same type. Without this conversion in place, the addition would not be allowed.

Floating-point to integer conversion must also be made explicit. An integer type can be initialized with a `Double` or `Float` value:

```
let integerPi = Int(pi)
// integerPi equals 3, and is inferred to be of type Int
```

Floating-point values are always truncated when used to initialize a new integer value in this way. This means that `4.75` becomes `4`, and `-3.9` becomes `-3`.

Type Aliases

Type aliases define an alternative name for an existing type. You define type aliases with the `typealias` keyword.

Type aliases are useful when you want to refer to an existing type by a name that's contextually more appropriate, such as when working with data of a specific size from an external source:

```
typealias AudioSample = UInt16
```

Once you define a type alias, you can use the alias anywhere you might use the original name:

```
var maxAmplitudeFound = AudioSample.min
```

```
// maxAmplitudeFound is now 0
```

Here, `AudioSample` is defined as an alias for `UInt16`. Because it's an alias, the call to `AudioSample.min` actually calls `UInt16.min`, which provides an initial value of `0` for the `maxAmplitudeFound` variable.

Booleans

Swift has a basic *Boolean* type, called `Bool`. Boolean values are referred to as *logical*, because they can only ever be true or false. Swift provides two Boolean constant values, `true` and `false`:

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

The types of `orangesAreOrange` and `turnipsAreDelicious` have been inferred as `Bool` from the fact that they were initialized with Boolean literal values. As with `Int` and `Double` above, you don't need to declare constants or variables as `Bool` if you set them to `true` or `false` as soon as you create them. Type inference helps make Swift code more concise and readable when it initializes constants or variables with other values whose type is already known.

Boolean values are particularly useful when you work with conditional statements such as the `if` statement:

```
if turnipsAreDelicious {
    print("Mmm, tasty turnips!")
} else {
    print("Eww, turnips are horrible.")
}
// Prints "Eww, turnips are horrible."
```

Conditional statements such as the `if` statement are covered in more detail in Control Flow.

Swift's type safety prevents non-Boolean values from being substituted for `Bool`. The following example reports a compile-time error:

```
let i = 1
if i {
```

```
    // this example will not compile, and will report an error
}
```

However, the alternative example below is valid:

```
let i = 1
if i == 1 {
    // this example will compile successfully
}
```

The result of the `i == 1` comparison is of type `Bool`, and so this second example passes the type-check. Comparisons like `i == 1` are discussed in Basic Operators.

As with other examples of type safety in Swift, this approach avoids accidental errors and ensures that the intention of a particular section of code is always clear.

Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and don't have to be of the same type as each other.

In this example, `(404, "Not Found")` is a tuple that describes an *HTTP status code*. An HTTP status code is a special value returned by a web server whenever you request a web page. A status code of `404 Not Found` is returned if you request a webpage that doesn't exist.

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

The `(404, "Not Found")` tuple groups together an `Int` and a `String` to give the HTTP status code two separate values: a number and a human-readable description. It can be described as "a tuple of type `(Int, String)`".

You can create tuples from any permutation of types, and they can contain as many different types as you like. There's nothing stopping you from having a tuple of type `(Int, Int, Int)`, or `(String, Bool)`, or indeed any other permutation you require.

You can *decompose* a tuple's contents into separate constants or variables, which you then access as usual:

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Prints "The status code is 404"
print("The status message is \(statusMessage)")
// Prints "The status message is Not Found"
```

If you only need some of the tuple's values, ignore parts of the tuple with an underscore (`_`) when you decompose the tuple:

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404"
```

Alternatively, access the individual element values in a tuple using index numbers starting at zero:

```
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
print("The status message is \(http404Error.1)")
// Prints "The status message is Not Found"
```

You can name the individual elements in a tuple when the tuple is defined:

```
let http200Status = (statusCode: 200, description: "OK")
```

If you name the elements in a tuple, you can use the element names to access the values of those elements:

```
print("The status code is \(http200Status.statusCode)")
// Prints "The status code is 200"
print("The status message is \(http200Status.description)")
// Prints "The status message is OK"
```

Tuples are particularly useful as the return values of functions. A function that tries to retrieve a web page might return the `(Int, String)` tuple type to describe the success or failure of the page retrieval. By returning a tuple with two distinct values, each of a different type, the function provides more useful information about its outcome than if it could only return a single value of a single type. For more information, see Functions with Multiple Return Values.

Optionals

You use *optionals* in situations where a value may be absent. An optional represents two possibilities: Either there *is* a value, and you can unwrap the optional to access that value, or there *isn't* a value at all.

Here's an example of how optionals can be used to cope with the absence of a value. Swift's `Int` type has an initializer which tries to convert a `String` value into an `Int` value. However, not every string can be converted into an integer. The string `"123"` can be converted into the numeric value `123`, but the string `"hello, world"` doesn't have an obvious numeric value to convert to.

The example below uses the initializer to try to convert a `String` into an `Int`:

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?", or "optional Int"
```

Because the initializer might fail, it returns an *optional* `Int`, rather than an `Int`. An optional `Int` is written as `Int?`, not `Int`. The question mark indicates that the value it contains is optional, meaning that it might contain *some* `Int` value, or it might contain *no value at all*. (It can't contain anything else, such as a `Bool` value or a `String` value. It's either an `Int`, or it's nothing at all.)

nil

You set an optional variable to a valueless state by assigning it the special value `nil`:

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

If you define an optional variable without providing a default value, the variable is automatically set to `nil` for you:

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

If Statements and Forced Unwrapping

You can use an `if` statement to find out whether an optional contains a value by comparing the optional against `nil`. You perform this comparison with the "equal to" operator (`==`) or the "not equal to" operator (`!=`).

If an optional has a value, it's considered to be "not equal to" `nil`:

```
if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}
// Prints "convertedNumber contains some integer value."
```

Once you're sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation point (`!`) to the end of the optional's name. The exclamation point effectively says, "I know that this optional definitely has a value; please use it." This is known as *forced unwrapping* of the optional's value:

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

For more about the `if` statement, see Control Flow.

Optional Binding

You use *optional binding* to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable. Optional binding can be used with `if` and `while` statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action. `if` and `while` statements are described in more detail in Control Flow.

Write an optional binding for an `if` statement as follows:

```
if let <#constantName#> = <#someOptional#> {
   <#statements#>
}
```

You can rewrite the `possibleNumber` example from the Optionals section to use optional binding rather than forced unwrapping:

```
if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\" has an integer value of \(actualN
umber)")
} else {
    print("The string \"\(possibleNumber)\" couldn't be converted to an integ
er")
}
// Prints "The string "123" has an integer value of 123"
```

This code can be read as:

"If the optional `Int` returned by `Int(possibleNumber)` contains a value, set a new constant called `actualNumber` to the value contained in the optional."

If the conversion is successful, the `actualNumber` constant becomes available for use within the first branch of the `if` statement. It has already been initialized with the value contained *within* the optional, and so you don't use the `!` suffix to access its value. In this example, `actualNumber` is simply used to print the result of the conversion.

If you don't need to refer to the original, optional constant or variable after accessing the value it contains, you can use the same name for the new constant or variable:

```
let myNumber = Int(possibleNumber)
// Here, myNumber is an optional integer
if let myNumber = myNumber {
    // Here, myNumber is a non-optional integer
```

```
    print("My number is \(myNumber)")
}
// Prints "My number is 123"
```

This code starts by checking whether `myNumber` contains a value, just like the code in the previous example. If `myNumber` has a value, the value of a new constant named `myNumber` is set to that value. Inside the body of the `if` statement, writing `myNumber` refers to that new non-optional constant. Before the beginning of the `if` statement and after its end, writing `myNumber` refers to the optional integer constant.

Because this kind of code is so common, you can use a shorter spelling to unwrap an optional value: write just the name of the constant or variable that you're unwrapping. The new, unwrapped constant or variable implicitly uses the same name as the optional value.

```
if let myNumber {
    print("My number is \(myNumber)")
}
// Prints "My number is 123"
```

You can use both constants and variables with optional binding. If you wanted to manipulate the value of `myNumber` within the first branch of the `if` statement, you could write `if var myNumber` instead, and the value contained within the optional would be made available as a variable rather than a constant. Changes you make to `myNumber` inside the body of the `if` statement apply only to that local variable, *not* to the original, optional constant or variable that you unwrapped.

You can include as many optional bindings and Boolean conditions in a single `if` statement as you need to, separated by commas. If any of the values in the optional bindings are `nil` or any Boolean condition evaluates to `false`, the whole `if` statement's condition is considered to be `false`. The following `if` statements are equivalent:

```
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber < se
condNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// Prints "4 < 42 < 100"

if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
```

```
        }
    }
}
// Prints "4 < 42 < 100"
```

Implicitly Unwrapped Optionals

As described above, optionals indicate that a constant or variable is allowed to have “no value”. Optionals can be checked with an `if` statement to see if a value exists, and can be conditionally unwrapped with optional binding to access the optional’s value if it does exist.

Sometimes it’s clear from a program’s structure that an optional will *always* have a value, after that value is first set. In these cases, it’s useful to remove the need to check and unwrap the optional’s value every time it’s accessed, because it can be safely assumed to have a value all of the time.

These kinds of optionals are defined as *implicitly unwrapped optionals*. You write an implicitly unwrapped optional by placing an exclamation point (`String!`) rather than a question mark (`String?`) after the type that you want to make optional. Rather than placing an exclamation point after the optional’s name when you use it, you place an exclamation point after the optional’s type when you declare it.

Implicitly unwrapped optionals are useful when an optional’s value is confirmed to exist immediately after the optional is first defined and can definitely be assumed to exist at every point thereafter. The primary use of implicitly unwrapped optionals in Swift is during class initialization, as described in Unowned References and Implicitly Unwrapped Optional Properties.

An implicitly unwrapped optional is a normal optional behind the scenes, but can also be used like a non-optional value, without the need to unwrap the optional value each time it’s accessed. The following example shows the difference in behavior between an optional string and an implicitly unwrapped optional string when accessing their wrapped value as an explicit `String`:

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation point

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation poin
t
```

You can think of an implicitly unwrapped optional as giving permission for the optional to be force-unwrapped if needed. When you use an implicitly unwrapped optional value, Swift first tries to use it as an ordinary optional value; if it can’t be used as an optional, Swift force-unwraps the value. In the code above, the optional value `assumedString` is force-unwrapped before assigning its value to `implicitString` because `implicitString` has an explicit, non-optional type of `String`. In code below, `optionalString` doesn't have an explicit type so it's an ordinary optional.

```
let optionalString = assumedString
// The type of optionalString is "String?" and assumedString isn't force-unwr
apped.
```

If an implicitly unwrapped optional is `nil` and you try to access its wrapped value, you'll trigger a runtime error. The result is exactly the same as if you place an exclamation point after a normal optional that doesn't contain a value.

You can check whether an implicitly unwrapped optional is `nil` the same way you check a normal optional:

```
if assumedString != nil {
    print(assumedString!)
}
// Prints "An implicitly unwrapped optional string."
```

You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

```
if let definiteString = assumedString {
    print(definiteString)
}
// Prints "An implicitly unwrapped optional string."
```

Error Handling

You use *error handling* to respond to error conditions your program may encounter during execution.

In contrast to optionals, which can use the presence or absence of a value to communicate success or failure of a function, error handling allows you to determine the underlying cause of failure, and, if necessary, propagate the error to another part of your program.

When a function encounters an error condition, it *throws* an error. That function's caller can then *catch* the error and respond appropriately.

```
func canThrowAnError() throws {
    // this function may or may not throw an error
}
```

A function indicates that it can throw an error by including the `throws` keyword in its declaration. When you call a function that can throw an error, you prepend the `try` keyword to the expression.

Swift automatically propagates errors out of their current scope until they're handled by a `catch` clause.

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

A `do` statement creates a new containing scope, which allows errors to be propagated to one or more `catch` clauses.

Here's an example of how error handling can be used to respond to different error conditions:

```
func makeASandwich() throws {
    // ...
}

do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

In this example, the `makeASandwich()` function will throw an error if no clean dishes are available or if any ingredients are missing. Because `makeASandwich()` can throw an error, the function call is wrapped in a `try` expression. By wrapping the function call in a `do` statement, any errors that are thrown will be propagated to the provided `catch` clauses.

If no error is thrown, the `eatASandwich()` function is called. If an error is thrown and it matches the `SandwichError.outOfCleanDishes` case, then the `washDishes()` function will be called. If an error is thrown and it matches the `SandwichError.missingIngredients` case, then the `buyGroceries(_:)` function is called with the associated `[String]` value captured by the `catch` pattern.

Throwing, catching, and propagating errors is covered in greater detail in Error Handling.

Assertions and Preconditions

Assertions and *preconditions* are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code. If the Boolean condition in the assertion or precondition evaluates to `true`, code execution continues as usual. If the condition evaluates to `false`, the current state of the program is invalid; code execution ends, and your app is terminated.

You use assertions and preconditions to express the assumptions you make and the expectations you have while coding, so you can include them as part of your code. Assertions help you find mistakes and incorrect assumptions during development, and preconditions help you detect issues in production.

In addition to verifying your expectations at runtime, assertions and preconditions also become a useful form of documentation within the code. Unlike the error conditions discussed in Error Handling above, assertions and preconditions aren't used for recoverable or expected errors. Because a failed assertion or precondition indicates an invalid program state, there's no way to catch a failed assertion.

Using assertions and preconditions isn't a substitute for designing your code in such a way that invalid conditions are unlikely to arise. However, using them to enforce valid data and state causes your app to terminate more predictably if an invalid state occurs, and helps make the problem easier to debug. Stopping execution as soon as an invalid state is detected also helps limit the damage caused by that invalid state.

The difference between assertions and preconditions is in when they're checked: Assertions are checked only in debug builds, but preconditions are checked in both debug and production builds. In production builds, the condition inside an assertion isn't evaluated. This means you can use as many assertions as you want during your development process, without impacting performance in production.

Debugging with Assertions

You write an assertion by calling the `assert(_:_:file:line:)` function from the Swift standard library. You pass this function an expression that evaluates to `true` or `false` and a message to display if the result of the condition is `false`. For example:

```
let age = -3
assert(age >= 0, "A person's age can't be less than zero.")
// This assertion fails because -3 isn't >= 0.
```

In this example, code execution continues if `age >= 0` evaluates to `true`, that is, if the value of `age` is nonnegative. If the value of `age` is negative, as in the code above, then `age >= 0` evaluates to `false`, and the assertion fails, terminating the application.

You can omit the assertion message — for example, when it would just repeat the condition as prose.

```
assert(age >= 0)
```

If the code already checks the condition, you use the `assertionFailure(_:file:line:)` function to indicate that an assertion has failed. For example:

```
if age > 10 {
    print("You can ride the roller-coaster or the ferris wheel.")
} else if age >= 0 {
    print("You can ride the ferris wheel.")
} else {
    assertionFailure("A person's age can't be less than zero.")
}
```

Enforcing Preconditions

Use a precondition whenever a condition has the potential to be false, but must *definitely* be true for your code to continue execution. For example, use a precondition to check that a subscript isn't out of bounds, or to check that a function has been passed a valid value.

You write a precondition by calling the `precondition(_:_:file:line:)` function. You pass this function an expression that evaluates to `true` or `false` and a message to display if the result of the condition is `false`. For example:

```
// In the implementation of a subscript...
precondition(index > 0, "Index must be greater than zero.")
```

You can also call the `preconditionFailure(_:file:line:)` function to indicate that a failure has occurred — for example, if the default case of a switch was taken, but all valid input data should have been handled by one of the switch's other cases.