Swift 4

Day 1

The Swift Programming Language

- Development of Swift Started in 2010.
- Version 1.0 was released in September 2014.
- Version 4.0 was released in September 2017.
- During its introduction, it was described simply as "Objective-C without the C”.
- Swift is open source: https://github.com/apple/swift
- Swift Evolution: https://apple.github.io/swift-evolution/

The Swift Programming Language

- Swift provides its own versions of all fundamental C and Objective-C types: `Int` for integers, `Double` and `Float` for floating-point values, `Bool` for Boolean values, and `String` for textual data.

- Swift also introduces *optional* types, which handle the absence of a value. Optionals are safer and more expressive than `nil` pointers in Objective-C.

- Swift is *type-safe*, which means the language helps you to be clear about the types of values your code can work with.

Constants and Variables

- Constants and variables must be declared before they are used.
- Declare constants with the `let` keyword and variables with the `var` keyword:

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

- Multiple constants or multiple variables can be declared on a single line, separated by commas:

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

- You can provide a *type annotation* when you declare a constant or variable:

```
var welcomeMessage: String
```

Printing Constants and Variables

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

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

- Use *string interpolation* to include the name of a constant or variable as a placeholder in a longer string:

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

Comments

- 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. */
```

Semicolons

- Swift doesn't require a semicolon (;) after each statement in your code. However, semicolons *are* required if you want to write multiple separate statements on a single line:

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

Integers

- 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: an 8-bit unsigned integer is of type `UInt8`, and a 32-bit signed integer is of type `Int32`.

- 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`, while on a 64-bit platform, `Int` is the same size as `Int64`.

Floating-Point Numbers

- 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. If part of your code requires a `String`, you can't pass it an `Int` by mistake.

- Swift compiler performs *type checks* when compiling your code and flags any mismatched types as errors.

- If you don't specify the type of value you need, Swift uses *type inference* to work out the appropriate type.

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

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

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

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
```

- 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.

```
typealias AudioSample = UInt16
```

- Type aliases are useful when you want to refer to an existing type by a name that is more appropriate:

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

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`.

- Swift's type safety prevents non-Boolean values from being substituted for `Bool`.

```
let i = 1
if i {
    // this example will not compile, and will report an error
}

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

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.

- The following tuple describes an *HTTP status code:*

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

- *This 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)`".

Tuples, cont'd

- 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"
```

Tuples, cont'd

- 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, then you can use the element names to access the values of those elements:

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

- 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.

Optionals

- *Optionals* are used 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.

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

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
```

- You can’t use `nil` with non-optional constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.

nil, cont'd

- 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
```

- Swift's `nil` isn't the same as `nil` in Objective-C. In Objective-C, `nil` is a pointer to a non-existent object. In Swift, `nil` isn't a pointer—it's the absence of a value of a certain type. Optionals of *any* type can be set to `nil`, not just object types.

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 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 mark (`!`) to the end of the optional’s name.

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

Optional Binding

- 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.

```
if let actualNumber = Int(possibleNumber) {
    print("\"\(possibleNumber)\" has an integer value of \(actualNumber)")
} else {
    print("\"\(possibleNumber)\" could not be converted to an integer")
}
// Prints ""123" has an integer value of 123"
```

- If the conversion is successful, the `actualNumber` constant becomes available for use within the first branch of the `if` statement.

Optional Binding, cont'd

- 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`.

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

- Constants and variables created with optional binding in an `if` statement are available only within the body of the `if` statement.

Implicitly Unwrapped Optionals

- 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

- These kinds of optionals are defined as *implicitly unwrapped optionals*. You write an implicitly unwrapped optional by placing an exclamation mark (`String!`) rather than a question mark (`String?`) after the type that you want to make optional.

Implicitly Unwrapped Optionals, cont'd

- 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.

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

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

- 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 mark after a normal optional that doesn't contain a value.

Implicitly Unwrapped Optionals, cont'd

- You can still treat an implicitly unwrapped optional like a normal optional, to check if it contains a value:

```
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."
```

- Don't use an implicitly unwrapped optional when there's a possibility of a variable becoming `nil` at a later point.

Error Handling

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

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

- A function indicates that it can throw an error by including the `throws` keyword in its declaration.

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

Error Handling, cont'd

- 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
}
```

Error Handling, cont'd

- Example of how error handling can be used to respond to different errors:

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

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

- 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.

Basic Operators

- Swift supports most standard C operators

- Unlike C, the assignment operator (`=`) doesn't return a value, to prevent it from being mistakenly used when the equal to operator (`==`) is intended.

- Arithmetic operators (`+`, `-`, `*`, `/`, `%` and so forth) detect and disallow value overflow, to avoid unexpected results when working with numbers that become larger or smaller than the allowed value range of the type that stores them.

- Swift also provides range operators that aren't found in C, such as `a..<b` and `a...b`, as a shortcut for expressing a range of values.

Basic Operators, cont'd

- *Unary* operators operate on a single target (such as `-a`). Unary *prefix* operators appear immediately before their target (such as `!b`), and unary *postfix* operators appear immediately after their target (such as `c!`).

- *Binary* operators operate on two targets (such as `2 + 3`) and are *infix* because they appear in between their two targets.

- *Ternary* operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (`a ? b : c`).

Assignment Operator

- The *assignment operator* (`a = b`) initializes or updates the value of `a` with the value of `b`:

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

- If the right side of the assignment is a tuple with multiple values, its elements can be decomposed into multiple constants or variables at once:

```
let (x, y) = (1, 2)
// x is equal to 1, and y is equal to 2
```

- Unlike the assignment operator in C and Objective-C, the assignment operator in Swift does not itself return a value.

```
if x = y {
    // This is not valid, because x = y does not return a value.
}
```

Arithmetic Operators

- Swift supports the four standard *arithmetic operators* for all number types:

```
1 + 2       // equals 3
5 - 3       // equals 2
2 * 3       // equals 6
10.0 / 2.5  // equals 4.0
```

- Unlike the arithmetic operators in C and Objective-C, the Swift arithmetic operators don’t allow values to overflow by default.

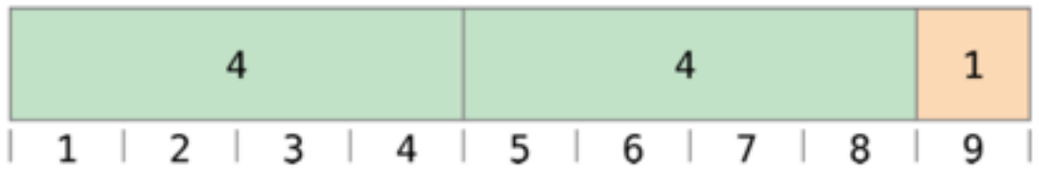
- The addition operator is also supported for `String` concatenation:

```
"hello, " + "world"  // equals "hello, world"
```

Remainder Operator

- The *remainder operator* (`a % b`) works out how many multiples of `b` will fit inside `a` and returns the value that is left over (known as the *remainder*).

- To calculate `9 % 4`, you first work out how many `4`s will fit inside `9`:




```
9 % 4    // equals 1
```

- The same method is applied when calculating the remainder for a negative value of `a`:

```
-9 % 4   // equals -1
```

Unary Minus & Plus Operators

- The sign of a numeric value can be toggled using a prefixed –, known as the *unary minus operator*:

```
let three = 3
let minusThree = -three       // minusThree equals -3
let plusThree = -minusThree   // plusThree equals 3, or "minus minus three"
```

- The *unary minus* operator (–) is prepended directly before the value it operates on, without any white space.

- The *unary plus operator* (+) simply returns the value it operates on, without any change:

```
let minusSix = -6
let alsoMinusSix = +minusSix  // alsoMinusSix equals -6
```

Compound Assignment Operators

- Like C, Swift provides *compound assignment operators* that combine assignment (=) with another operation. One example is the *addition assignment operator* (+=):

```
var a = 1
a += 2
// a is now equal to 3
```

- The expression `a += 2` is shorthand for `a = a + 2`. Effectively, the addition and the assignment are combined into one operator that performs both tasks at the same time.

- The compound assignment operators don’t return a value. For example, you can’t write `let b = a += 2`.

Comparison Operators

- Swift supports all standard C *comparison operators*:

```
1 == 1   // true because 1 is equal to 1
2 != 1   // true because 2 is not equal to 1
2 > 1    // true because 2 is greater than 1
1 < 2    // true because 1 is less than 2
1 >= 1   // true because 1 is greater than or equal to 1
2 <= 1   // false because 2 is not less than or equal to 1
```

- Swift also provides two *identity operators* (`===` and `!==`), which you use to test whether two object references both refer to the same object instance.

Comparison Operators, cont'd

- You can compare two tuples if they have the same type and the same number of values. Tuples are compared from left to right, one value at a time, until the comparison finds two values that aren't equal.

```
(1, "zebra") < (2, "apple")   // true because 1 is less than 2; "zebra" and "apple"
  are not compared
(3, "apple") < (3, "bird")    // true because 3 is equal to 3, and "apple" is less
  than "bird"
(4, "dog") == (4, "dog")      // true because 4 is equal to 4, and "dog" is equal
  to "dog"
```

- Tuples can be compared with a given operator only if the operator can be applied to each value in the respective tuples.

```
("blue", -1) < ("purple", 1)        // OK, evaluates to true
("blue", false) < ("purple", true)  // Error because < can't compare Boolean values
```

Ternary Conditional Operator

- The *ternary conditional operator* is a special operator with three parts, which takes the form `question ? answer1 : answer2`.

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight is equal to 90
```

- Avoid combining multiple instances of the ternary conditional operator into one compound statement.

Nil-Coalescing Operator

- The *nil-coalescing operator* (`a ?? b`) unwraps an optional `a` if it contains a value, or returns a default value `b` if `a` is `nil`. The expression `a` is always of an optional type. The expression `b` must match the type that is stored inside `a`.

```
let defaultColorName = "red"
var userDefinedColorName: String?   // defaults to nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is nil, so colorNameToUse is set to the default of "red"
```

- If the value of `a` is non-`nil`, the value of `b` is not evaluated. This is known as *short-circuit evaluation*.

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is not nil, so colorNameToUse is set to "green"
```

Range Operators

- The *closed range operator* (`a...b`) defines a *CountableClosedRange* that runs from `a` to `b`, and includes the values `a` and `b`. The value of `a` must not be greater than `b`.

- The *half-open range operator* (`a..<b`) defines a *CountableRange* that runs from `a` to `b`, but doesn't include `b`. It's said to be *half-open* because it contains its first value, but not its final value.

- The closed range operator has an alternative form for ranges that continue as far as possible in one direction, called *CountablePartialRange*:

```
for name in names[2...] {
    print(name)
}

for name in names[...2] {
    print(name)
}
```

Logical Operators

- The *logical NOT operator* (`!a`) inverts a Boolean value so that `true` becomes `false`, and `false` becomes `true`. This operator is a prefix operator, and appears immediately before the value it operates on, without any white space.

- The *logical AND operator* (`a && b`) is an infix operator that creates logical expressions where both values must be `true` for the overall expression to also be `true`.

- The *logical OR operator* (`a || b`) is an infix operator made from two adjacent pipe characters. You use it to create logical expressions in which only *one* of the two values has to be `true` for the overall expression to be `true`.

String Literals

- You can include predefined `String` values within your code as *string literals*. A string literal is a sequence of characters surrounded by double quotation marks (“).

```
let someString = "Some string literal value"
```

- If you need a string that spans several lines, use a multiline string literal—a sequence of characters surrounded by three double quotation marks:

```
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

-

Initializing an Empty String

- To create an empty `String` value as the starting point for building a longer string, either assign an empty string literal to a variable, or initialize a new `String` instance with initializer syntax:

```
var emptyString = ""               // empty string literal
var anotherEmptyString = String()  // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

- Find out whether a `String` value is empty by checking its Boolean `isEmpty` property:

```
if emptyString.isEmpty {
    print("Nothing to see here")
}
// Prints "Nothing to see here"
```

String Mutability

- You indicate whether a particular `String` can be modified (or *mutated*) by assigning it to a variable (in which case it can be modified), or to a constant (in which case it can't be modified):

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"

let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error - a constant string cannot be modified
```

- This approach is different from string mutation in Objective-C and Cocoa, where you choose between two classes (`NSString` and `NSMutableString`) to indicate whether a string can be mutated.

Strings Are Value Types

- If you create a new `String` value, that `String` value is *copied* when it's passed to a function or method, or when it's assigned to a constant or variable.

- In each case, a new copy of the existing `String` value is created, and the new copy is passed or assigned, not the original version.

- Behind the scenes, Swift's compiler optimizes string usage so that actual copying takes place only when absolutely necessary.

Collection Types

- Swift provides three primary *collection types*, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.

- If you create an array, a set, or a dictionary, and assign it to a variable, the collection that is created will be *mutable*. This means that you can change (or *mutate*) the collection after it's created by adding, removing, or changing items in the collection.

- If you assign an array, a set, or a dictionary to a constant, that collection is *immutable*, and its size and contents cannot be changed.

Arrays

- An *array* stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

- The type of a Swift array is written in full as `Array<Element>`.

- You can also write the type of an array in shorthand form as `[Element]`.

- You can create an empty array of a certain type using initializer syntax:

```
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
// Prints "someInts is of type [Int] with 0 items."
```

Arrays, cont'd

- You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array's type is inferred from the type of the two arrays you add together:

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

- You can also initialize an array with an *array literal*, which is a shorthand way to write one or more values as an array collection.

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

Arrays, cont'd

- You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array's type is inferred from the type of the two arrays you add together:

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

- You can also initialize an array with an *array literal*, which is a shorthand way to write one or more values as an array collection.

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

- nUse the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
```

Arrays, cont'd

- You can add a new item to the end of an array by calling the array's `append(_:)` method:

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

- Alternatively, append an array of one or more compatible items with the addition assignment operator (`+=`):

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
```

Arrays, cont’d

- You can iterate over the entire set of values in an array with the `for-in` loop:

```
for item in shoppingList {
    print(item)
}
```

- If you need the integer index of each item as well as its value, use the `enumerated()` method to iterate over the array instead. For each item in the array, the `enumerated()` method returns a tuple composed of an integer and the item.

```
for (index, value) in shoppingList.enumerated() {
    print("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
// Item 5: Bananas
```

Sets

- A *set* stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items is not important, or when you need to ensure that an item only appears once.

- A type must be *hashable* in order to be stored in a set—that is, the type must provide a way to compute a *hash value* for itself. A hash value is an `Int` value that is the same for all objects that compare equally, such that if `a == b`, it follows that `a.hashValue == b.hashValue`.

- All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default, and can be used as set value types or dictionary key types.

- You can create an empty set of a certain type using initializer syntax:

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \(letters.count) items.")
// Prints "letters is of type Set<Character> with 0 items."
```

- You can also initialize a set with an array literal, as a shorthand way to write one or more values as a set collection.

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
```

Sets, cont'd

- A set type cannot be inferred from an array literal alone, so the type `Set` must be explicitly declared. However, because of Swift's type inference, you don't have to write the type of the set if you're initializing it with an array literal containing values of the same type.

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

- Swift's `Set` type does not have a defined ordering. To iterate over the values of a set in a specific order, use the `sorted()` method, which returns the set's elements as an array sorted using the `<` operator.

```
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

Dictionaries

- A *dictionary* stores associations between keys of the same type and values of the same type in a collection with no defined ordering.

- The type of a Swift dictionary is written in full as `Dictionary<Key, Value>`

- A dictionary `Key` type must conform to the `Hashable` protocol, like a set's value type.

- You can also write the type of a dictionary in shorthand form as `[Key: Value]`.

- You can create an empty `Dictionary` of a certain type by using initializer syntax:

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers is an empty [Int: String] dictionary
```

- You can also initialize a dictionary with a *dictionary literal*, which has a similar syntax to the array literal:

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]

var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Dictionaries, cont'd

- You can iterate over the key-value pairs in a dictionary with a `for-in` loop. Each item in the dictionary is returned as a `(key, value)` tuple:

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// YYZ: Toronto Pearson
// LHR: London Heathrow
```

- You can also retrieve an iterable collection of a dictionary's keys or values by accessing its `keys` and `values` properties:

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
// Airport code: YYZ
// Airport code: LHR

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
// Airport name: Toronto Pearson
// Airport name: London Heathrow
```

Dictionaries, cont'd

- If you need to use a dictionary's keys or values with an API that takes an `Array` instance, initialize a new array with the `keys` or `values` property:

```
let airportCodes = [String](airports.keys)
// airportCodes is ["YYZ", "LHR"]
```

```
let airportNames = [String](airports.values)
// airportNames is ["Toronto Pearson", "London Heathrow"]
```

- Swift's `Dictionary` type does not have a defined ordering. To iterate over the keys or values of a dictionary in a specific order, use the `sorted()` method on its `keys` or `values` property.

Control Flow

- You use the `for-in` loop to iterate over a sequence, such as items in an array, ranges of numbers, or characters in a string.

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
```

- You can also iterate over a dictionary to access its key-value pairs. Each item in the dictionary is returned as a `(key, value)` tuple.

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
```

- You can also use `for-in` loops with numeric ranges.

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
```

Control Flow, cont'd

- In the previous example, `index` is a constant whose value is automatically set at the start of each iteration of the loop. As such, `index` does not have to be declared before it is used. It is implicitly declared simply by its inclusion in the loop declaration

- You can ignore the values by using an underscore in place of a variable name.

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
```

Control Flow, cont'd

- A `while` loop performs a set of statements until a condition becomes `false`. These kinds of loops are best used when the number of iterations is not known before the first iteration begins. You can ignore the values by using an underscore in place of a variable name.

- A `while` loop starts by evaluating a single condition. If the condition is `true`, a set of statements is repeated until the condition becomes `false`.

```
while condition {
    statements
}
```

- The other variation of the `while` loop, known as the `repeat-while` loop, performs a single pass through the loop block first, *before* considering the loop's condition.

```
repeat {
    statements
} while condition
```

Control Flow, cont'd

- The `if` statement has a single `if` condition. It executes a set of statements only if that condition is `true`.

- The `if` statement can provide an alternative set of statements, known as an *else clause*, for situations when the `if` condition is `false`. These statements are indicated by the `else` keyword.

- You can chain multiple `if` statements together to consider additional clauses.

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
```

- The final `else` clause is optional, however, and can be excluded if the set of conditions does not need to be complete.

Lab

- Work through all the examples in the previous slides

Swift 4

Day 2