A Swift Introduction

Part One

Hello World

```swift
print("Hello, world!”)
```

In Swift, this line of code is a complete program. You don’t need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don’t need a `main()` function. You also don’t need to write semicolons at the end of every statement 🎉🎉.

Import Declaration

- An **import declaration** allows access to symbols that are declared outside the current file
- The basic form imports the entire module

```
import module
```

- Ex:

```swift
import Foundation
```

- Providing more detail limits which symbols are imported

```
import import kind module.symbol name
```

- Ex:

```swift
import struct CoreGraphics.CGFloat
```

Values and Types

Simple Values

- Use `let` to make a constant (immutable) and `var` to make a variable (mutable)

```swift
var myVariable = 42
myVariable = 50
let myConstant = 42
```

- The value of a constant doesn't need to be known at compile time, but you must assign it a value exactly once

- This means you can use constants to name a value that you determine once but use in many places

Type Inference

- A constant or variable must have the same type as the value you want to assign to it
- Constants and values can be
  - **Implicitly inferred** by the compiler
    - Providing a value when you create a constant or variable lets the compiler infer its type

```swift
let implicitInteger = 70
let implicitDouble = 70.0
```

  - **Declared explicitly**:
    - If the initial value doesn’t provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon

```swift
let explicitDouble: Double = 70
```

Built-In Types

- Two kinds of types:
  - `named types`
  - `compound types`

Named Types

- A type that can be given a particular name when it's defined

- Includes `classes`, `structures`, `enumerations`, and `protocols`

- Data types that are normally considered basic or primitive in other languages—such as types that represent numbers, characters, and strings—are actually named types

```swift
let projected4220Grade: Character = "A"
let language: String = "Swift"
let yearIntroduced: Int = 2014
let currentVersion: Double = 5.1
let isOpenSource: Bool = true
let pi: Float = 3.14159265359
```

Named Types

- The Swift standard library defines many commonly used named types
- Including those that represent
    - Arrays
    - Sets
- Although we will cover these types more in-depth later on in this course, the following lines provide examples of each:

```swift
let shoppingListArray: Array<String> = ["bread", "milk", "eggs", "coffee"]
let shoppingListSet: Set<String> = ["bread", "milk", "eggs", "coffee"]
let colorsByFeeling: Dictionary<String, String> = ["happy": "yellow", "sad": "blue", "angry": "red"]
let tomorrowIsASnowDay: Bool? // Optional
```

Compound Types

- A type without name
- **Two** compound types:
    - function types
    - tuple types

Tuple Types

- A comma-separated list of types enclosed in parentheses
- Elements can be named

```
var someTuple = (top: 10, bottom: 12)  // someTuple is of type (top: Int, bottom: Int)
someTuple = (top: 4, bottom: 42)       // OK: names match
someTuple = (9, 99)                    // OK: names are inferred
someTuple = (left: 5, right: 5)        // Error: names don't match
```

- All tuple types contain two or more types, except for `Void` which is a type alias for the empty tuple type, `()`

Function Types

- Represents the type of a function, method, or closure and consists of a parameter and return type separated by an arrow (–>)

```
(parameter type) -> return type
```

- More on functions in Part Three!

Type Conversion

- Values are never implicitly converted to another type
- If you need to convert a value to a different type, explicitly make an instance of the desired type

```
let label = "The width is "
let width = 94
let labelWidth = label + String(width)
```

String Interpolation

- Using a `+` operator to combine two or more `String` objects is called **string concatenation**
- Another method is to utilize **string interpolation**:
    - Write the value in parentheses, and write a backslash (`\`) before the parentheses

```
let labelWidth = "\(label) \(width)"
```

Collection Types

- Swift provides three primary collection types for storing collections of values
    - **Arrays**: Ordered collections of values
    - **Sets**: Unordered collections of unique values
    - **Dictionaries**: Unordered collections of key-value associations
- These types are always clear about the kinds of values and keys that they can store
- A value of the wrong type cannot be inserted into a collection by mistake

Arrays

- Written in full as `Array<Element>`, where `Element` is the type of values the array is allowed to store
- Written in shorthand form as `[Element]`
- The two forms are functionally identical
- The shorthand form is preferred
- Create by using brackets (`[]`)
- Access elements by writing the index in the brackets
- Zero (`0`) indexed

```
var shoppingList = ["catfish", "water", "tulips"]
shoppingList[1] = "bottle of water"
```

Arrays

- Automatically grow as elements are added

```
shoppingList.append("blue paint")
```

- To create an empty array, use the initializer syntax

```
let emptyArray = [String]()
```

- If type information can be inferred, write an empty array as `[]`—for example, when you set a new value for a variable or pass an argument to a function

```
shoppingList = []
```

- Alternatively when **declaring** an empty array

```
var newEmptyArray: [String] = []
```

Sets

- Much like arrays, with two primary differences typically encountered:
  - Elements are unordered
  - Do not contain duplicate values

Dictionaries
(aka Associative Arrays or Hash Maps)

- The long form of the type's declaration is written as `Dictionary<Key, Value>` where:
- `Key` is a unique, hashable type that is used to retrieve a corresponding value
- `Value` corresponds to the type of object stored.
- Shorthand form can be represented as [Key: Value].
- Create dictionaries using brackets (`[:]`)
- Access their elements by writing the key in the brackets

```
var occupations = ["Malcolm": "Captain", "Kaylee": "Mechanic"]
occupations["Jayne"] = "Public Relations"
```

Dictionaries

- To create an empty dictionary, use the initializer syntax

```
let emptyDictionary = [String: Float]()
```

- If type information can be inferred, write an empty dictionary as `[:]`—for example, when you set a new value for a variable or pass an argument to a function

```
occupations = [:]
```

- Alternatively when **declaring** an empty dictionary

```
var newEmptyDictionary: [String: String] = [:]
```

Optionals

- An optional value either contains a value or contains `nil` to indicate that a value is missing
- Write a question mark (`?`) after the type of a value to mark the value as optional

```
var optionalName: String? // value is nil
optionalName = "John Appleseed" // value is now "John Appleseed"
```

Handling Optional Values

- Rudimentary way to check for `nil`

```
if optionalName == nil {
    // do something
}

if optionalName != nil {
    // do something
}
```

- Best practices propose a **better** method: Using `if` and `let` together to work with values that might be missing

- If the optional value is `nil`, the conditional is `false` and the code in braces is skipped.

- Otherwise, the optional value is unwrapped and assigned to the constant after `let`, which makes the unwrapped value available inside the block of code

```
var optionalName: String? // value is nil
var greeting = "Hello!"
if let name = optionalName {
    // this code only executes if optionalName is NOT nil
    // the type of name in this block is String, not String?
    greeting = "Hello, \(name)"
    print(greeting)
}
// the unwrapped name is out of scope here
```

Handling Optional Values

- Another way to handle optional values is to provide a default value using the **nil coalescing operator** (`??`)
- If the optional value is missing, the default value is used instead

```swift
let nickName: String? = nil
let fullName = nickName ?? "John Appleseed"
let informalGreeting = "Hi \(fullName)" // the value will be "Hi John Appleseed"
```

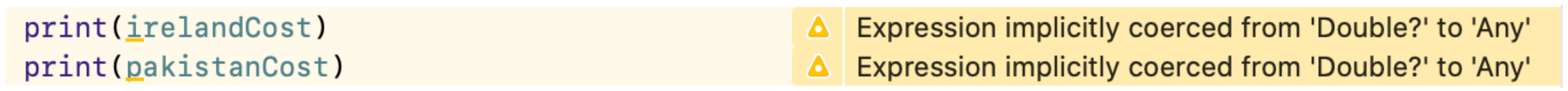
Dictionaries and Optionals

- A `Dictionary` will return an `Optional` for any given key

```
var airfare = [String:Double]()
airfare["Ireland"] = 1200.50

let irelandCost = airfare["Ireland"] // If there is no value for "Ireland", nil will be returned
let pakistanCost = airfare["Pakistan"]
```

- The following lines generate a warning because the values they are attempting to print are optional



```
print(irelandCost)        Expression implicitly coerced from 'Double?' to 'Any'
print(pakistanCost)       Expression implicitly coerced from 'Double?' to 'Any'
```

Points to Remember

- Types can be declared explicitly or implicitly inferred by the compiler
- **Prefer inferred typing** as it improves readability
- The value of a constant doesn’t need to be known at compile time, but you must assign it a value exactly once
- **Prefer immutable constants**. Change to mutable variables as needed
- Constant and variable names should be **meaningful** and **lowerCamelCased**
- **Clarity** is more important than **brevity** (no names like 'x' or ‘i')
- Values are never implicitly converted/coerced to another type
- String interpolation is kind of an exception, where values with types that conform to a special protocol get converted to Strings

Points to Remember

- Arrays are **homogeneous**: each element must be of the same type
- Arrays use zero-based indexing
- `Dictionary` is a **key-value** data structure
- Prefer `if let` unwrapping of optionals instead of nil-checking
- **Nil coalescing operator** `??` allows setting a default value
- You can declare variables to use an optional type with `?`
- You can force unwrap an Optional type with `!` but it is typically considered bad practice in production applications
- Unlike `Array` (which is an ordered collection), iterating through a `Dictionary` (an unordered collection) will not necessarily yield the same order each time