Chapter 5

Data Types

Integers

- supports 8-16-32-64 bit signed and unsigned integers
- `Int` uses the appropriate integer size for the platform
- `Int32.min` `Int32.max`

Floats (32-bit floating point) / Doubles (64-bit floating point)

```
var float: Float16 = 0.1
var double: Double64 = 3.14
```

Bools

```
var cond: Bool = true
var cond: Bool = false
```

Characters

- stored in the form of ***Grapheme Clusters***
  - made of two or more Unicode scalars that are combined to represent a single visible character

```
var unicode_code_points = "\u{0058}" // four-byte unicode scalar of 8-hex digits
```

Strings

- constructed through ***String Interpolation***
  - combinations of strings, variables, constants, expressions, function calls embedded in the “**\( )”** symbol
- ***Multi-Line***
  - indentation determined by $indent(line_n) - indent(closing)$

```
var first_name: String = "John"
var last_name: String = "Doe"
var full_name = "\(first_name) \(last_name)" // string interpolation

var multiline = """
  This is a multi-line string
"""

// Escape Characters/Sequences
var single_byte_unicode_scalar = "\u{nn}"
var four_byte_unicode_scalar = "\u{nnnn}"
```

Tuples

- store 2 or more elements of any data type

```
let tuple = (10, 9.5, "string")
let second = tuple.2       // accessing the 2nd indexed element in the tuple
let (first, _, _) = tuple  // unpacking (optimization)

let name = (first: "John", last: "Doe") // labels
print("\(name.first) \(name.last)")
```

Variables/Constants

- must be *assigned* a value during declaration in absence of ***Type-Annotation***
- the *Swift Compiler* uses ***Type-Inference*** in absence of ***Type-Annotation***
  - the compiler checks the value's type during initialization as assigns that type to the variable/constant
  - integer values default to ***Zeros*** and floating-point values default to ***Doubles***

Constants

- can be assigned ***once*** later if using ***Type-Annotation*** without ***Initialization***

```
var number                 // ERROR (compiler wouldn't be able to depict its data type)
var number: Int            // type-annotation (uninitialized Int variable with data type)
var number = 4             // type-inference  (initalized Int variable w/o data type)
var number: UInt16 = 10    // initialized UInt16 variable with data type
var number: Int32 = -10;   // semi-colons are optional

let tax_rate = 0.075       // type-inference (initialized Double constant w/o data type)
let pi: Float = 3.14       // type-annotation (initialized Float constant with data type)

let unknown: String        // uninitialized String constant (can be assigned later)
unknown = "known"
```

Optionals

- variables/constants that can store a value of a specified data type or ***nil (assigned by default)***
- provides a **safe/consistent *(nil-safety)*** approach to handling situations where a variable/constant may be ***nil*** or not

- forces the developers to ***unwrap*** the value regardless if its value ***(i.e. checking if you can access the memory location of the variable/constant)***

- ***only optionals can be assigned a "nil" value***
  - the problem is the **data type** of a variable/constant would be **undefined (optional or not)** to the compiler

- ***Access***

1. ***Forced Unwrapping (NOT RECOMMENDED)***

```
var index: Int? = 3
var treeArray = ["Oak", "Pine", "Yew", "Birch"]
print(treeArray[index!]) // forced unwrapping the optional index varable
```

1. ***Optional Binding***

   a. validates if the variable/constant is an *optional data type*

   b. validates if the optional is ***not nil*** by verifying if it can be assigned to a ***non-optional*** variable/constant assigned the *same data type*

   - can unwrap ***multiple optionals*** and include a ***boolean condition***
     - does not bind optionals if boolean test condition does not hold

```
if let index = index {
    print(treeArray[index])
} else {
    print("index does not contain a value")
}

var pet1: String? = "cat"
var pet2: String? = "dog"

if let firstPet = pet1, let secondPet = pet2, petCount > 1 {
    print(firstPet, secondPet)
} else {
    print("insufficient pets")
}
```

1. ***Implicitly Unwrapped (NOT RECOMMENDED)***

- the underlying value can be accessed without having to perform *forced unwrapping* or *optional binding*

```
var index: Int!
```

Type Casting

- uses the ***"as"*** keyword to let the compiler verify the expected value type

Upcasting (Guaranteed-Conversion)

- when an object of a particular class is casted to one of its superclasses

```
let myControl = myButton as UIControl // upcasting
```

Downcasting (Forced-Conversion)

- occurs when a conversion is made from one class to another where there is no guarantee that the cast can be made safely or that an invalid casting attempt will be caught by the compiler
- usually involves converting from a class to one of its subclasses

```
let myButton = myControl as! UIButton // downcasting
```

Safer Approach (Optional Bindings)

- returns an **optional value of the specified type** upon a successful conversion

```
if let myTextView = myScrollView as? UITextView {
    print("Type cast to UITextView succeeded")
} else {
    print("Type cast to UITextView failed")
}
```

Type Checking

- uses the ***“is”*** keyword to validate if a specific object is an instance of a class

```
if object is SomeClass {
    print("object is an instance of SomeClass"
}
```