Lecture-2

Constants, Variables and Data Types

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.

e.g
```
maximumNumberOfLoginAttempts = 10
welcomeMessage = “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
```

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.

Eg.

```
var message: String = "Hello World"
```

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'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.

```
• var color = “red”
• var color = “blue”
 // This is a compile-time error: languageName cannot be changed.
```

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!"
```

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!"
```

Comments

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

Single-line comments begin with two forward-slashes (`//`):

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

Multiline comments start with a (`/*`) and end with (`*/`):

```
/* 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).

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.

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
```

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`

Strings

A string is a series of characters, such as "Swift", that forms a collection. Strings in Swift are Unicode correct and locale insensitive, and are designed to be efficient.

let greeting = “Welcome!”

let name = “Rosa”

String Interpolation ★

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!”`

Modifying and Comparing Strings

Strings always have value semantics. Modifying a copy of a string leaves the original unaffected.

```
var otherGreeting = greeting
otherGreeting += " Have a nice time!"
// otherGreeting == "Welcome! Have a nice time!"

print(greeting)
// Prints "Welcome!"
```

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.

- `let phone = (“IPhone”, 14)`
- `// Phone is of type (String, Int), and equals (“IPhone”, 14)`

The (“IPhone”, 14) tuple groups together an `Int` and a `String` to give the user two separate values: model number and a human-readable phone name. It can be described as “a tuple of type `(String, Int)`”.

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

```
let mobile = (modelName: "IPhone", modelNumber: 14)
```

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

```
print("The Model number is \
(http200Status.modelNumber)")
// Prints "The Model number is 14"
print("The Model Name is \
(http200Status.modelName)")
// Prints "The Model Name is IPhone”
```