L-4
Integers, Double & Tuples

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."

Int	UInt
signed integer type (+, 0, -)	unsigned integer type (+, 0)
Unless you need to work with a specific size of integer, always use Int for integer values in your code.	Use UInt only when you specifically need an unsigned integer type
"On a 32-bit platform, Int is the same size as Int32.	"On a 32-bit platform, UInt is the same size as UInt32.
On a 64-bit platform, Int is the same size as Int64."	On a 64-bit platform, UInt is the same size as UInt64."
Default	Specified
Int can store any value between -2,147,483,648 and 2,147,483,647, and is large enough for many integer ranges	Int can store any value between 0 and 4,294,967,295.
let meaningOfLife = 42	let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8

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."*

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
```

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.

E.g. *let http404Error = (404, "Not Found")*

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

NOTE: 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"
```