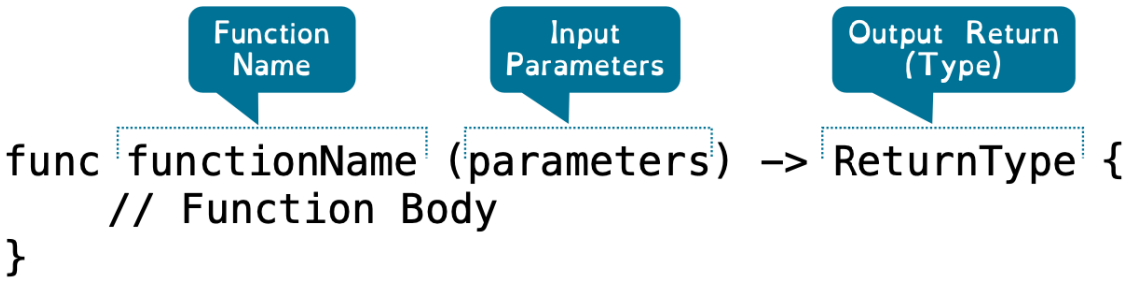
Lesson 4

PDF Slides Lesson 4

Lesson4_slides.pdf https://drive.google.com/file/d/15biOcFzJKLWoh9NJ-UbezV4MWeqJ07wi/view?usp=drivesdk

Functions

- Functions are blocks of code that can be run over and over again doing the same things
- Saves the effort of writing out code over and over again, possibly introducing transcription errors
- Functions should do one thing and do the one thing well
- Write two functions if two things are required
- Two parts to a function
  - function definition, where what the function does is specified
  - function call, where the function is executed
- Function has four components:
  - name
  - input parameters
  - output return values
  - function body
- function body does all the processing
- Example of function definition:



- Example of function call:

Function Parameters

- Parameters allow the name, type and quantity of data that will be processed by the function body be specified
- It means when the function is called, the same steps to different data can be applied
- Functions can:
  - Have no input parameters
  - one input parameters
  - multiple input parameters

No Parameters

- Sometimes functions will take input in different way, or not at all
- In these cases, it's defined as a function with no parameters

- Example of no parameter function

```
func thisPrintsStuff(){
  print("Stuff")
}

thisPrintsStuff()
```

One Parameter

- Often functions will only take a single parameter, which means they only want to operate on one thing
  - A name (or label) for the parameter that describe its purpose must be specified
  - The parameter becomes a variable that can be used within the function
  - Data type of the parameter also has to be specified
- Example of one parameter function definition

```
func thisSquaresNumber(number: Int) {
  var result = number * number
  print("Result of \(result).")
}

thisSquaresNumber(number: 4)
```

Multiple Parameters

- Most of the time functions will have multiple parameters

```
func thisMultiplies(firstNumber: Int, secondNumber: Int) {
  var result = firstNumber * secondNumber
  print("Result of \(result)")
}

thisMultiplies(firstNumber: 10, secondNumber: 5)
```

Default Value

- It's not required to specify every parameter when calling a function

- The function needs a default value so the function could still run if a vale isn't provided
- Example

```
func thisMultiplies(firstNumber: 0, secondNumber: 0) {
  var result = firstNumber * secondNumber
  print("Result of \(result)")
}

thisMultiplies(firstNumber: 10, secondNumber: 5)
```

Function Returns

- 'send' or return a vale of the caculation back to where the function was called is generally what's wanted
  - Rather than printing on the screen
- This means the value(s) calculated within the function can be used outside the function
- When calling functions that return, same names for the variables storing the returns is not needed as within the function
  - To avoid confusion, it's best practice to use different names
- any amount of parameters and returns to suit the situation can be combined
- Like parameters, there can be three situations:
  - no returns
  - one return
  - multiple returns

No Returns

- functions that has no return
- Example of no return function:

```
func thisPrintsStuff() {
  print("Stuff")
```

```
}

thisPrintsStuff()
```

One Return

- Most of the time it's desired to return a single value
- All that is required is to make the last line of the function start with return ,then specify the variable containing the value to return
- If return happens in any earlier part of the function, any line line afterwards will be ignored
- Example:

```
func thisSquaresNumber(number: Int) -> Int {
  var result = number * number
  return result
}

var value = thisSquaresNumber(number: 4)
print(value)
```

Multiple Returns

- not used often since functions generally do one thing which often implies one output
- seperate the returns by a comma is all that's required
- Example:

```
func thisDivides(number: Int) -> (Int, Int) {
  let calcOne = number / 2
  let calcTwo = number / 4
  return (calcOne, calcTwo)
}

var (div_two, div_four) = thisDivides(number: 16)
print(div_four)
print(div_two)
```

Structure

- Why Structures?
  - Often when building an app, there will be a need to store complex and related data
  - Creating a separate variable for each would get messy very quick
  - As such, structures can be used to group these together as a personalised data type:
    - A structure contains one or more variables
    - Can also add functions to it as well
- Example of a simple structure:

```
struct Person{
  var name: String
  func printHello(){
    print("Hello, \(name)!")
  }
}
var aPerson = Person(name: "Tim")
print(aPerson.name)
aPerson.printHello()
```

Advanced Structure

Initializers

- An initializer creates an instance of a structure
  - This involves creatinag an instance of each property
  - Can specify default values for properties
- Initializing is important
  - Definind a structure simply indicates what it does
  - Initializing creates an instance of the structure
- Example:

```
struct WaterMeter {
  var litresUsed: Int = 0
}
```

Custom initializer

- In more complicated situations, it might be prepered to write custom initializer functions
- Consider storing a speed in kilmeters per hour but also wished to allow initialization with miles per hour
- Example:

```
struct CarSpeed {
  var kph: Double

  init(kph: Double){
    self.kph = kph
  }

  init(mph: Double){
    self.kph = mph * 1.6
  }
}

var firstSpeed = CarSpeed(kph: 100)
var sameSpeed = CarSpeed(mph: 60)
```

Mutating Methods

- It's possible to add methods to a structure that change the values of the properties
  - These are called mutating methods
- Example

```
struct CarSpeed {
  var kph: Int = 0

  mutating func reset(){
    kph = 0
  }
}
```

Type Properties and Methods

- Type properties and methods stay the same for all instances of a structure

- To do so, simply add the word static before a property or method.

```
struct CarSpeed {
  static var legalLimit = 110
}
```

Self

- Self simply refer to the current instance of a structure
- This allows interaction with the current instance, for example, to set or access its properties

```
return "\(self.kph) kilometres per hour"

self.kph = kph
```

Classes

- Classes differ from structures as classes have hierarchical relationships
- Classes can have parents (superclasses) or children (subclasses)

Subclasses

- Subclasses can inherit properties and methods from superclasses
- Subclasses can also add on to or change the implementatino of superclass method

Base Classes

- A class that doesn't have a parent is called a base class
- Base classes are very similar to structures.
- Example:

```
class Animal {
  var animalName: String
  var numberOfLimbs: Int
  func makeNoise() {
    print("NOISE")
```

```
  }
}
```

Subclassing

- Subclassing allows extending of an existing class by basing it on an existing class
- Can change and add to these to make them more specific and relevant
- Example:

```
class Dog:Animal {
  var breed: String
  override func makeNoise() {
    print("Woof!")
  }
}
```

Overriding Initializers

- If properties needed to be added to the subclass, override the initializers are required as it will only initialize the superclasses' properties
- Example

```
class Dog:Animal {
  var breed: String
  init(animalName: String, numberOfLimbs: Int, breed: String){
    self.breed = breed
    super.init(animalName: animalName, numberOfLimbs: numberOfLimbs)
  }
}
```

Exercise

If you would like to practice some more coding, try the following extension exercises.

All exercises are to be done using a Playground.

You will find that the extension exercises below **build** on the demonstrations provided earlier in this Lesson, so you may like to go back and watch (or create) the demos for yourself before you begin.

You're also welcome to download the playground file instead of creating it yourself.

Exercise One: Multiple Returns

1. Define a function called 'calcSumDiff' that for two input parameters 'firstNum' and 'secondNum' will output the sum and the difference of the two numbers.
2. Call your function and store the results in appropriately named variables.
3. Then, print the values of these to confirm the function works as intended.

Exercise Two: Detailed Person

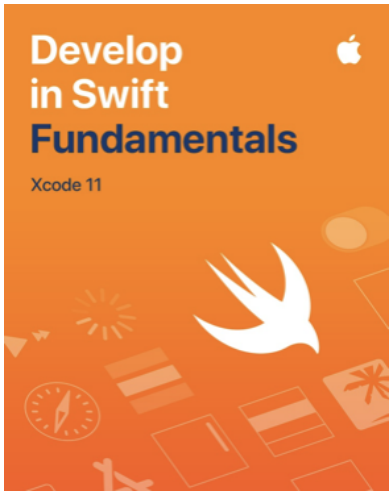
1. Modify the Person structure such that we also store their favourite food and their height. Choose appropriate data types for these properties.
2. Add a second function to the Person structure named 'foodAndHeight' to output these new properties in a human-readable manner.
3. Initialise the Structure you've created, and call your new function to confirm the changes work as intended.

Exercise Three: Another Animal

1. Create a new Subclass of Animal for a Snake. It should store a true or false value regarding whether it is poisonous, a value of its length as well as ensuring its makeNoise function outputs a 'Hiss'.
2. Ensure that you have define the initialiser for the Snake.
3. As snakes can be (for this purpose) considered to have zero limbs, pass the value directly into the superclass initialiser.

Extra Resources

- In Apple Books:



Source Code:

Ace5584/IOS-Dev-Notes

Contribute to Ace5584/IOS-Dev-Notes development by creating an account on GitHub.

https://github.com/Ace5584/IOS-Dev-Notes/tree/main