BLCKBIRDS

Swift Programming

For Beginners

IMPORTANT:

! We strongly recommend you to not only read this guide but to try everything yourself immediately. This helps strengthen your skills! You can use Xcode playgrounds to try out and practice the learned concepts. Read this tutorial on how to create and use playgrounds in Xcode.

CONTENT

Chapter 1

VARIABLES, CONSTANTS & DATA TYPES

- What variables and constants are
- When to use them
- Data types in Swift

What are variables? 🤷‍♀️

Every program stores information at some point in some way.

Variables exist exactly for this purpose. You can assign any data to a variable and name it appropriately. You can imagine a variable like a named drawer into which you put your data and retrieve it again if necessary. You can store all kinds of data in it. E.g. numbers, texts or a certain date. If necessary, you simply „open" the corresponding drawer and get access to the data stored in it.

Example: You want to save to your program's code how many pets you have. To do this, simply create a variable with a suitable name and assign the value 4 to it.

Syntax ✍️

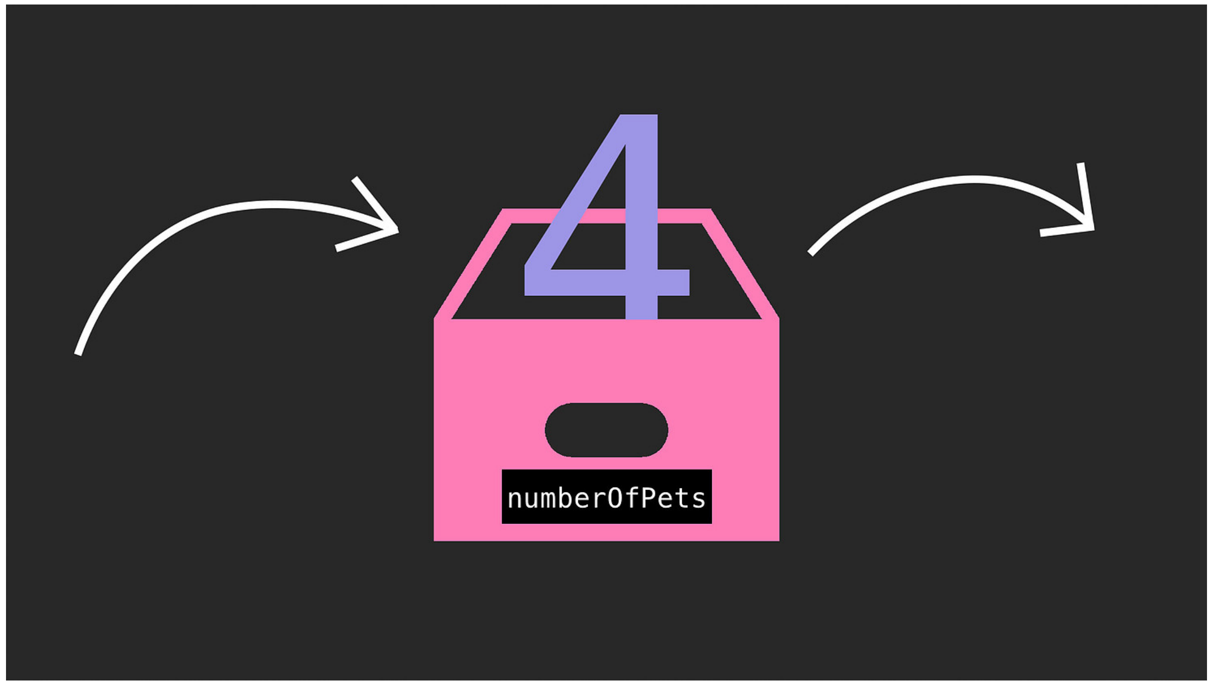
How something is written in a programming language is called syntax.

The syntax of a variable in Swift is as follows:

```swift
var numberOfPets = 4
```

The left side starts with the "var" keyword. With this keyword we tell Swift that we want to create a variable, also called declaring it. Which follows is the name we want to give the variable.

Next to the assignment operator "=" is the data we want to assign to the variable. In our example its „4“.



A variable is like a drawer where you can store data in.

Ok! But what’s a constant? 🤨

Quite simple: A constant is basically just a variable whose data cannot be changed afterwards (we also say it is not “mutable").

If, for instance, we have a variable assigned to the value 3, we can change this value at any time, for example by increasing it by 7. The value of the variable would then be 10.

If we would try to increase a constant with the value 3 by 7 instead, we would get an error.

```
var population = 1000
population = population + 300
print(population)
let degress = 30
degress = degress + 10 //causes error
```

Output:

```
1300
```

Excursus Data types 🤓

Variables and constants can only be assigned to certain types of data. Integers, floats, doubles, booleans, characters and strings. They are the basic components with which all information can be displayed. For example, an image is nothing more than an arrangement of numerical information that can be represented by integers. This numerical information is then used for the colouring of the individual pixels in the screen.

Data type	Explanation	Example
Int	Int stants for Integer and is used for whole numbers	-30,0,4,128
Float	Floats are used for representing fractional numbers.	3.14, -56.234
Double	Doubles are also used for fractional numbers, but a more precisely and more suitable for numbers with high decimal places. A double needs twice as much memory as a float.	34.99, -3.148992830101, 2.0
Boolean	Booleans (Bool) can accept either TRUE or FALSE. We use them to check whether a certain condition is met or not.	true, false
Character*	Represents a single letter of the alphabet or a single character	A, !
String*	Strings are series of characters. String data is surrounded by quotes.	„A“, „How are you?“

*It is very rare to use characters as a data type on their own. They are usually used to display strings. Also note that A is not a string, but "A" is. The string "How are you?" consists of 12 characters, spaces are also some.

Data type inference & type annotations

When you declare variables and constants, Swift usually knows which data type to use. This is called type inference. So we can sit back and relax while Swift assigns the correct data type to our variable/constant. As you can see in the syntax example above, we didn't tell Swift that the 4 should be of type Int. Since 4 is a whole number, Swift can guess the appropriate data type.

Sometimes, however, Swift may not be able to guess the data type from the context. Or we explicitly want another data type than Swift would "choose". Then we have to specify it explicitly in the declaration. This is called type annotation.

If, for instance, we want to represent the number 4 as a double, we have to write this down explicitly, since Swift infer the type Int for whole numbers. Our syntax then changes as follows:

```
var numberOfPets: Double = 4
```

As we said, when assigning whole numbers, Swift uses the data type Int. When assigning fractional numbers, Swift assumes the data type Double unless we explicitly require otherwise.

See the following example:

```
var numberOfPets = 4
print(type(of: numberOfPets))
var numberOfPetsDouble: Double = 4
print(type(of: numberOfPetsDouble))
var inchesDouble = 4.1243
print(type(of: inchesDouble))
var inchesFloat: Float = 4.1243
print(type(of: inchesFloat))
```

Output:

```
Int
Double
Double
Float
```

Chapter 2

OPERATORS, IF AND SWITCH STATEMENTS

- How to perform mathematical operations
- Logical operators
- How to use if and switch statements

Arithmetic operators +

Let’s talk about handling that data with operators. Operators are the symbols we are using to make our code work. With operators we combine, check or change the variables in our code. For example, we use the „+“ operator for the addition of the values

To perform mathematical operations we are using arithmetic operators.

These are:

"+" for addition

"-" for subtraction

"*" for multiplication

"/" for division and

"%" for getting the remainder of a division.

Let's look at some basic arithmetic operations:

```
var myApples = 3
var yourApples = 5

var ourApples = myApples+yourApples
print(ourApples)
ourApples = ourApples+10
print(ourApples)
var sumOfGrades = 32
var numberOfGrades = 8

var averageGrade = sumOfGrades/
numberOfGrades
print(averageGrade)
var firstValue = 74
var secondValue = 9

var remainder = firstValue%secondValue
//9 fits in 74 eight times. The rest of this
division is 2.
print(remainder
```

Output:

```
8
18
4
2
```

You maybe noticed that if we want to change a variables data, for instance through an addition of a certain value we have to use the variable itself in order to change it.

```
var score = 7
score = score + 3
```

Fortunately we can perform these operations with a shorter syntax, which we call **„compound assignment"**.

```
score += 3 //adds 3 to value
score -= 4 //substracts 4 from value
score *= 7 //multiplicates value with 7
score /= 2 //divides value by 2
```

Using compound assignment saves us a lot of time and leads to a more efficient workflow.

If you remember math lessons at school: Mathematical operations follow certain rules of order, which are especially important for more complex calculations: Multiplication and division before addition and subtraction. However, brackets always have priority.

```
var firstValue = 4
var secondValue = 3
var thirdValue = 7

var result =
firstValue*secondValue+thirdValue
print(result)
var resultAlt =
firstValue*(secondValue+thirdValue)
print(resultAlt)
```

Output:

```
19
40
```

Note: At this point the different data types become relevant. It is not possible to perform mathematical operations with two different data types. If, for example, we have an Int value and a Double value, these cannot be mixed.

To perform the desired mathematical operation, we must first convert the one data type accordingly.

```
var x = 3 //This is a Int
var y = 7.14 //This is a Double
var sum = x+y //Compiler Error due to
different data types
let convertedX = Double(x)

var sum = convertedX + y
print(sum)
```

Output:

```
10.14
```

Note that information can be lost through type conversion. For example, if you convert a double to an int, you will lose the fractions because the value gets rounded to the next whole number.

Logical operators 🆚

To check if a certain condition is given or to compare two values we use the logical operators.

The best known logical operators are:

"==": Both values must be equal

"!=": Both values must not be equal

">": First value must be greater than second

"<": Second value must be greater than first

">=": First value must be greater or equal to second

"<=": Second value must be greater or equal to first

"&&": Left and right statements must be true (AND)

"||": At least one of both statements must be true (OR)

Control Flow

Imagine an app that checks the user's password when logging in.

We have stored the correct password as a string in our code. We now check if the entered password is the same as the stored one. If this is the case, we grant the user access. If the password is incorrect, we deny the user access.

Such conditions are checked with conditional statements. Conditional statements are part of the control flow concept. The best known conditional statement is the if statement.

If and if-else statements

An if statement checks whether a condition is true and then executes the desired code.

If the condition is not met we can decide with the else keyword that another block of code should be executed. We can also decide, for example, that a new if statement should be executed and thus nest multiple if statements.

```
if somethingIsTrue {
   //execute block of code
} else {
   //execute another block of code
}
```

```
switch valueToConsider{
case value1:
    //execute this block of code
case value2:
    //execute this block of code
default:
    //otherwise execute this block
}
```

```
var degressCelsius = -4

if degressCelsius<0 {
    print("It's snowing")
}
```

Output:

```
Its snowing
```

```
degressCelsius = 10

if degressCelsius<0 {
    print("It's snowing")
} else {
    print("It's not snowing")
}
```

Output:

```
It’s not snowing
```

```
var isItCloudy = true

if degressCelsius<0 {
    print("It's snowing")
} else if isItCloudy == true {
    print("It's raining")
} else {
    print("It's not raining and not snowing")
}
```

Output:

```
It’s raining
```

Switch statements

If we nest a lot of if statements, our code can become quite messy and hard to read. To check many conditions, switch statements are recommended instead.

A switch statement takes a value with several options and executes code according to the various options (the „cases“). With the „default“ keyword we can also determine what happens if none of the cases occurs.

```
var country = "Sweden"

switch country {
case "Germany":
    print("This country is in Europe")
case "Sweden":
    print("This country is in Europe")
case "USA":
    print("This country is in North America")
case "India":
    print("This country is in Asia")
case "Egypt":
    print("This country is in Africa")
default:
    print("Sorry i don't know the continent
of this country")
}
```

Output:

```
This country is not in Europe.
```

Chapter 3

FOR AND WHILE LOOPS, ARRAYS AND DICTIONARIES

- What Dictionaries and Arrays are and how to use them
- Iterating with for- and while loops

Collection types 💡

In Swift we can also declare so-called collection types. These are arrays, dictionaries and sets. In this tutorial we will cover the two most important ones: arrays and dictionaries. Collection types are mutable when we assign them to a variable. Collection types assigned to a constant are not mutable, which means that we cannot change their content afterwards.

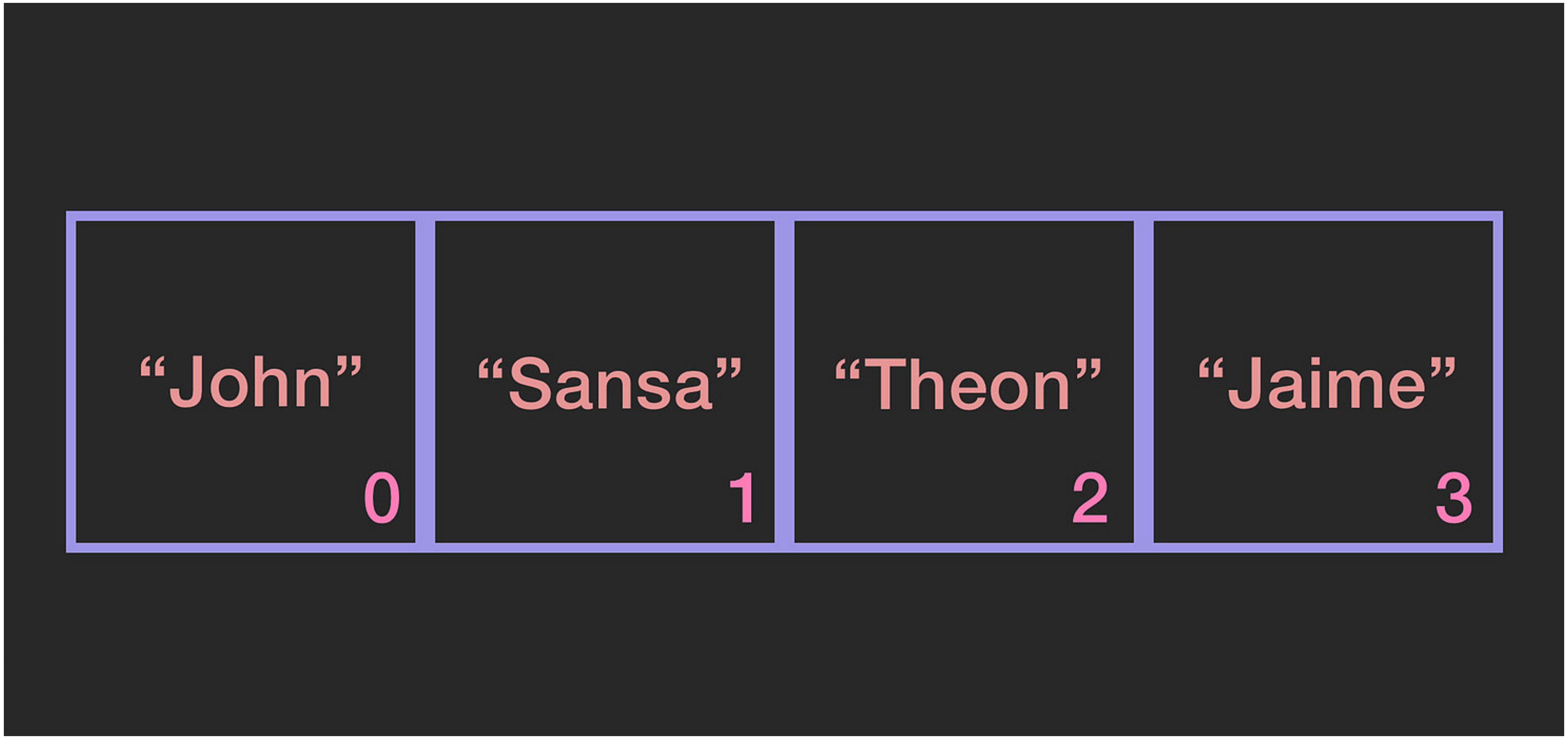
Arrays 🤷‍♀️ ❓

An array is simply an ordered collection of values of the same data type. The values in an array can occur multiple times.

```
var names = ["John", "Sansa", "Theon", "Jaime"]

var number1 = 3
var number2 = 9
var number3 = 21

var numbers = [number1, number2, number3]
```



Swift infers the data type of our values, so we don't have to make that explicit.

But if we create an empty array, Swift can't know which data types this array will contain later on. Therefore we have to specify this when declaring.

```
var emptyAray = [String]()
```

We can access the individual elements in an array and also mutate them (as long as the array is assigned to a variable).

To do this we put a square bracket behind our array's variable with the position, the index, of the element we want to access. **Note:** The index of a collection type always starts at 0, so the first element has an index of 0, the third an index of 2.

```
print(names[1]) //Output: Sansa
print(names[3]) //Output: Jaime
names[3] = "Cersei"
print(names) //Output: ["John", "Sansa",
"Theon", „Cersei"]
```

Output:

```
Sansa
Jaime
[„John“,“Sansa“, „Theon“, „Cersei“]
```

We can insert a value into an array using the insert method. We specify the index where the new element should be. With the append method we add a new element behind the last element of an array. If we want to remove a certain element we use the remove method

```
names.insert("Varys", at: 2)
print(names)
names.append("Robb")
print(names)
names.remove(at: 3)
print(names)
```

Output:

```
 ["John", "Sansa", "Varys", "Theon",
 “Cersei”]
 ["John", "Sansa", "Varys", „Theon",
"Cersei", „Robb“]
["John", "Sansa", "Varys", "Cersei", "Robb"]
```

Dictionaries 📚

The concept of dictionaries, also known as dicts, is pretty straight forward. A dict consists of different keys. Each key is assigned to a certain value. Every key is unique and must only occur once.

To get a picture of an dict, imagine a lexicon. The term we are looking for is the key. The explanation of the term is the corresponding value. So if we search for a certain explanation we look up the corresponding term

We declare a dict like this:

```
var capitalCities = ["Russia":"Moscow",
"Canada": "Ottawa", "France": "Paris"]

var populationInMillions = ["Moscow": 3.2,
"Ottawa": 1.0, "Paris": 3.2]
```

All keys in a dict must have the same data type. Also all values must have the same data type. However, keys and values must not be of the same type. Similar as arrays, we don't have to specify the data type for the values and keys as long as they are consistent. But we must also specify the data types if we want to declare an empty dict.

```
var emptyDict = [Int:Double]()
```

We can also add a new key-value pair to a dict.If we want to access a certain value we do this similar as with arrays, but we specify the corresponding key instead of the index (unlike arrays, dicts are unordered, which is the reason for elements in dicts not having a index). With the same syntax we we can also mutate the value of a key.

```
print(capitalCities["Canada"])
capitalCities["Germany"] = "Berlin"
print(capitalCities)
capitalCities["Canada"] = "Toronto" //I know
that's wrong:D
print(capitalCities)
```

Output:

```
Optional(“Ottawa”)
["Canada": "Ottawa", "France": "Paris",
"Germany": "Berlin", "Russia": “Moscow"]
["Canada": "Toronto", "France": "Paris",
"Germany": "Berlin", "Russia": "Moscow"]
```

Note: What does the optional keyword in the output of a key’s value mean? Simplified: If we access the value of a key, Swift does not know if this key or its corresponding value exists. Therefore, Swift treats this value as an optional. This means that there is „maybe“ a value. Don't worry if you don't understand this. Optionals will be covered in the last chapter. For now, just take away, that optionals are totally normal in Swift and don’t mean that there is anything wrong with your code.

For-In Loops

```
for element in sequence {
   //do something
}
```

In the last chapter we learned two components of the control flow concept in Swift: if and switch statements. Now we look at the other part of control flow: loops.

For loops in Swift give us the opportunity to iterate over a sequence of elements and perform a task for each element of that sequence.

For example, imagine you have an array with 100 names as a string in it and you want to print each name using a print statement. If you would write a single print statement for each name, your code would be extremely long and you would need a lot of time. With a for loop we can achieve the same goal in a more smart way syntax. Look at this example (I was too lazy to create an array with 100 names :D):

```
var listOfNames = ["John", "Sansa", "Theon",
"Jaime", "Robb", "Thyrion"]

print(listOfNames[0])
print(listOfNames[1])
print(listOfNames[2])
print(listOfNames[3])
print(listOfNames[4])
print(listOfNames[5])
```

Output:

```
John
Sansa
Theon
Jaime
Robb
Thyrion
```

For loops are a more smart way to iterate over a sequence of elements and perform a task for each element of that sequence.

```
for name in listOfNames {
    print(name)
}
```

Output:

```
John
Sansa
Theon
Jaime
Robb
Thyrion
```

We can also iterate over dictionaries:

```
for city in capitalCities {
    print("The capital city of \(city.key) is
\(city.value).")
}
```

Output:

```
The capital city of Canada is Toronto.
The capital city of France is Paris.
The capital city of Germany is Berlin.
The capital city of Russia is Moscow.
```

(The" \(xy)" within the print statement is called string interpolation and is used to print a element within an string, more in another tutorial)

While loops

```
while conditionIsTrue {
   //do something
}
```

In contrast to the for loop, the while loop executes the same code over and over until a certain condition is not met anymore.

For example, we can set a start value and increase the value until the start value reaches a maximum value, which looks like this:

```
var startValue = 1
let endValue = 10

while startValue <= endValue {
    print(startValue)
    startValue += 1
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

But be aware: We have to make sure that our condition gets wrong sometime, otherwise our code executes the while loop infinitely long and our program will eventually crash.

```
var startValue = 1
let endValue = 10

while startValue <= endValue {
    print(startValue)
}

//THIS WILL INFINETLY PRINT 1 UNTIL PROGRAM
CRASHES
```

Chapter 4

FUNCTIONS

- What functions are and how to use them
- The different components of a function
- How to process input data and return output with functions

What functions are 💡

Functions are chunks of code which we can use to perform certain tasks. A function contains statements that are executed when calling that function.

Functions allow us to make our code reusable. So instead of using the same code multiple times in our program, we can simply define a function with that code and call it as many times as we want. This saves us time and makes our code smarter and therefore better.

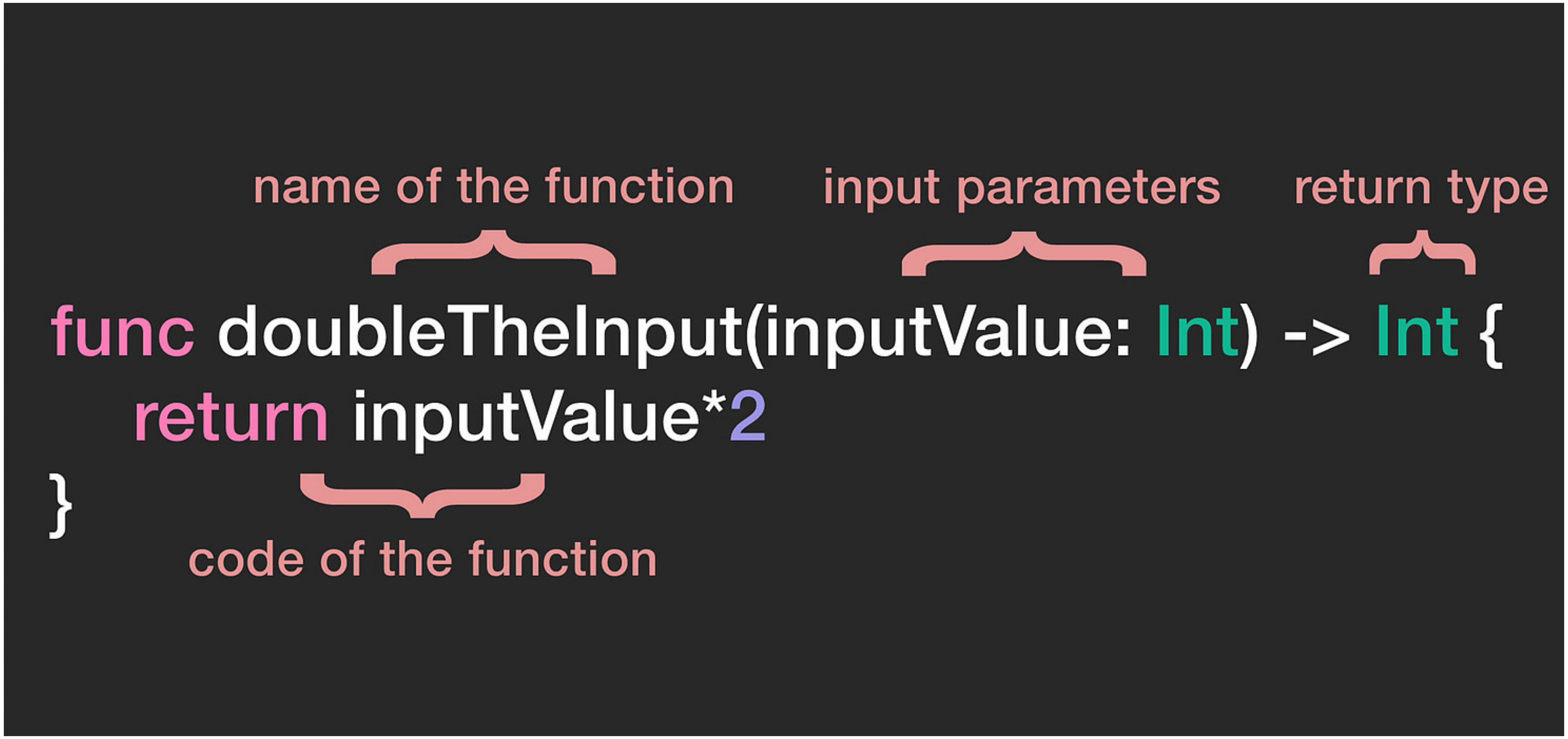
Let's start with a simple one:

```swift
func ourFirstFunction() {
    print("I love Swift!")
}
```

Output:

```
I love Swift!
```

This function outputs "I love Swift" to our console when we call it. Now let's have a look at how a function in Swift is structured



```swift
func doubleTheInput(inputValue: Int) -> Int {
    return inputValue*2
}
```

A function consists of:

- The function's name
- The code of the function
- Parameters for data input (optional)
- Return type for data output (optional)

The function's name ❝

When declaring a function we have to give it a name and thus an identity. With this name, we can call our function as desired.

Function's code

Here we define what should happen when the function is called. We can write almost everything within the function from simple print-statements to complex algorithms.

Here are a few examples to get started:

```
func simpleFunction() {
    print("This is a simple function")
}

simpleFunction()
```

Output:

```
This is a simple function
```

```
func secondFunction() {
    let friend1 = "Mike"
    let friend2 = "Lisa"

    print("My best friends are \(friend1) and 
\(friend2)")
}

secondFunction()
```

Output:

```
My best friends are Mike and Lisa
```

```
func oneLastFunction() {
    let pi = Double.pi

    print("The square root of pi is \
(pi.squareRoot())")
}

oneLastFunction()
```

Output:

```
The square root of pi is 1.7724538509055159
```

Function input with parameters ➡️

We can also add parameters to our function. Parameters allow us to feed our function with data when we call it, which the function can then process. If we to declare a function with parameters we must name the parameters in order to access them within the function's code.

Imagine for example a function which should take two values as input and add these up. For this purpose, we define a function which has two parameters "firstValue" and "secondValue". Within the function, we tell our program to add these two input values up and to print out the sum to the console.

Note: If we define a function with parameters we have to write which data type we accept as input values.

```
func sumOfTwoValues(firstValue: Int,
secondValue: Int) {
    let sum = firstValue + secondValue
    print(sum)
}

sumOfTwoValues(firstValue: 3, secondValue: 6
```

Output:

```
9
```

We can now call our function and specify any values for both parameters. The executed function display the sum of the entered values to our console.

It is possible to give a parameter within the function's code a different name than the one displayed when calling that function.

Have a look at this example:

```
func yourHome(from location: String) {
    print("I come from \(location)")
}

yourHome(from: "Canada")
```

Output:

```
I come from Canada
```

When defining the function, we specify two names for the same parameter. The first is called argument label (from) and the second is called parameter name (location). We use the parameter name within the function itself and the argument label when calling the function.

We can also omit argument labels by using an underscore "_".

If we do not specify an argument label when defining a function, the parameter name is also the default argument label.

```
func omitArgumentLabel(_ word: String) {
    print(word)
}

omitArgumentLabel("This function has no
argument label.")


func defaultArgumentLabel(number: Int){
    print(number)
}

defaultArgumentLabel(number: 3)
```

Output:

```
This function has no argument label
3
```

Function output with return values ↩️

Of course, a function can not only accept input but also return output. In order to so, we are supposed to give our function a return type. Within the function's code, we give back the desired value with the return keyword. The returned value must have the same data type as the specified return type.

We can use, for instance, functions with return values to assign them to a variable. Or we print out the return value of a function within a print statement.

```
func doubleTheInput(inputValue: Int) -> Int {
    return inputValue*2
}

let twoTimesFour = doubleTheInput(inputValue:
4)

print(twoTimesFour)
print(doubleTheInput(inputValue: 16))
```

Output:

```
8
32
```

Chapter 5

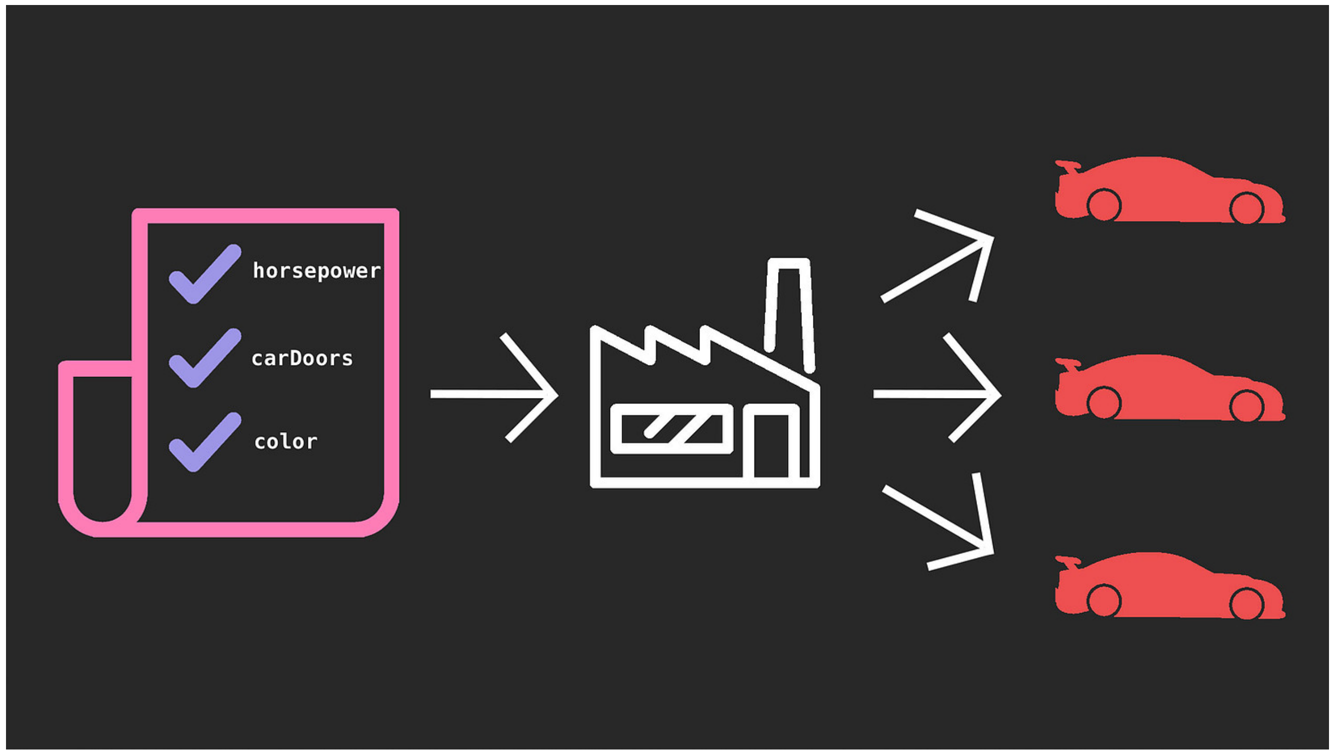
CLASSES

- Structuring your code with object-oriented programming
- Declaring classes and storing stuff in it
- Class inheritance

The concept of object-oriented programming 💡

Object-oriented programming (OOP) is an approach that's helps you structuring your code. The idea behind object-oriented programming is that you define a blueprint for something and create several instances from that blueprint.

Imagine a car manufacturer, for example. The company uses a blueprint, a design with the different properties of one of the cars, like the number of horsepowers, the color, the number of doors, etc. The car factory of the company then produces the individual cars based on this blueprint -> it creates instances!



Classes

In Swift, we can use classes to create such a blueprint. Classes are constructs which we can use to create the different components of our code, the individual objects.

Let's follow the car example and define a class called ***Car***.

```swift
class Car {

}
```

We can now provide this class with various attributes. For example, we can specify which properties an instance created from the class should have. We can then create an instance of this class. If our class contains constants, we cannot change them after creating the instance. Variables, on the other hand, can.

```swift
class Car {
    let horsepower = 100
    let color = "red"
    var numberOfCarDoors = 4
}

let myCar = Car()

myCar.colour = "blue" //Compiler Error
because color is a constant
myCar.numberOfCarDoors = 2
print(myCar.numberOfCarDoors)
```

Output:

```
2
```

However, we do not have to determine in advance which values the properties of the instances should have. Instead, we can tell the program that the instance should have such a property, but the corresponding value is not set until the instance is created. If we want to do this we have to equip our class with an ***init*** function. With the ***init*** function we say that the values entered when creating the instance should be assigned to the individual properties.

```swift
class Car {
    let horsepower: Int
    let color: String
    let numberOfCarDoors: Int

    init(horsepower: Int, color: String,
numberOfCarDoors: Int) {
        self.horsepower = horsepower
        self.color = color
        self .numberOfCarDoors =
numberOfCarDoors
    }
}

let myCar = Car(horsepower: 120, color:
"green", numberOfCarDoors: 4)
```

Note: With the ***self***-keyword, we refer to the properties of the class. We assign the property of the class' instance (***self*** keyword) to the data which we enter when creating the instance.

So we've already jumped to the next thing which we can equip classes with: Functions. Functions of a class are the skillset which we provide to the instances. Functions of a class are called methods. For example, we can give our class the method "motorOn" and "motorOff". The created instances are then able to execute exactly these functions!

```
class Car {
    let horsepower: Int
    let color: String
    let numberOfCarDoors: Int

    init(horsepower: Int, color: String,
numberOfCarDoors: Int) {
        self.horsepower = horsepower
        self.color = color
        self .numberOfCarDoors =
numberOfCarDoors
    }

    func engineOn() {
        print("Engined turned on.")
    }

    func engineOff() {
        print("Engine turned off.")
    }
}

let myCar = Car(horsepower: 120, color:
"green", numberOfCarDoors: 4)

myCar.engineOn()
myCar.engineOff()
```

Output:

```
Engine turned on.
Engine turned off.
```

Class Inheritance 👩‍👧

A class can inherit the properties and methods from another class. This means that if we have a class (in this case known as subclass) that inherits from another class (known as superclass), an instance of that class can also access the methods of its superclass.

For example, imagine a class ***Person*** with the attributes ***firstName*** and ***lastName*** and the method ***greetings***.

```
class Person {

    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String)
{
        self.firstName = firstName
        self.lastName = lastName
    }

    func greetings() {
        print("Hi, my name is \(firstName) \
(lastName)!")
    }
}
```

Now we have a second class *Employee*. This class inherits from the ***Person*** class and has ***stuffNumber*** as its own property. It also has a function "identify" which accesses the first and last name property of the superclass. In this case, ***Person*** is the superclass and ***Employee*** the subclass.

```
class Employee: Person {

    var staffNummer: Int

    init(staffNummer: Int, firstName: String,
lastName: String) {

        self.staffNummer = staffNummer

        super.init(firstName: firstName,
lastName: lastName)

    }

    func identity() {
        print("First name \(firstName), Last
name \(lastName), Staff Number \
(staffNummer)")
    }
}

let christine = Employee(staffNummer: 780,
firstName: "Christine", lastName: "Mayer")

christine.greetings()
christine.identity()
```

Output:

```
Hi, my name is Christine Mayer!
First name Christine, Last name Mayer, Staff
number 780
```

Note: The ***init*** function of our subclass must call the ***init*** function of the superclass and provide it with the necessary information, because when creating an instance of a subclass we are implicitly are also creating an instance of the superclass.

Class overriding ✏️

It is possible to override methods of the superclass. Properties can also be overwritten. The latter we do within the ***init*** function of the subclass after we have called the ***init*** function of the superclass.

```
class Person {

    var firstName: String
    var lastName: String
    var zipCode = 12345

    init(firstName: String, lastName: String)
{
        self.firstName = firstName
        self.lastName = lastName
    }
    func greetings() {
        print("Hi, my name is \(firstName) \
(lastName)!")
    }
}

class Employee: Person {
    var staffNummer: Int

    init(staffNummer: Int, firstName: String,
lastName: String) {

        self.staffNummer = staffNummer

        super.init(firstName: firstName,
lastName: lastName)

        zipCode = 12345 //We override to
inherited property of the superclass

    }

    func identity() {
        print("First name: \(firstName), Last
name \(lastName), Staff Number \
(staffNummer)")
    }

    override func greetings() {
        print("Hello, how are you?") //We
override the function of the superclass
    }
}

let christine = Employee(staffNummer: 780,
firstName: "Christine", lastName: „Mayer")

christine.greetings()
```

Output:

```
Hello, how are you?
```

If we want to prevent methods and properties from being overwritten, we use the keywords ***final var*** and ***final func*** within the superclass definition.

Chapter 6

OPTIONALS

- What optionals are and how to use them
- How to safely unwrap optionals
- Understanding „?“ and „!“ in Swift code

What are optionals? 🎁

In Swift variables and constants can't be nil. This means we cant assign nothing to them.

```
var emptyVar: Int = nil
//this will always fail
```

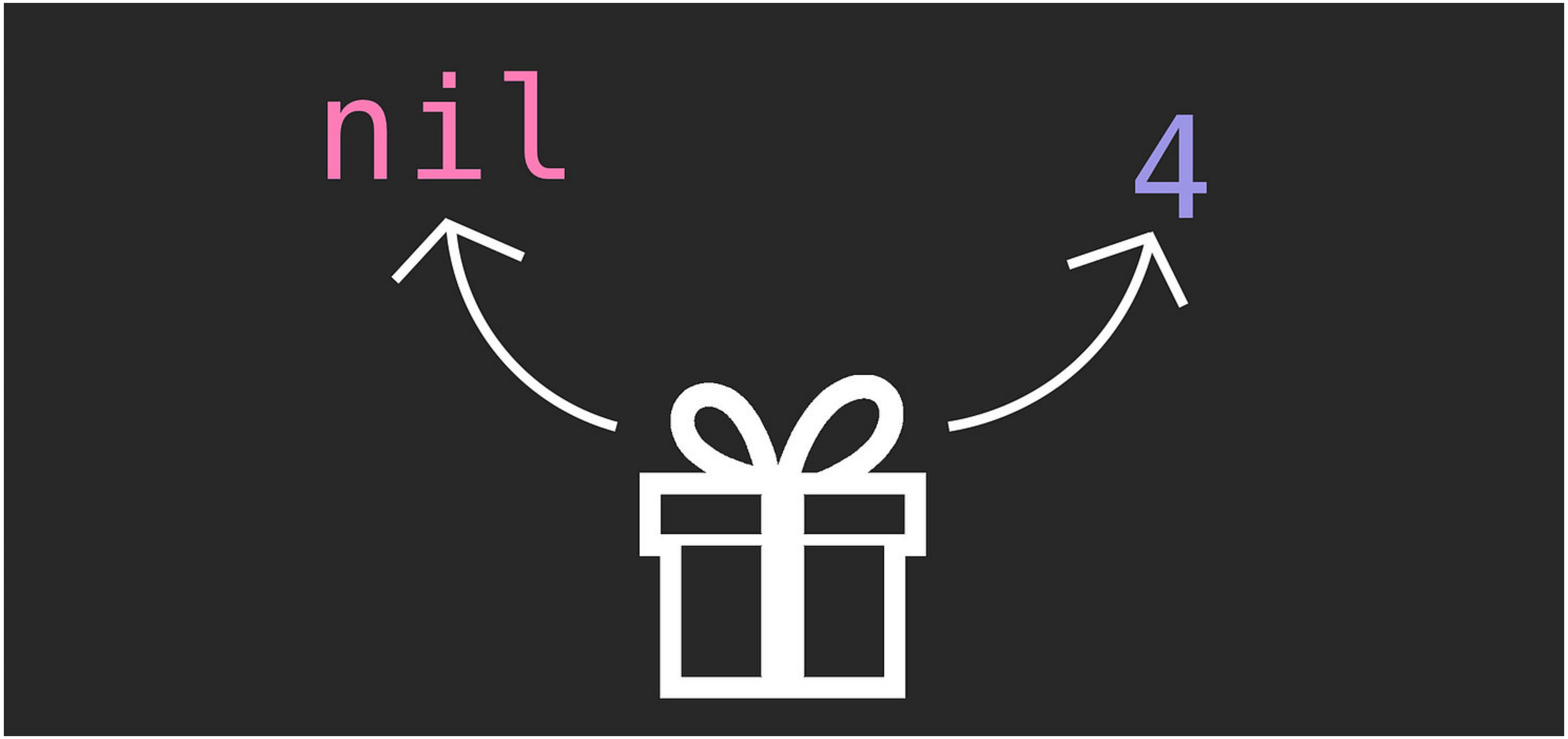
What we can do instead is to declare a variable or constant as an optional. Imagine an optional as a container for your variable. Optionals can contain two things: A value or nothing. Therefore the name: An optional may contain a value or may simply be nil.

So instead of using a variable that could become nil and crash our program, we use optionals.

To create an optional we have to declare a variable with type annotation but instead of assigning a value to it we write a "?" after the data type. We can now assign a value to the option or assign nil because optionals can represent both.

```
var ourFirstOptional: Int?

ourFirstOptional = nil
print(ourFirstOptional)
ourFirstOptional = 4
print(ourFirstOptional)
```



Output:

```
nil
Optional(4)
```

There are several ways to access the value inside an optional (so to "grab the content" out of the container).

Force unwrapping 🔍

When we want to force unwrap an optional, we tell our program that we know for sure that the optional contains a value and is not nil.

We force unwrap an optional with a „!".

```
var insideValue = ourFirstOptional!
print(insideValue)
```

Output:

```
4
```

But beware: If we do this when our optional is nil, our program will crash. So we should only use force unwrap if we are absolutely sure that the optional is not nil.

```
ourFirstOptional = nil

var insideValue = ourFirstOptional!
//This will crash our program because
optional contains nil
```

To make sure that our optional is not nil before we unwrap it, we can use a technique called optional binding.

Optional binding 🤓

We can unwrap an optional in a safe way with an if-let statement. So we write an if-statement which unwraps the optional only after checking that the optional contains a value. We check this condition by trying to assign the optional to a constant. If the optional is nil our code will return false, because, as we have learned, a constant can't become nil. However, if the optional contains a value we can assign it to the constant and the condition of our if statement is met. We can then use the declared constant within our if statements and output it for instance with a print statement.

```
if let insideValue = ourFirstOptional {
    print(insideValue)
} else {
    print("Optional contains nil")
}
```

Output:

```
4
```

Note: The declared constant is only available within the if statement and can't be used outside.

It is also possible to have multiple if-let statements at once:

```
var firstOptional: Int?
var secondOptional: String?

firstOptional = 7
secondOptional = "Hello!"

if let firstValue = firstOptional, let
secondValue = secondOptional {
    print(firstValue)
    print(secondValue)
}
```

Output:

```
7
Hello
```

Of course, we can also use a variable instead of a constant. We can then change its value within the statement.

Use the optional binding technique whenever you are not sure whether an optional contains a value or not.

Optional Chaining ⛓

There is one more thing you should know about: Optional chaining. Optional chaining is used in Swift when querying or accessing properties, methods, etc. of an optional.

To understand this, we create a class ***Person***.

```
class Person {

    let firstName: String
    let lastName: String

    init(firstName: String, lastName: String)
{
        self.firstName = firstName
        self.lastName = lastName
    }

    func greetings() {
        print("Hello my Name is \(firstName)
\(lastName)!")
    }
}
```

Now we declare an optional of this class. The optional can now either contain an instance of the class ***Person*** or it contains nil.

```
var personOptional: Person?
```

To access the properties and method of the class we use optional chaining. To do this we mark the optional with a "?" before calling a property or method. If the optional contains nil the statement will also return nil, but unlike with forced unwrapping, it won't crash. If the optional contains an instance of the ***Person*** class, our code will execute the appropriate code.

```
var personOptional: Person?

personOptional?.greetings() //Nothing will
happen
personOptional = Person(firstName: "John",
lastName: "Snow")

personOptional?.greetings()
```

Output:

```
Hello my name is John Snow!
```

If our statement returns a value, this value itself is always an optional too. To access it we also have to unwrap it.

```
print(personOptional?.firstName)
```

Output:

```
 Optional(„John“)
```

```
print((personOptional?.firstName)!)

if let firstName = personOptional?.firstName {
    print(firstName)
}
```

Output:

```
 John
```

```
if let firstName = personOptional?.firstName {
    print(firstName)
}
```

Output:

```
 John
```

CONCLUSION

Thank you for reading this guide! You should now have enough knowledge about the Swift programming language to get started with iOS development!

Please visit www.blckbirds.com for more great content on iOS development.

If you have any feedback, criticism or suggestions, feel free to write us a message!

Also make sure you follow us on Instagram and Facebook to not miss any updates.