We ❤ Swift

Swift Programming from Scratch

in 100+ exercises

made with ❤ by Andrei Puni and Silviu Pop

Table of Contents

1. First Steps

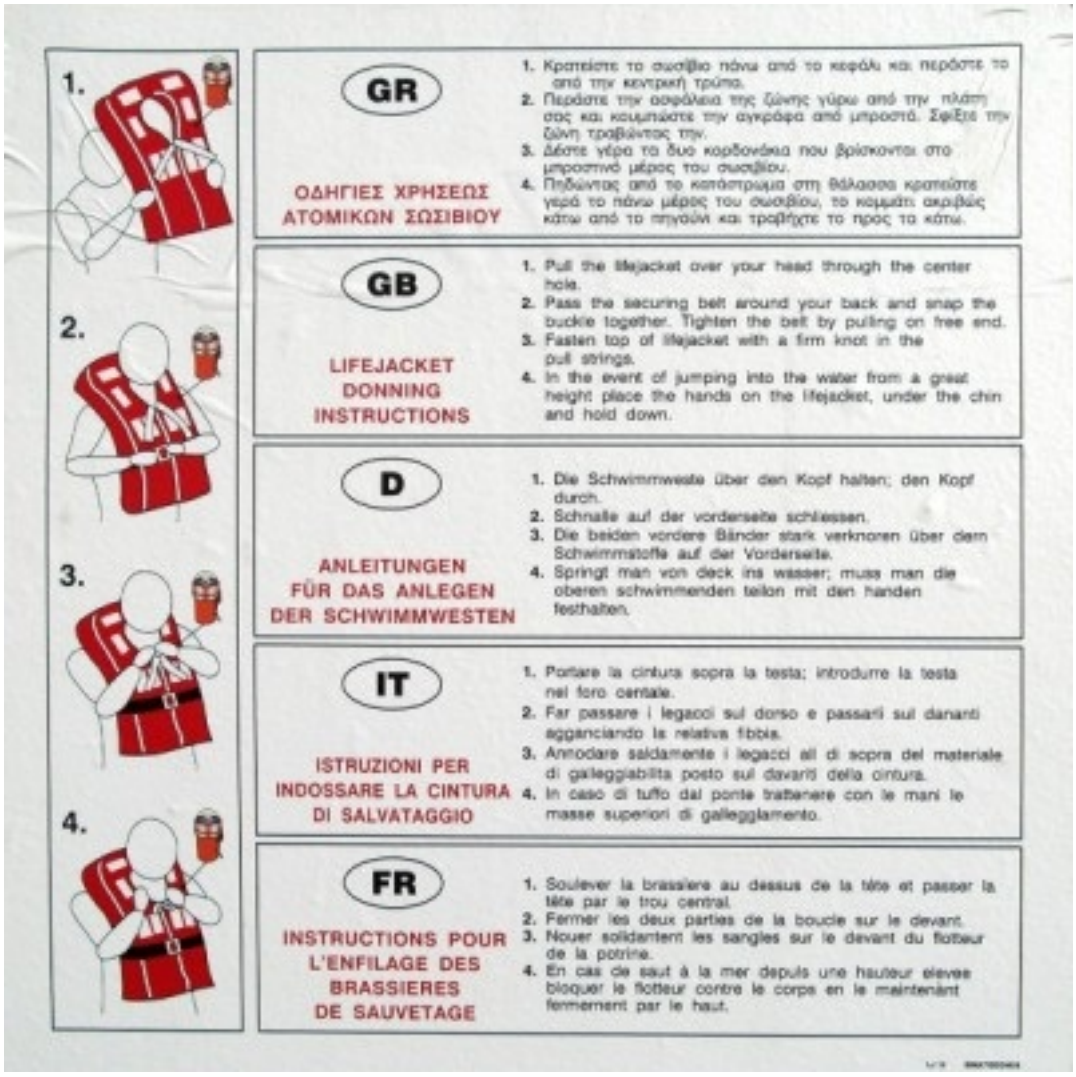
What is a computer program?

A program is a list of instructions that are followed one after the other by a computer. You are most likely familliar with lists of instructions in everyday life. An example of a list of instructions would be a cooking recipe:

Fried eggs:

1. heat 2 tablespoons of butter in a non-stick frying pan.
2. break the eggs and slip into pan one at a time.
3. cook until whites are completely set and yolks begin to thicken.
4. carefully flip eggs. cook second side to desired doneness.
5. sprinkle with salt and pepper. serve immediately.

Another example is this list of instructions on how to put on a life jacket:



The lists of instructions mentioned above are made to be executed by people. Computer programs are simillarly just lists of instructions but they are meant to be executed by computers. They are meant to be readable and understandable by humans but executing them would often be highly impracticable.

For example the program used for drawing a single screen in a modern game executes hundred of millions of mathematical operations like additions and multiplications. Executing such a list of instructions would take any person an embarasing amount of time, yet computers can happily do it 60 times per second.

Why do we need a programming language?

Lists of instructions like cooking recipes and putting on a life jacket are quite easy to understand for humans but they're incrediblly difficult to understand for a computer. Programming languages are designed as a way of giving a computer instructions that it can easily understand. That is because a programming language (like Swift) is much less ambigous than a language like english. Also it closely resembles the way in which a computer works.

In this book you'll learn the basics of programming using Swift. More importantly this will teach you about the kind of instructions that your computer understands and building programs for it to execute.

Whether you want to build an app, a game or a website the basic principles remain the same. You have to write a program for the computer to execute and writing such a program is done using a programming language.

Why Swift?

The Swift programming language was introduced in June 2014 by Apple, since then it has grown immensly in popularity. Swift is primarly used for developing apps and games for the iPhone and the Mac and provides an easier and more enjoyable way of doing that.

The great news is that Swift is also a great programming language for learning to code because of the **Playgrounds** feature described below.

Using Playgrounds

Playgrounds provide a fun and interactive way of writing code. Traditionally you would write a program and run it to see its results. With playgrounds you can see the results of your program imedially as you type it. This gives you a lot of opportunity for experimenting and makes learning faster.

If you have the companion app for this book than clicking on an exercise will open a playground for you to start coding.

If you don't have the companion app installed than you can open Xcode and create a new playground by clicking the "Get started with a playground" button. Select OS X As your Platform and choose a destination where you want to save the Playground.

Note: If you don't have Xcode installed, download the latest version from here

We'll start looking at basic concepts one by one now. We encourage you to experiment with the code we introduce by typing the statements into a playground and changing values around.

Variables and Constants

Use the `var` keyword to declare a variable and the `let` keyword to declare a constant. Variables and constants are named values. Variable can change their value over time and constants don't. To change the value of a variable you need to asign it a new one.

```
// declares a variable named a that has the value 1
var a = 1
// assigns the value 2 to the variable a
a = 2
// a has the value 2
```

```
// declares a constant named one with the value 1
let one = 1
one = 2 // this gives an error because we cannot change the value of a constant
```

> the text after `//` is called a comment. Comments are ignored by the computer when executing the program. They are usually used to explain parts of code

Naming Variables

Variables should usually be named using alphabetical characters. For example: `sum`, `number`, `grade`, `money`

If you want your variable's name to contain multiple words then you should start each word in the name with an uppercase letter except for the first one. For example you want a variable that holds the number of students in a class than you should name it `numberOfStudents` instead of `numberofstudents` because the first one is more readable.
This naming convention is called CamelCase.

It's recommanded to use descriptive names for variables. But don't overdo it, for example `numberOfStudents` is a reasonable name while `numberOfStudentsInTheCurrentClass` is too long. A good rule of thumb is to use at most 3 words for the name of a variable.

We could have used a way shorter name for the variable above, for example we could have called it `n`. The disadvantage with short variable names is that they're not expressive. If you read your code after 2 months you most likely won't remember what `n` means. But `numberOfStudents` is immediately obvious.

Generally its not a good idea to have variables that consist of a single letters but there are some exceptions.
When dealing with numbers that don't represent something it's ok to use single letter names.

Basic Operators

You can write arithmetic expressions using numbers, variables, operators and parentheses.

```
// The + operator returns the sum of two numbers
let sum = 1 + 2 // 3

// The - operator returns the difference of two numbers
let diff = 5 - sum // 5 - 3 = 2

// The * operator returns the product of two numbers
let mul = sum * diff // 3 * 2 = 6

// The / operator returns the numbers of times the  divisor(the number on
// the right side) divides into the dividend(the number on the left side)
// For example, when dividing 6 by 3, the quotient is 2, while 6 is called
// the dividend, and 3 the divisor.
// 13 divided by 5 would be 2 while the remainder would be 3.
let div = mul / diff // 6 / 2 = 3
```

```
// The remainder(modulo) operator returns the remainder of the division
let mod = 7 % 3 // 1 because 7/3 = 2 and remainder 1 (2 * 3 + 1 = 7)

// You can use parentheses to group operations
(1 + 1) * (5 - 2)

// Multiplication, division and remainder have higher precedence than
// addition and subtraction.
// For example: 5 + 2 * 3 = 5 + 6 = 11
```

Integer Division

Addition, subtraction and multiplication behave pretty much as you expect. The tricky operations are division and remainder.
Take for example `5 / 2`. Normally you'd expect the result to be `2.5`. In Swift dividing two integers also produces an integer this is acomplished by discarding the part of the number after the decimal point. So `5 / 2 = 2`.
The remainder operator or modulo operator (%) is used to get the remainder of an integer division. `5 % 2 = 1`
For `5 / 2`:
`quotient = 5 / 2 = 2`
`remainder = 5 % 2 = 1`
`quotient * 2 + remainder = 5`

Generally speaking for two integers `a` and `b` this equations always hold
`quotient = a / b`
`remainder = a % b`
`b * quotient + remainder = a`

NOTICE: `remainder = a - b * quotient`
This implies that `remainder = a - b * (a / b)` and
`a % b = a - b * (a / b)`
You can view `a % b` as a shorthand way of computing `a - b * (a / b)`

NOTICE: if `a % b = 0` then `b` divides `a`, that is `a` is a multiple of `b`.
Example:
`15 / 5 = 3`
`15 % 5 = 0` ->
`15 = 5 * 3`

Order of statemets and more Playgrounds

The order of statements in a program matters. Like lists of instructions programs are executed from top to bottom.

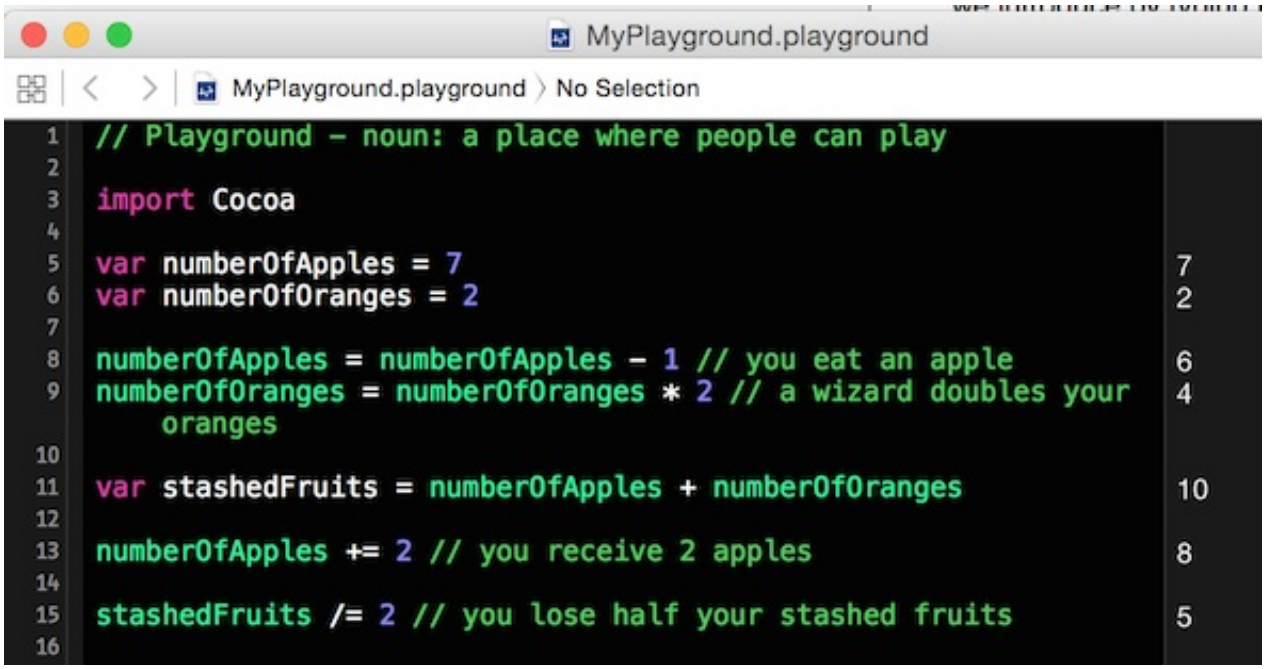
```
var numberOfApples = 7 // you have 7 apples
var numberOfOranges = 2 // you have 2 orages

// you eat an apple (numberOfApples = 6)
```

```
numberOfApples = numberOfApples - 1

// a wizard doubles your oranges (numberOfOranges = 4)
numberOfOranges = numberOfOranges * 2

var stashedFruits = numberOfApples + numberOfOranges // 10 (6 + 4)

// you receive 2 apples (numberOfApples = 8). stashedFruits remains unchanged!
numberOfApples += 2

stashedFruits /= 2 // you lose half your stashed fruits 5 (10 / 2)
```

In the program above the variable `stashedFruits` gets a value only after all the previous statements are run. Also notice that the value assigned to a variable is computed at the time of assignment. Changing `numberOfApples` after declaring `stashedFruits` will not have any effect on the value of `stashedFruits`.

Looking at the code above in a playground will give you an idea of why playgrounds are incredibly helpful for visualizing how code behaves.

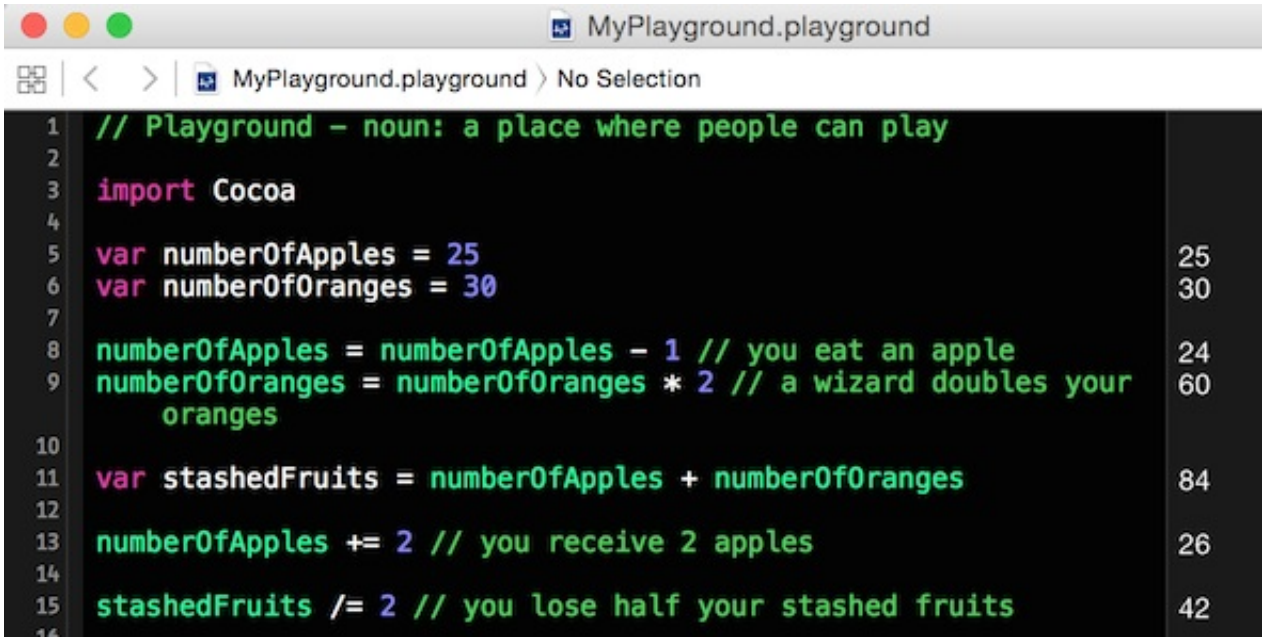
As you can see each line of code in a playground has the value of the expression on that line printed in the right area of the screen.



Now, the cool thing is that if we modify any values. All lines are imediatelly updated to reflect our changes. For example we modify:

```
var numberOfApples = 25
var numberOfOranges = 30
```

and everything is recalculated and displayed immediately.



Try playing around!

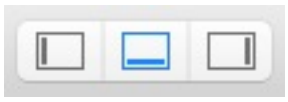
Print Statement

After making some computations you will want to show your results somehow. The simplest way to do it is with `print()` statement.

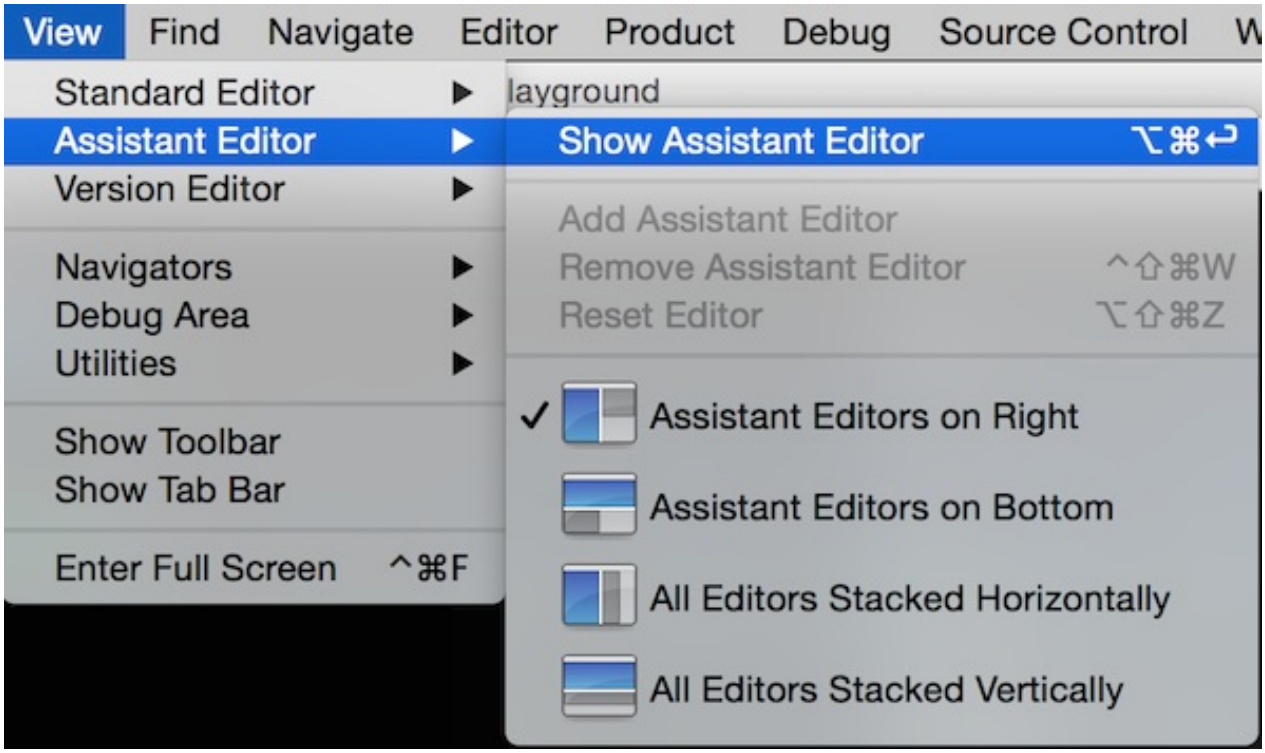
```
// will print Hello Swift! in the console
// you can print any text between quotes
print("Hello Swift!")

print(1 + 2) // will print 3

var ten = 10

print(ten) // will print 10
```

To see the console output in Playground make sure to show the `Debug Area`.

You can do that by pressing the middle button from the top right corner of the Playground.



Or from the menu:



This is how the code from the example would look with the `Debug Area` visible:

Exercises

1.1 Sum

You are given two variables `a` and `b`, compute their sum and store it in another variable named `sum` then print the result.

Code

```
var a = 1
var b = 2

// your code here
```

Example 1

Input:

```
var a = 1
var b = 2
```

Expected values:

```
sum = 3
```

Output:

```
3
```

Example 2

Input:

```
var a = 13
var b = 22
```

Expected values:

```
sum = 35
```

Output:

```
35
```

1.2 Seconds

Determine the number of seconds in a year and store the number in a variable named `secondsInAYear`.

Code

```
// your code here
```

Hint

The number of seconds in a year is 365 times the number of seconds in a day.
The number of seconds in a day is 24 times the number of seconds in a hour.
The number of seconds in a hour is 60 times the number of seconds in a minute, which is 60.

1.3 Pixels

Your are given the `width` and `height` of a screen in pixels. Calculate the total number of pixels on the screen and store the result in a variable named `numberOfPixels`.

Code

```
var width = 1920
var height = 1080

// your code here
```

Example 1

Input:

```
var width = 4
var height = 3
```

Expected values:

```
numberOfPixels = 12
```

Example 2

Input:

```
var width = 1920
var height = 1080
```

Expected values:

```
numberOfPixels = 2073600
```

Hint

Consider a 5x3 screen like this:

```
*****
*****
*****
```

The number of pixels on this screen is 5 + 5 + 5 = 5 * 3

1.4 Sum and Difference

You are given the `sum` and the `difference` of two numbers. Find out the values of the original numbers and store them in variables `a` and `b`.

Code

```
let sum = 16 // a + b
let diff = 4 // a - b

// your code here
```

Example 1

Input:

```
var sum = 16
var dif = 4
```

Expected values:

```
a = 10
b = 6
```

Example 2

Input:

```
var sum = 2
var dif = 0
```

Expected values:

```
a = 1
b = 1
```

Example 3

Input:

```
var sum = 4
var dif = 2
```

Expected values:

```
a = 3
b = 1
```

Hint 1

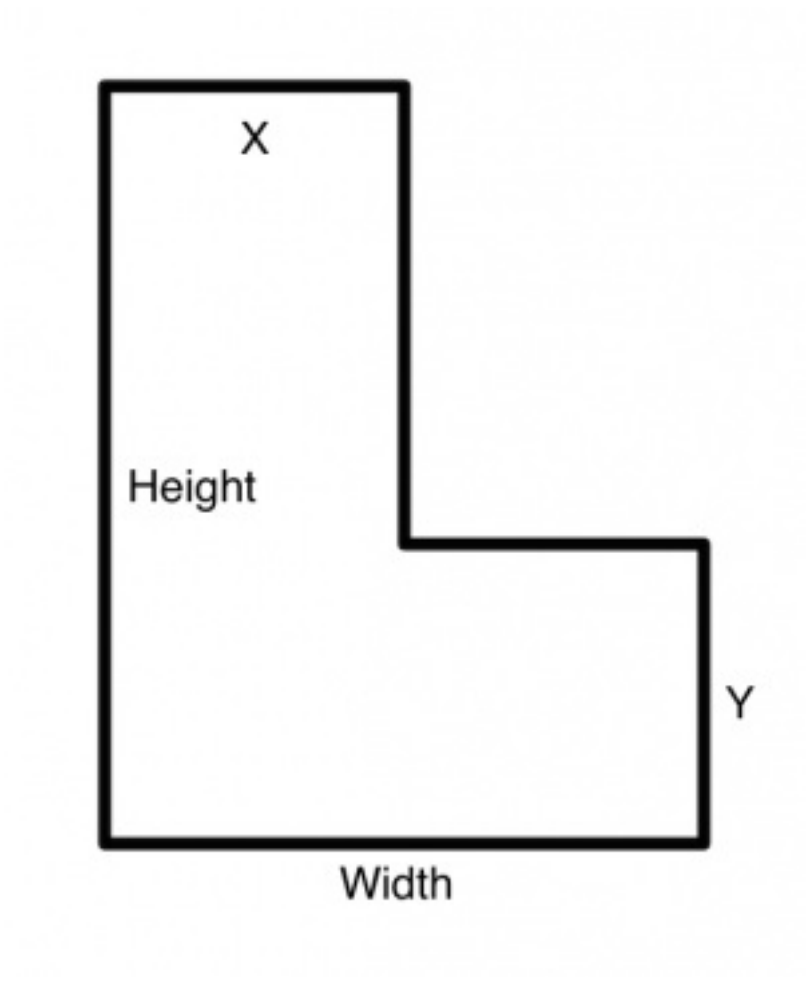
`sum + diff = a + a + b - b`
`sum + diff = 2 * a`

Hint 2

```
sum = a + b
b = sum - a
```

1.5 L Area

You are given four variables `width`, `height`, `x`, `y` that describe the dimensions of a L-shape as shown in the image below. Determine the `perimeter` and `area` of the described L-shape. Store the value of the perimeter in a variable named `perimeter`, and the area in a variable named `area`.



Code

```
var width = 8
var height = 12
var x = 2
var y = 3

// your code here
```

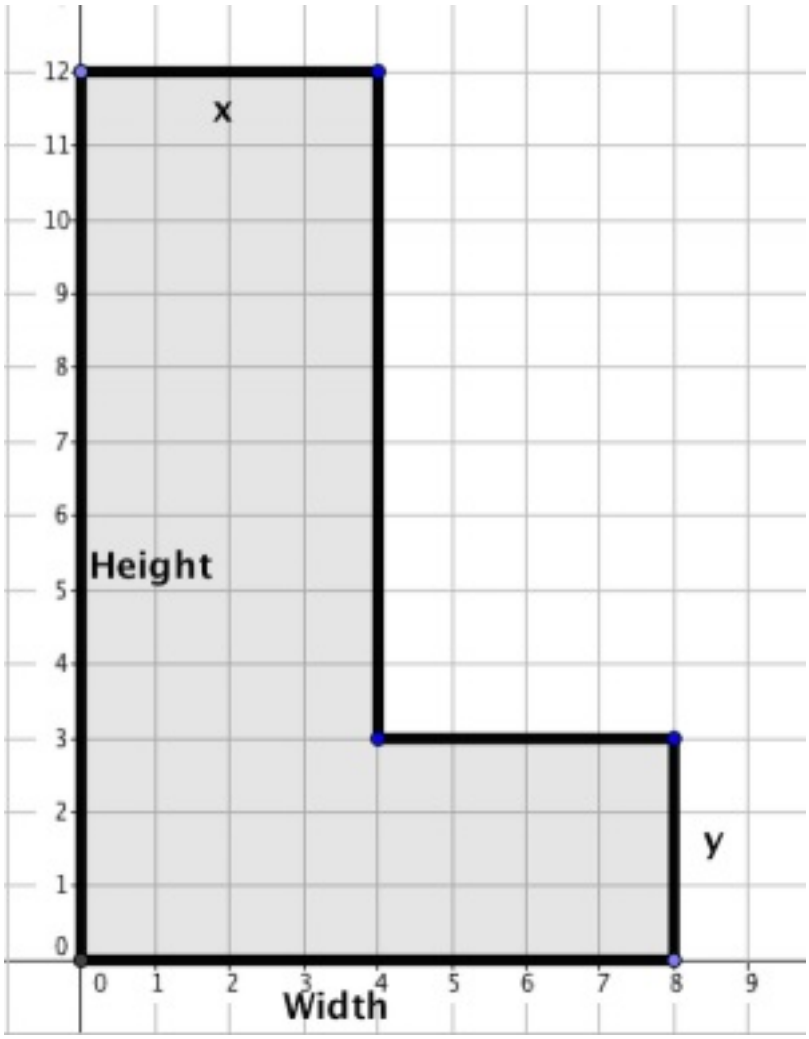
Example 1

Input:

```
var width = 8
var height = 12
var x = 4
var y = 3
```

Expected values:

```
perimeter = 40
area = 60
```



Example 2

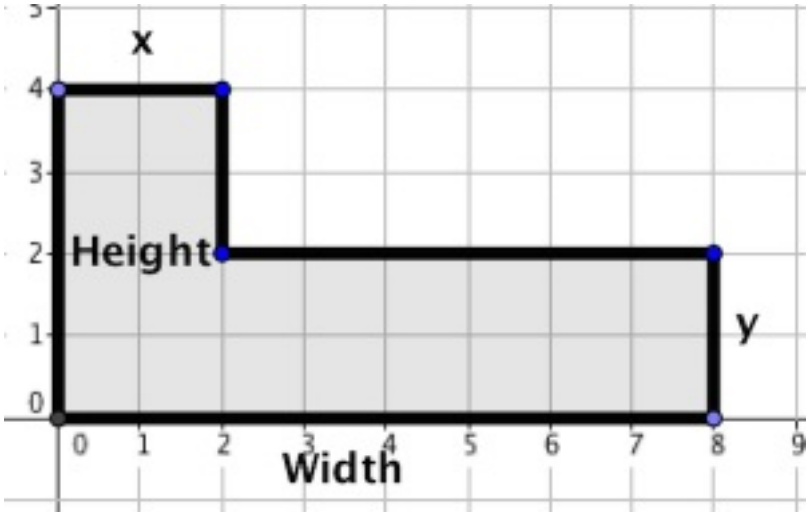
Input:

```
var width = 8
var height = 4
```

```
var x = 2
var y = 2
```

Expected values:

```
perimeter = 24
area = 20
```



Hint

The `perimeter` of the L-shape is the same as of a rectangle of size `width X height`.

To compute the `area` you can imagine the L-shape as rectangle of size `width X height` with a rectangle of size `(width-x) X (height-y)` cut out.

1.6 Swap

Given two variable `a` and `b`, swap their values. That is the new value of `a` will become the old value of `b` and vice versa.

Code

```
var a = 1
var b = 2

// your code here
```

Example 1

Input:

```
var a = 1
var b = 2
```

Expected values:

```
a = 2
b = 1
```

Example 2

Input:

```
var a = 13
var b = 7582
```

Expected values:

```
a = 7582
b = 13
```

Hint 1

Just assigning `a` to the value of `b` and `b` to the value of `a` will not work.

```
var a = 1
var b = 2

a = b // a will have the value 2. The original value of a is lost
b = a // b will remain the same
```

Hint 2

Use a third variable to save the original value of `a`.

1.7 Last digit

You are given a number `a`. Print the last digit of `a`.

Code

```
var a = 123

// your code here
```

Example 1

Input:

```
var a = 123
```

Output:

```
3
```

Example 2

Input:

```
var a = 337
```

Output:

```
7
```

Example 3

Input:

```
var a = 100
```

Output:

```
0
```

Hint

Use the remainder `%` operator.
Remember that `a = k * (a / k) + a % k`
Can you think of a value for k that gives the last digit?

1.8 Dog Years

You are given Rocky's age in dog years. Print Rocky's age in human years. You know that 1 human year is 7 dog years.

Code

```
var rockysAge = 50

// your code here
```

Example 1

Input:

```
var rockysAge = 50
```

Output:

```
7
```

Example 2

Input:

```
var rockysAge = 14
```

Output:

```
2
```

Example 3

Input:

```
var rockysAge = 15
```

Output:

```
2
```

Hint

Use division.

1.9 Brothers

Everyone hates solving word problems by hand so let's make a program to solve them for us. `x` years from now Alice will be `y` times older than her brother Bob. Bob is `12` years old. How many years does Alice have?

Code

```
var x = 3
var y = 2
var bob = 12

var alice = ?
// your code here
```

Example 1

Input:

```
var x = 3
var y = 2
var bob = 12
```

Expected values:

```
alice = 27
```

Example 2

Input:

```
var x = 1
var y = 3
var bob = 12
```

Expected values:

```
alice = 38
```

Hint

`alice + x = y * (bob + x)`
Solve for `alice`

1.10 Apples and Oranges

You have `x` apples. Bob trades `3` oranges for `5` apples. He does not accept trades with cut fruit. How many oranges can you get from Bob and how many apples will you have left?

The number of apples you will have left should be stored in a variable named `apples`. The number of oranges you will have after the trade should be stored in a variable named `oranges`.

Code

```
var x = 17

// your code here
```

Example 1

Input:

```
var x = 17
```

Expected values:

```
apples = 2
oranges = 9
```

Example 2

Input:

```
var x = 25
```

Expected values:

```
apples = 0
oranges = 15
```

Example 3

Input:

```
var x = 4
```

Expected values:

```
apples = 4
oranges = 0
```

Hint

Use the division(`/`) and the remainder(`%`) operator

1.11 Boys and Girls

A class consists of `numberOfBoys` boys and `numberOfGirls` girls.
Print the percentage of boys in the class followed by the percentage of girls in the class. The percentage should be printed rounded down to the nearest integer. For example `33.333333333333` will be printed as `33` .

Code

```
var numberOfBoys = 20
var numberOfGirls = 60

// your code here
```

Example 1

Input:

```
var numberOfBoys = 20
var numberOfGirls = 60
```

Output:

```
25 // percentage of boys
75 // percentage of girls
```

Example 2

Input:

```
var numberOfBoys = 20
var numberOfGirls = 20
```

Output:

```
50 // percentage of boys
50 // percentage of girls
```

Example 3

Input:

```
var numberOfBoys = 10
var numberOfGirls = 20
```

Output:

```
33 // percentage of boys
66 // percentage of girls
```

Hint

First you'll have to compute the total number of students in the class

Hint 1

`numberOfStudents` ... `100%`
`numberOfBoys` ... `X%`

Hint 2

`numberOfStudents / 100` = `numberOfBoys / X`

Solutions

1.1 Sum

To solve this problem we will create a variable named `sum` and initialize it with the sum of `a` and `b` (`a + b`). The next step is to print the value of `sum` using a print statement.

```
var a = 1
var b = 2

var sum = a + b

print(sum)
```

1.2 Seconds

To keep the math simple are are going to make only one multiplication on each line. We are going to compute the number of seconds in a hour, day and year starting from the number of seconds in a minute.

```
let secondsInAMinute = 60

// The number of seconds in a hour is 60 times the number
// of seconds in a minute, which is 60.
let secondsInAHour = 60 * secondsInAMinute

// The number of seconds in a day is 24x the number of seconds in a hour.
let secondsInADay = 24 * secondsInAHour

// The number of seconds in a year is 365x the number of seconds in a day.
let secondsInAYear = 365 * secondsInADay
```

1.3 Pixels

The screen can be seen as a rectangle of size width x height. The number of pixels on the screen is equal to the area of the rectangle, which is `width * height` .

```
var width = 1920
var height = 1080

var numberOfPixels = width * height
```

1.4 Sum and Difference

We notice that if we add the sum and the difference of two numbers we get the double of one of them:

```
sum + diff =
(a + b) + (a - b) =
a + a = 2*a
```

To get one of the numbers (`a`) we will half the sum of `sum` and `diff` . To get the other one (`b`) we can subtract the first one (`a`) from their `sum` .

```
let sum = 16 // a + b
let diff = 4 // a - b

// sum + diff = a + b + a - b = a + a = 2*a
// -> sum + diff = 2*a
// -> a = (sum + diff) / 2

var a = (sum + diff) / 2 // 10
var b = sum - a // 6
```

1.5 L Area

The `perimeter` of the L-shape is the same as of a rectangle of size `width X height` . Which is equal to 2 * (width + height).

To compute the `area` you can imagine the L-shape as rectangle of size `width X height` with a rectangle of size `(width-x) X (height-y)` cut out. We compute the area of the big rectangle as `width * height` and the area of the small rectangle as `(width - x) * (height - y)` . Their difference gives us the total area.

```
var width = 8
var height = 12
var x = 4
var y = 3

var perimeter = 2 * (width + height)

var bigArea = width * height
var smallArea = (width - x) * (height - y)
var area = bigArea - smallArea
```

1.6 Swap

First we'll keep a backup of the value of `a` in a new variable `temp` . We'll assign `a` to `b` which will overwrite the current value of `a` (`a` will now be equal to `b`). Next we assign to `b` the backed up value of `a` which is stored in `temp` . A useful analogy: Imagine you have 2 glasses and you want to interchange their contents. To accomplish this, you'll have to use a 3rd glass to temporarly hold the contents of one of

them.

```
var a = 1
var b = 2

var temp = a
a = b
b = temp
```

1.7 Last digit

You can get the last digit of a number by computing the reminder of division by `10`.

```
var a = 123

print(a % 10)
```

1.8 Dog Years

The problem is solved using a division by `7`. If `7` dog years = `1` human year then `X` dog years is equal to `X / 7` human years.

```
var rockysAge = 50

var rockysAgeInHumanYears = rockysAge / 7

print(rockysAgeInHumanYears) // 7
```

1.9 Brothers

We know that `x` years from now Alice will be `y` times older then her brother. We also know that her brother is currently 12 years old. Mathematically we can say that `alice + x = y * (bob + x)`. Solving this equation for `alice` gives us `alice = y * (bob + x) - x`. This can be straightforwardly written in code.

```
var x = 3
var y = 2
var bob = 12

// alice + x = (bob + x) * y
// alice = (bob + x) * y - x
var alice = (bob + x) * y - x
```

1.10 Apples and Oranges

Bob trades 3 oranges for 5 apples. If the number of apples we have would be divideable by 5 then we could trade all our apples for oranges. The number of oranges we would have would be `(x / 5) * 3` and the number of apples would be `0`. If the number of apples we have is not divisible by 5 then we can still get the number of oranges using `(x / 5) * 3` because integer divison ignores any remainder. The number of apples we would be left with would be exactly the reminder of our division that is `x % 5`.

```
var x = 17

var apples = x % 5
var oranges = x / 5 * 3
```

1.11 Boys and Girls

The problem can be solved using Cross Multiplication. `numberOfStudents / 100 = numberOfBoys / boyPercentage`. This gives us `boyPercentage = numberOfBoys * 100 / numberOfStudents`. Simillarly for the girl percentage we have `numberOfStudents / 100 = numberOfGirls / girlPercentage` giving us `girlPercentage = numberOfGirls * 100 / numberOfStudents`.

```
var numberOfBoys = 20
var numberOfGirls = 60

var numberOfStudents = numberOfBoys + numberOfGirls
var boyPercentage = numberOfBoys * 100 / numberOfStudents
print(boyPercentage)
var girlPercentage = numberOfGirls * 100 / numberOfStudents
print(girlPercentage)
```

2. Conditionals

Introduction

Sometimes you want to run some code only **if** some conditions are met. For example:

```
var numberOfOranges = 1
var numberOfApples = 5

if numberOfApples > numberOfOranges {
    print("You have more apples than oranges!")
}
```

You can compare numbers using these operators:

- `<` Less than
- `<=` Less than or equal
- `>` Greater than
- `>=` Greater than or equal
- `==` Equal
- `!=` Not equal

```
1 != 2 // true
1 == 2 // false
1 < 2 // true
1 > 2 // false
1 <= 2 // true
3 >= 3 // true
```

Anatomy of an if statement

An `if` statement has the following form:

```
if CONDITION {
    STATEMENT
    STATEMENT
    ...
    STATEMENT
}
```

The statements between curly braces (`{}`) will only be executed if the given condition is `true` ! The statements that follow after the curly brace ends will be executed independently of the condition.

An if statement can also have an `else` branch:

```
if CONDITION {
    STATEMENT
    STATEMENT
    ...
    STATEMENT
} else {
    STATEMENT
    STATEMENT
    ...
    STATEMENT
}
```

The statements in the `else` branch i.e. between `else {` and `}` will only be executed if the condition is false.

Consider the following code as an example:

```
var money = 20 // you have 20$
var burgerPrice = 10 // you ate a good burger

// if you have enough money pay for the burger
if money >=  burgerPrice {
    print("pay burger")
    money -= burgerPrice
} else {
// otherwise you will need to go wash dishes to pay for your meal
// hopefully this will not be the case
    print("wash dishes")
}

// if you have some money left order desert
if money > 0 {
    print("order desert")
}
```

Nesting conditions

If statements can be nested inside other if statements.

```
if CONDITION {
    STATEMENT

    if CONDITION2 {
        STATEMENT
        STATEMENT
        ...
        STATEMENT
    }

    STATEMENT
```

```
    ...
    STATEMENT
}
```

For example lets say we have two variables `age` and `money`. We'll write some code to determine if you can buy a car that costs 20000. For this you'll need at least 20000 money and at least an age of 18:

```
var age = 23
var money = 25000

if age >= 18 {
    if money >= 20000 {
        print("Getting a new car, baby!")
    } else {
        print("Sorry, you don't have enough money.")
    }
} else {
    print("Sorry, you're not old enough.")
}
```

Multiple conditions

Mutiple conditions can be chained together using the `&&` (AND) operator and the `||` (OR) operator

The `&&` (AND) operator is used to check if two conditions are simultaneously true. For example consider we have the age of a person stored in a variable `age` and want to determine if the person is a teenager (age is between 13 and 19). We have to check that the age is greater than or equal to 13 **AND** less than or equal to 19. This is accomplished by the code below:

```
var age = 18
if age >= 13 && age <= 19 {
    print("Teenager")
}
```

This is equivalent to the following code:

```
var age = 18
if age >= 13 {
    if age <= 19 {
        print("Teenager")
    }
}
```

The `||` (OR) operator is used to check that at least one of two conditions is true.

Consider again that we have the age of a person stored in a variable `age`. We want to print a warning if

the age is less than or equal to 0 **OR** the age is greater than or equal to 100. This is accomplished by the code below:

```
var age = 123
if age <= 0 || age >= 100 {
    print("Warning age is probably incorrect!")
}
```

Note: The **OR** in programming is not equivalent to the or in everyday language. If someone asks you if you want beef or chicken that means that you can have only one of two. In programming an or statement is also `true` when both conditions are `true` at the same time. For example:

```
var numberOfSisters = 1
var numberOfBrothers = 2

if numberOfSisters > 0 || numberOfBrothers > 0 {
    print("Has siblings")
}
```

To get a better understanding of how **AND**(`&&`) and **OR**(`||`) behave have a look at the truth tables below:

```
// AND
true && true // true
true && false // false
false && true // false
false && false // false

// OR
true || true // true
true || false // true
false || true // true
false || false // false
```

Negating a condition

You can negate a condition using the `!` operator. A negated condition has oposite value to the original condition. i.e. if the initial condition was `true` than it's negation is `false`. If the initial condition is `false` than it's negation is `true`.

For example if we wanted to check if an age is **NOT** the age of a teenager we could use the following code

```
var age = 18
if !(age >= 13 && age <= 19) {
    print("Not a teenager!")
}
```

Note:

```
if condition {
    // DO SOMETHING WHEN CONDITION IS TRUE
} else {
    // DO SOMETHING WHEN CONDITION IS FALSE
}
```

is equivalent of :

```
if !condition {
    // DO SOMETHING WHEN CONDITION IS FALSE
} else {
    // DO SOMETHING WHEN CONDITION IS TRUE
}
```

Note: If you have an if statement with an else branch than it's not recommended to negate the condition.

The below table shows the values of negating some conditions:

```
!true // false
!false // true
!(true && true) // false
!(true || false) // false
!(false || false) // true
```

Exercises

2.1 Max

You are given two numbers `a` and `b` print the largest one.

Code

```
var a = 11
var b = 22

// your code here
```

Example 1

Input:

```
var a = 11
var b = 22
```

Output:

```
22
```

Example 2

Input:

```
var a = 23
var b = 12
```

Output:

```
23
```

Example 3

Input:

```
var a = 2
var b = 4
```

Output:

```
4
```

2.2 Even or Odd

You are given a `number`. Print `even` if the number is even or `odd` otherwise.

Code

```
let number = 2

// your code here
```

Example 1

Input:

```
var number = 1
```

Output:

```
odd
```

Example 2

Input:

```
var number = 12
```

Output:

```
even
```

Hint

Use the remainder (`%`) operator to determine if the number is `even` or `odd`

2.3 Divisibility

You are given two numbers `a` and `b`. Print `"divisible"` if `a` is divisible by `b` and `"not divisible"` otherwise.

Code

```
var a = 12
var b = 3

// your code here
```

Example 1

Input:

```
var a = 22
var b = 11
```

Output:

```
divisible
```

Example 2

Input:

```
var a = 12
var b = 3
```

Output:

```
divisible
```

Example 3

Input:

```
var a = 12
var b = 5
```

Output:

```
not divisible
```

Hint 1

Use the remainder (`%`) operator to check if `b` divides `a` .

Hint 2

To check if `b` divides `a` you need to check if the remainder of the division of `a` to `b` is 0.

2.4 Two of the same

You are given three variables `a` , `b` and `c` . Check if at least two variables have the same value. If that is true print `At least two variables have the same value` otherwise print `All the values are different` .

Code

```
var a = 2
var b = 3
var c = 2

// your code here
```

Example 1

Input:

```
var a = 1
var b = 2
var c = 3
```

Output:

```
All the values are different
```

Example 2

Input:

```
var a = 1
var b = 2
var c = 1
```

Output:

```
At least two variables have the same value
```

Example 3

Input:

```
var a = 3
var b = 3
var c = 3
```

Output:

```
At least two variables have the same value
```

Hint

Use the OR (`||`) operator to chain multiple equality checks

Twist

Check if all three variables are the same.

2.5 Breakfast

You are working on a smart-fridge. The smart-fridge knows how old the eggs and bacon in it are. You know that eggs spoil after 3 weeks (`21 days`) and bacon after one week (`7 days`).
Given `baconAge` and `eggsAge` (in days) determine if you can cook bacon and eggs or what ingredients you need to throw out.
If you can cook bacon and eggs print `you can cook bacon and eggs`.
If you need to throw out any ingredients **for each one** print a line with the text `throw out <ingredient>` (where `<ingredient>` is `bacon` or `eggs`) in any order.

Code

```
var baconAge = 6 // the bacon is 6 days old
var eggsAge = 12 // eggs are 12 days old

// your code here
```

Example 1

Input:

```
var baconAge = 3
var eggsAge = 2
```

Output:

```
you can cook bacon and eggs
```

Example 2

Input:

```
var baconAge = 9
var eggsAge = 20
```

Output:

```
throw out bacon
```

Example 3

Input:

```
var baconAge = 9
var eggsAge = 23
```

Output:

```
throw out bacon
throw out eggs
```

Hint 1

Check for the case where `you can cook bacon and eggs` first.

Hint 2

In the else branch check the ingredients that need to be thrown out.

2.6 Leap Year

You are given a `year`, determine if it's a `leap year`. A `leap year` is a year containing an extra day. It has `366 days` instead of the normal `365 days`. The extra day is added in February, which has `29 days` instead of the normal `28 days`. Leap years occur every `4 years`. `2012` was a `leap year` and `2016` will also be a `leap year`.
The above rule is valid except that every `100 years` special rules apply. Years that are divisible by `100` are not `leap years` if they are not also divisible by `400`. For example `1900` was not a `leap year`, but `2000` was. Print `Leap year!` or `Not a leap year!` depending on the case.

Code

```
let year = 2014

// your code here
```

Example 1

Input:

```
var year = 2000
```

Output:

```
Leap year!
```

Example 2

Input:

```
var year = 2005
```

Output:

```
Not a leap year!
```

Example 3

Input:

```
var year = 1900
```

Output:

```
Not a leap year!
```

Example 4

Input:

```
var year = 1992
```

Output:

```
Leap year!
```

Hint

Use the remainder (`%`) operator to check for divisibility by `4` . Don't forget to check the special case when `year` is divisible by `100` .

2.7 Coin toss

If you use `arc4random()` it will give you a random number. Generate a random number and use it to simulate a coin toss. Print `heads` or `tails` .

Code

```
// this imports the Swift standard library which has arc4random
import Foundation

var randomNumber = arc4random()

// your code here
```

Hint

Use the remainder operator (`%`) to check if the `randomNumber` is even(`heads`) or odd(`tails`).

2.8 Min 4

You are given four variables `a` , `b` , `c` and `d` . Print the value of the smallest one.

Code

```
var a = 5
var b = 6
var c = 3
var d = 4

// your code here
```

Example 1

Input:

```
var a = 3
var b = 5
var c = 4
var d = 2
```

Output:

```
2
```

Example 2

Input:

```
var a = 1
var b = 3
var c = 4
var d = 2
```

Output:

```
1
```

Example 3

Input:

```
var a = 6
var b = 7
var c = 4
var d = 5
```

Output:

```
4
```

Hint

Use a variable to hold the minimum value and initalize it with `a`. Assume that `a` is the smallest value. You'll have to update the value in case `a` is not the smallest value.

2.9 Testing

Test if `number` is divisible by `3`, `5` and `7`. For example `105` is divisible by `3`, `5` and `7`, but `120` is divisible only by `3` and `5` but not by `7`. If `number` is divisible by `3`, `5` and `7` print `number is`

`divisible by 3, 5 and 7` otherwise print `number is not divisible by 3, 5 and 7`.

Code

```
let number = 210

// your code here
```

Example 1

Input:

```
var number = 60
```

Output:

```
number is not divisible by 3, 5 and 7
```

Example 2

Input:

```
var number = 105
```

Output:

```
number is divisible by 3, 5 and 7
```

Hint

Use the remainder (`%`) operator to check for divisibility.

2.10 Point

Find out if the point (`x` , `y`) is inside of the rectangle with the lower-left corner in (`lowX` , `lowY`) and the upper-right in (`highX` , `highY`). Print `inside` or `not inside` depending on the case.

Code

```
var x = 1
var y = 2
var lowX = 1
var lowY = 1
var highX = 3
var highY = 3

// your code here
```

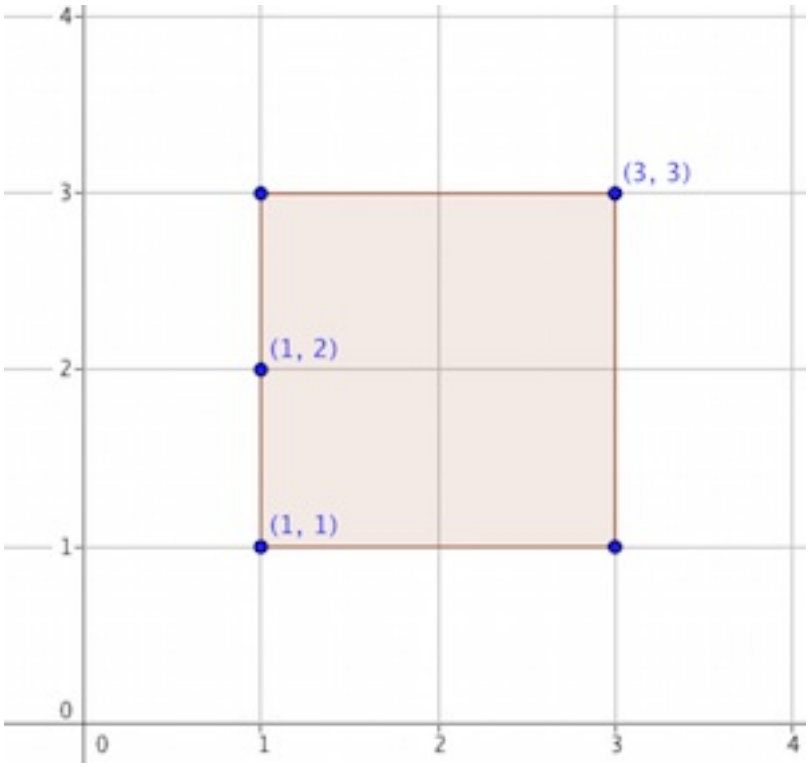
Example 1

Input:

```
var x = 1
var y = 2
var lowX = 1
var lowY = 1
var highX = 3
var highY = 3
```

Output:

```
"inside"
```



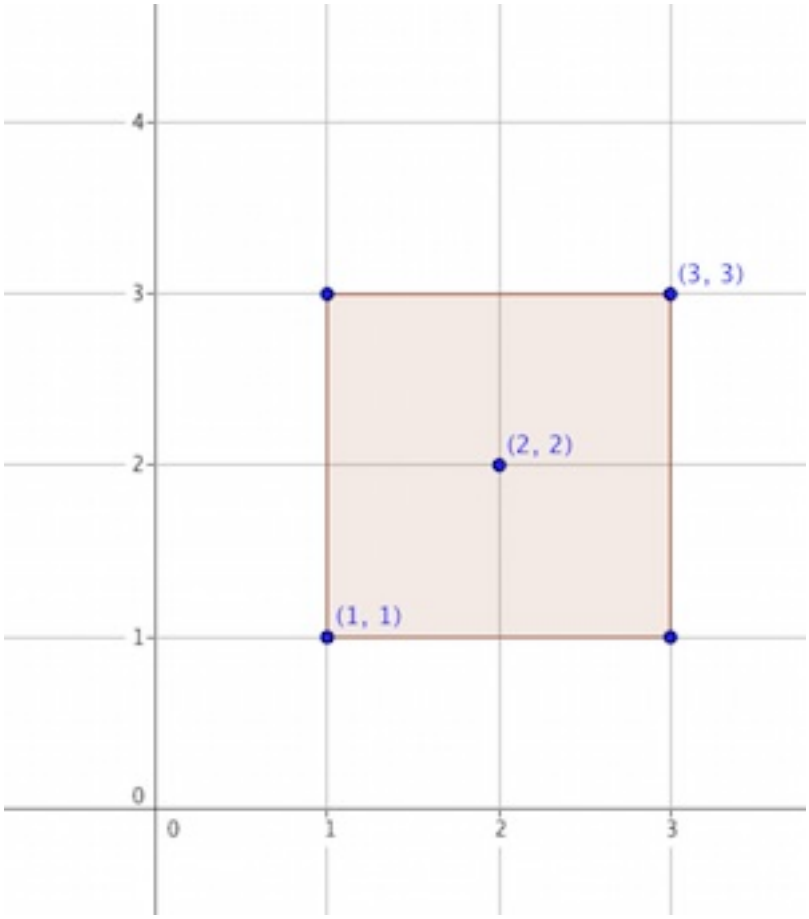
Example 2

Input:

```
var x = 2
var y = 2
var lowX = 1
var lowY = 1
var highX = 3
var highY = 3
```

Output:

```
"inside"
```



Example 3

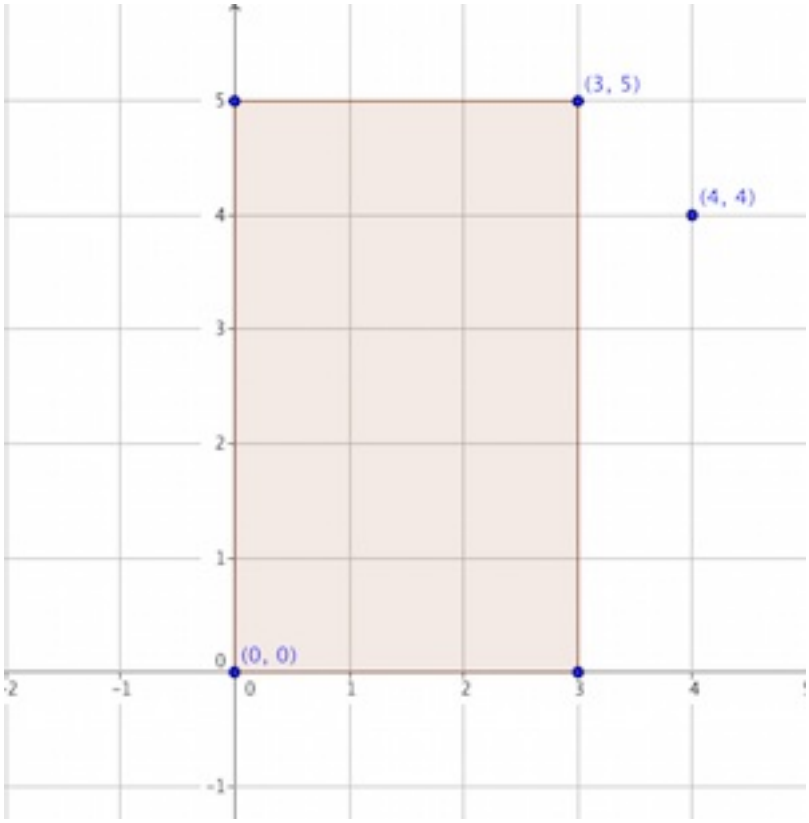
Input:

```
var x = 4
var y = 4
var lowX = 0
var lowY = 0
var highX = 3
```

```
var highY = 5
```

Output:

```
"not inside"
```



Hint

Chain multiple comparisons together using the AND (`&&`) operator.

2.11 Hitpoints

You are working on a videogame where the character has a certain number of hitpoints(HP) ranging from `0` to `100` .
`100` represents full health
`0` represents dead.
You want to add regenerating health to the game using the following rules:

- HP always regenerates up to numbers of the form `X0` (`75` -> `80` , `32` -> `40` ...)
- When HP is below `20` it regenerates up to `20` (`13` -> `20` , `5` -> `20` , ...)
- If the character has `0` HP then he doesn't regenerate life (he's dead)

Given the current hp of the character stored in a variable `hp` print the `hp` the player will have after

regenerating life.

Code

```
var hp = 75

// your code here
```

Example 1

Input:

```
var hp = 60
```

Output:

```
60
```

Example 2

Input:

```
var hp = 26
```

Output:

```
30
```

Example 3

Input:

```
var hp = 12
```

Output:

```
20
```

Example 4

Input:

```
var hp = 4
```

Output:

```
20
```

Example 5

Input:

```
var hp = 95
```

Output:

```
100
```

Hint

Check for the case when `hp` is between 1 and 19 first

Hint

Discard the last digit of `hp` via division

Solutions

2.1 Max

The maximum of two numbers is the largest of the two numbers. i.e. if a > b then the maximum is a otherwise it's be. Note that this problem can also be solved using `>=` or by using the condition `b < a` and interchanging the order of the print statements.

```
var a = 11
var b = 23

if a > b {
    print(a)
} else {
    print(b)
}
```

2.2 Even or Odd

A number is even if it devides evenly into 2. That is the remainder of the division with 2 is 0. Otherwise the number is odd. We'll have to check if the remainder is equal to 0 and print "even" or "odd" accordingly.

```
let number = 2

if number % 2 == 0 {
    print("even")
} else {
    print("odd")
}
```

2.3 Divisibility

A number `a` is divisible by a number `b` if the remainder of the division is `0`. We'll have to check if that is the case and print a message accordingly.

```
var a = 12
var b = 3

if a % b == 0 {
    print("divisible")
} else {
    print("not divisible")
}
```

2.4 Two of the same

To check if at least two variables have the same value we only have to check 3 of the posible 6 pairs(ab, ac, ba, bc, ca, cb) because equality is symmetric (is a = b then b = a).

```
var a = 2
var b = 2
var c = 2

if (a == b) || (a == c) || (b == c) {
    print("At least two variables have the same value")
} else {
    print("All the values are different")
}
```

Twist

```
var a = 2
var b = 2
var c = 2

if (a == b) && (b == c) {
    print("All three variables have the same value")
}
```

2.5 Breakfast

The only case where we can cook bacon and eggs is when the bacon's age is less than or equal to 7 and the egg's age is less than or equal to 21. In code this is equivalent to the conditions `baconAge <= 7` and `eggsAge <= 21`. To check that both these conditions are true we use the `&&` operator. In case one of these conditions is not met we check which ingredients have gone bad (`baconAge > 7` or `eggsAge > 21`) and print a message accordingly.

```
var baconAge = 6
var eggsAge = 12

if baconAge <= 7 && eggsAge <= 21 {
    // bacon and eggs are ok, we can cook
    print("you can cook bacon and eggs")
} else {
    // either eggs or bacon or both are spoiled
    if baconAge > 7 {
        print("throw out bacon")
    }
    if eggsAge > 21 {
        print("throw out eggs")
    }
}
```

2.6 Leap Year

First we'll check for divisibility by 4. Next we'll check if the year is divisible by 100 and not divisible by 400, if that's the case then the year is not a leap year despide being divisible by 4, otherwise it is a leap year. If the year is not divisible by 4 then its certainly not a leap year.

```
let year = 2014
if year % 4 == 0 {
    if year % 100 == 0 && year % 400 != 0 {
        print("Not a leap year!")
    } else {
        print("Leap year!")
    }
} else {
    print(year, terminator: "")
    print("Not a leap year!")
}
```

2.7 Coin toss

`arc4random()` gives us a random integer. Given a random integer we know that 50% of the time it will be even and 50% of the time it will be odd. We can simulate a coin flip by considering the remainder when dividing our number by 2. This effectively simulates a coin flip.

Solution

```
import Foundation

var randomNumber = arc4random()

if randomNumber % 2 == 0 {
    print("heads")
} else {
    print("tails")
}
```

Funny Solution

```
import Foundation

let randomNumber = arc4random()

if randomNumber == 0 {
    print("it fell under the couch")
} else if (randomNumber % 2 == 0) {
    print("tails")
} else {
    print("head")
}
```

2.8 Min 4

We start by assuming that the minimum is equal to `a`. Next we check if `b`, `c` or `d` are less than our current minimum, updating our minimum if that is the case. **Note:** We could have used 4 conditions of the form `if a <= b && a <= c && a <= d` but that would have resulted in a more complicated solution.

```
var a = 5
var b = 6
var c = 3
var d = 4

var min = a

if b < min {
    min = b
}

if c < min {
    min = c
}

if d < min {
    min = d
}

print(min)
```

2.9 Testing

We'll have to check if the number is simultaneously divisible by 3, 5 and 7. We combine these divisibility checks via the `&&` operator.

```
let number = 210

if number % 3 == 0 && number % 5 == 0 && number % 7 == 0 {
    print("number is divisible by 3, 5 and 7")
} else {
    print("number is not divisible by 3, 5 and 7")
}
```

2.10 Point

The point is inside the rectangle if it's x coordinate is greater or equal to the rectangle's lowest x coordinate (lowX) and less than or equal to the rectangles highest x coordinate (highX). We'll also have to do a simillar check for the y coordinate.

```
var x = 1
var y = 2
var lowX = 1
```

```
var lowY = 1
var highX = 3
var highY = 3

if x >= lowX && y >= lowY && x <= highX && y <= highY {
    print("inside")
} else {
    print("not inside")
}
```

2.11 Hitpoints

Lets start with the simple case, if `hp` is greater than `0` and less than `20` we just set it to `20`. Otherwise we want to round `hp` to the next multiple of `10` if `hp` is not already a multiple of `10` (`hp` is divisible by `10`). To round an integer to the next multiple of `10` we divide by `10` and add `1` then multiply by `10`.

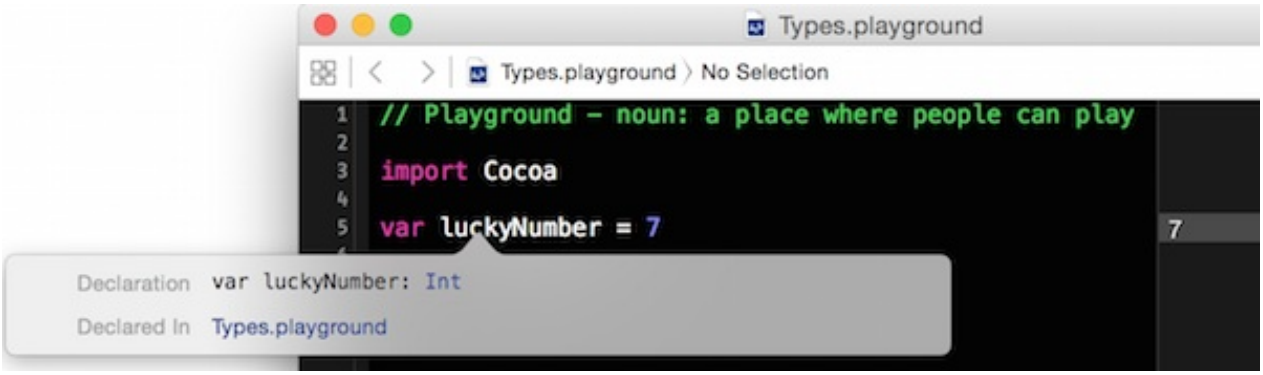
```
var hp = 75

if hp > 0 && hp < 20 {
    hp = 20
} else if hp % 10 != 0 {
    hp = hp / 10
    hp = hp + 1
    hp = hp * 10
}

print(hp)
```

3. Types

Introduction

All the values we've worked with so far have been integer numbers.
All variables and constants in Swift have a type. Think of the type as describing what kind of values a variable can have. The type for integer numbers is `Int`.

You can see the type of a variable in Xcode by option clicking on it's name (hold down option(⌥) and click on its name). A popup will appear were you can see the type of the variable.



Here you can see that our lucky number 7 is of type `Int`.

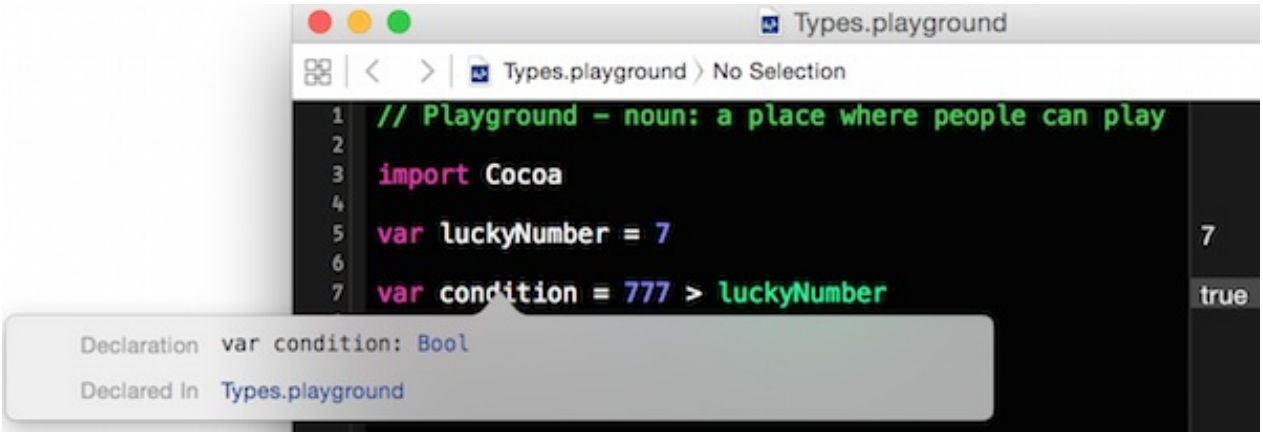
We often need variables of types that aren't `Int`. You already encountered a different type. The expresssions inside an if statement. These values have type `Bool` also known as `Boolean` named after mathematician George Boole who layed out the mathematical foundations of Logic.

Comparision operators (`<, <=, >, >=, ==, !=`) produce values of type `Bool`. Boolean expressions can have only one of 2 types `true` or `false`

For example

```
var luckyNumber = 7

var condition = 777 > luckyNumber
```

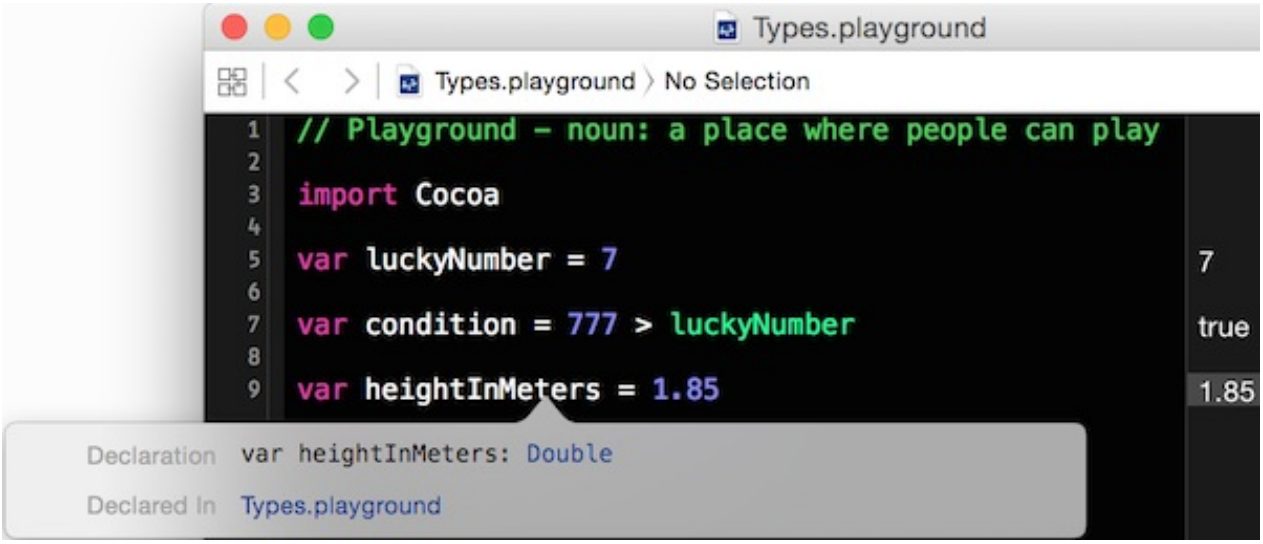
`condition` will be of type `Bool` and have a value of `true` as seen below:



The Double type

What if we want to use numbers of the form (`1.5, 1.7, 1.6,...`)? We can of course do that in Swift! If you declare a variable lets say `heightInMeters` with a value of `1.85` and check its type you'll see that its type is `Double`

```
var heightInMeters = 1.85
```



Variables of type `Double` hold fractional numbers and can be used for calculating fractional quantities. Any number of the form `X.Y` is a `Double` Examples: (35.67, 2.0, 0.5154, ...)

Doubles can be added, subtracted, multiplied and divided using the familiar operators (`+, -, *, /`).

There is no equivalent to the remainder(`%`) operator for doubles.

```
var a = 3.5
var b = 1.25

print(a + b) // 4.75
```

```
print(a - b) // 2.25
print(a * b) // 4.375
print(a / b) // 2.8
```

[spoiler title='Why are fractional numbers called Double?(Optional)' collapse_link='true']
Numbers of type double have a limited precision. consider the following code

```
print(1.0 / 3.0) // 0.333333333333333
```

Mathematically speaking the number (`1.0 / 3.0`) should go on forever having an infinite number of 3s after the decimal `.` . Computers can't hold an infinite amount of information so they truncate the number at some point.

Representing decimal numbers in a computer is done via so called Floating Point Numbers. Represented by the type `Float` in Swift. The `Double` type is a kind of floating point number but compared to the `Float` type it can hold twice as many digits hence the name `Double` . [/spoiler]

Declaring variables of a certain type

To explicitly declare a variable of a certain type you use the syntax:

var `variable` : `Type` = `...`

For example:

```
var integer:Int = 64
var boolean:Bool = false
var double:Double = 7.2
```

If you don't provide a type for a variable a type is automatically inferred for you using the value you provide

Examples:

```
// We don't declare a type for a it is implicitly Int
// because the value 7 is an Int
var a = 7

print(a / 2) // 3
```

```
// We don't declare a type for a it is implicitly Double
// because the value 7.0 is a Double
var a = 7.0
```

```
print(a / 2) // 3.5
```

If you explicilty declare a variable as having type `Double` then you can initialize it with an integer but the variable will hold a `Double`

```
var a:Double = 7 // We explicitly declare a type for a

print(a / 2) // 3.5
```

Type Casting

Initializing a variable of type `Double` with an integer only works if you use a constant value. If you try initializing a variable of type `Double` with a variable of type `Int` then you'll get an error.

```
var a = 64
var b:Double = a // Error
```

To solve this problem we need to convert the value from `Int` to `Double`. Converting a value of some type to a different type is known as `type casting` or just `casting`.

To cast a variable to a certain type we use `TypeName(variableName)` or the `as` operator, `variableName as TypeName`. For example:

```
var a = 64
var b:Double = Double(a) // b = 64.0
var c:Double = a as Double // c = 64.0
```

Casting a `Double` to an `Int` discards all the digits after the decimal point. Note that this digits can't be recovered by casting the variable back to `Double`.

Example:

```
var number = 5.25
var integerNumber = Int(number) // 5
var doubleNumber = Double(integerNumber) // 5.0
```

Exercises

3.1 Average

You are given 2 Doubles `a` and `b`. Print their average.

Code

```
var a = 2.0
var b = 5.0

// your code here
```

Example 1

Input:

```
var a = 2.0
var b = 5.0
```

Output:

```
3.5
```

Example 2

Input:

```
var a = 20.0
var b = 40.0
```

Output:

```
30.0
```

Hint

Adding 2 numbers together and dividing by 2 gives you their average.

3.2 Weighted Average

You are given 3 grades stored in 3 variables of type `Double`: `finalsGrade`, `midtermGrade`, `projectGrade`. These grades are used to compute the grade for a class. `finalsGrade` represents 50% of the grade. `midtermGrade` represents 20% of the grade. `projectGrade` represents 30% of the final grade. Print the grade for the class.

Code

```
var finalsGrade = 2.0
var midtermGrade = 4.0
var projectGrade = 3.0

// your code here
```

Example 1

Input:

```
var finalsGrade = 2.0
var midtermGrade = 5.0
var projectGrade = 3.0
```

Output:

```
2.7
```

Example 2

Input:

```
var finalsGrade = 5.0
var midtermGrade = 5.0
var projectGrade = 5.0
```

Output:

```
5.0
```

Hint

`x%` of a `value` = `value * x / 100`

3.3 Tipping

You have the cost of a meal at a restaurant stored in a variable `mealCost` of type `Double`.
You would like to leave a tip of a certain percentage. The percentage is stored in a variable `tip` of type `Int`.
Print the total cost of the meal.

Code

```
var mealCost:Double = 3.5
var tip:Int = 20 // 20% tip

// your code here
```

Example 1

Input:

```
var mealCost:Double = 3.5
var tip:Int = 20
```

Output:

```
4.2
```

Example 2

Input:

```
var mealCost:Double = 10.0
var tip:Int = 10
```

Output:

```
11.0
```

Hint 1

Don't forget to convert `tip` to `Double`

Hint 2

`x%` of a `value` is equal to `value * x / 100`

3.4 Rounding

You are given a variable `number` of type `Double`. Create a new variable called `roundedNumber` that has at most `1` digit after the decimal dot.

Code

```
var number = 5.1517
// your code here
```

Example 1

Input:

```
var number = 5.1517
```

Expected values:

```
roundedNumber = 5.1
```

Example 2

Input:

```
var number = 32.5
```

Expected values:

```
roundedNumber = 32.5
```

Example 3

Input:

```
var number = 2.0
```

Expected values:

```
roundedNumber = 2.0
```

Hint

Converting a `Double` to an `Int` discards all the digits after the decimal point.

3.5 Above average

You are given three grades obtained by 3 students in a class stored in variables `grade1`, `grade2`, `grade3` of type `Double`.
You are also given your grade in the class stored in a variable `yourGrade` of type `Double`.
Print `above average` if your grade is greater than the class average or `below average`" otherwise.
Note: the average of the class includes your grade.

Code

```
var grade1 = 7.0
var grade2 = 9.0
var grade3 = 5.0
var yourGrade = 8.0

// your code here
```

Example 1

Input:

```
var grade1 = 7.0
var grade2 = 9.0
var grade3 = 5.0
```

```
var yourGrade = 8.0
```

Output:

```
"above average"
```

Example 2

Input:

```
var grade1 = 10.0
var grade2 = 9.0
var grade3 = 10.0
var yourGrade = 9.0
```

Output:

```
"below average"
```

Hint

Compare the average with your grade.

3.6 Fields

A farmer is harvesting wheat from a number of wheat fields, given in a variable `numberOfFields` of type `Int`.
Each field produces the same quantity of wheat given in a variable `wheatYield` of type `Double`.
Sometimes the harvest is increased by `50%` due to favorable weather conditions. You are given this information in a variable `weatherWasGood` of type `Bool`.
Print the total amount of wheat that the farmer will harvest.

Code

```
var numberOfFields:Int = 5
var wheatYield:Double = 7.5
var weatherWasGood:Bool = true

// your code here
```

Example 1

Input:

```
var numberOfFields:Int = 5
var wheatYield:Double = 7.5
var weatherWasGood:Bool = true
```

Output:

```
56.25
```

Example 2

Input:

```
var numberOfFields:Int = 5
var wheatYield:Double = 7.5
var weatherWasGood:Bool = false
```

Output:

```
37.5
```

Solutions

3.1 Average

We'll just have to add our numbers and divide by 2. Keep in mind that division has higher predence then addition so `(a + b) / 2` is correct while `a + b / 2` would not be correct.

```
var a = 2.0
var b = 5.0

print((a + b) / 2)
```

3.2 Weighted Average

To get `X%` of a value we have to multiply the value by `X / 100`. For 50, 20 and 30 percent we have to multiply with 0.5, 0.2 and 0.3 respectively, these are our weights in the average. The weighted average we need will be the grades multiplied by the correspoding weight.

```
var finalsGrade = 2.0
var midtermGrade = 4.0
var projectGrade = 3.0

print(0.5 * finalsGrade + 0.2 * midtermGrade  + 0.3 * projectGrade)
```

3.3 Tipping

First we'll calculate how much the tip will cost. We use the same approach as `Problem 3.2` to get a percentage using multiplication. Keep in mind that we have to conver the tip to type `Double`, multiplying a variable of type `Int` with a variable of type `Double` is not allowed.

```
var mealCost:Double = 3.5
var tip:Int = 20

var tipCost = mealCost * Double(tip) / 100.0
var totalCost = mealCost + tipCost

print(totalCost)
```

3.4 Rounding

Converting a `Double` to an `Int` will discard all the digits after the decimal point we want to keep a digit

tough so we'll multiply the number by `10` first then we'll discard all the digits by converting to `Int`, finally we'll convert back to `Double` and divide by `10` to shift the decimal point to the right.

```
var number = 5.1517

var intNumber = Int(number * 10.0)

var roundedNumber = Double(intNumber) / 10.0
```

3.5 Above average

First we'll calculate the `averageGrade` of the class. Then we are goind to determine if you are above or below that grade.

```
var grade1 = 7.0
var grade2 = 9.0
var grade3 = 5.0
var yourGrade = 8.0

var averageGrade = (grade1 + grade2 + grade3 + yourGrade) / 4.0

if yourGrade > averageGrade {
    print("above average")
} else {
    print("below average")
}
```

3.6 Fields

The total yield is given by `numberOfField * wheatYield`, keep in mind that you'll have to convert `numberOfFields` to `Double` as its an `Int`. Finally We want to multiply the yield by `1.5` if the year was good.

```
var numberOfFields:Int = 5
var wheatYield:Double = 7.5
var weatherWasGood:Bool = true

var totalYield = Double(numberOfFields) * wheatYield
if (weatherWasGood == true) {
    totalYield = totalYield * 1.5
}

print(totalYield)
```

4. Loops

Introduction

Let's make make some pancakes!

So far we only looked at programs that have a fixed number of steps. For example look the *algorithm* to make a pancake:

```
- put 1/4 cup of batter in the frying pan
- cook for about 2 minutes
- flip and cook for another 2 minutes
- remove the pancake
```

How would the algorithm to make 10 pancakes would look? Would it be much different?

```
10 times do:
    - put 1/4 cup of batter in the frying pan
    - cook for about 2 minutes
    - flip and cook for another 2 minutes
```

```
    - remove the pancake
```

Loops let you describe repetitive processes. They could have a fixed amount of steps like the example above. Or they could have an unknow number of steps, for example a more realistic algorithm for making pancakes:

```
while you have pancake batter do:
    - put 1/4 cup of batter in the frying pan
    - cook for about 2 minutes
    - flip and cook for another 2 minutes
    - remove the pancake
```

`while`

A `while` loop performs a set of statements until a condition becomes `false`.

while `condition` {
  `statements`
}

For example in order to `print` all the numbers from 1 to 10. We need to create a variable with the initial value of 1. Print the value and increase it by one and until it becomes bigger than 10.

```swift
var i = 1
while i <= 10 {
    print(i)
    i = i + 1
}
```

`repeat`

`repeat` loops while a condition is met. The difference between a `while` and a `repeat` loop is that the repeat loop evaluates the condition after executing the statements from the loop.

repeat {
  `statements`
} **while** `condition`

```swift
var i = 1
repeat {
    print(i)
    i = i + 1
} while i < 10
```

Both `while` and `repeat` are best used in loops where the numbers of stepts is unkown. Take for example the algorithm of converting a number to binary: divide the number by two until it becomes 0. Write the reminders from right to left to get the binary form of the number.

```
var number = 123

var binary = 0
var digit  = 1

while number > 0 {
    let reminder = number % 2

    // add the new digit to the number
    binary = digit * reminder + binary

    // move the digit to the left
    digit *= 10

    // remove the last binary digit
    number /= 2
}

binary // 1111011
```

`for` loops

Swift provides two kinds of loops that perform a set of statements a certain number of times:

The **for-in** loop performs a set of statements for each item in a range or collection.

Swift also provides two range operators `lowerBound...upperBound` and `lowerBound..<upperBound`, as a shortcut for expressing a range of values.

```
1...3 // 1, 2, 3
1..<3 // 1, 2
```

for `value` **in** `range` {
   `statements`
}

```
// prints 1-10
for i in 1...10 {
    print(i)
}

// prints 0-9
for i in 0..<10 {
    print(i)
}
```

If `lowerBound` is greater than `upperBound` you code will crash:

```
// this will crash - don't do it! :)
for i in 10...1 {
    print(i)
}
```

If you want to loop on a range in reverse order you can use the `reversed` range method:

```
// this will print the numbers from 10 to 1
for i in (1...10).reversed() {
    print(i)
}
```

stride

Stride is a function from the swift standard library that returns the sequence of values `start`, `start + stride`, `start + 2 * stride`, ... `end`) where last is the last value in the progression that is less than end.

The `stride` function works with any kind of number:

```
stride(from: 1, to: 10, by: 2)  // 1, 3, 5, 7, 9
stride(from: 1, to: 2, by: 0.1) // 1.0, 1.1 ... 1.9
```

Let's take for example a program that counts from 1 to 10 by 3:

```
for i in stride(from: 1, to: 10, by: 3) {
    print(i)
}
```

You can use `stride` to create decreasing sequences if the `stride` parameter is negative:

```
for i in stride(from: 3, to: 1, by: -1) {
    print(i)
}
// prints: 3 2 1
```

`print` and terminators

For the drawing exercises below you will need use the `terminator` parameter for the `print` function. The `terminator` refers to the thing that is printed at the end. The default terminator is the new line character `"\n"`.

- `print(value)` will print the value and a new line
- `print(value, terminator: "")` will print the value

```
print("BAT", terminator: "") // prints BAT
print("MAN", terminator: "") // prints MAN
print("") // prints a newline character
// BATMAN

print("BAT")
// BAT
print("MAN")
// MAN
```

Executing a statement multiple times

Sometimes you just want to execute some statements multiple times but don't care about having an index. A swift convention in `for` loops is to use `_` as the loop variable name when you don't intend to use the variable in the loop.

For examplet to to print "Hello World" 5 times you can use:

```
for _ in 1...5 {
    print("Hello World")
}
```

Naming your loop variable `_` is useful because you immediately tell that the variable is not used in the loop.

Exercises

4.1 Chalkboard

Write a program that writes "I will not skip the fundamentals!" `N` times.

Code

```
var N = 10

// your code here
```

Example 1

Input:

```
var N = 3
```

Output:

```
I will not skip the fundamentals!
I will not skip the fundamentals!
I will not skip the fundamentals!
```

Example 2

Input:

```
var N = 5
```

Output:

```
I will not skip the fundamentals!
I will not skip the fundamentals!
I will not skip the fundamentals!
I will not skip the fundamentals!
I will not skip the fundamentals!
```

Hint

The solution to a similar problem was shown in the theory, you can use either `for` or `while` to solve this problem.

4.2 Squares

Print the first `N` square numbers. A square number, also called perfect square, is an integer that is obtained by squaring some other integer; in other words, it is the product of some integer with itself (ex. `1`, `4` = `2` *`2`*, `9` = *`3`* `3` ...).

Code

```
var N = 10

// your code here
```

Example 1

Input:

```
var N = 2
```

Output:

```
1
4
```

Example 2

Input:

```
var N = 5
```

Output:

```
1
4
9
16
```

```
25
```

4.3 Powers of 2

Print the powers of `2` that are less than or equal to `N`.

Code

```
var N = 10

// your code here
```

Example 1

Input:

```
var N = 5
```

Output:

```
2
4
```

Example 2

Input:

```
var N = 100
```

Output:

```
2
4
8
16
32
64
```

Hint

The first power of `2` is `2`. Given a power of `2`, `power`, the next power of `2` is `power * 2`.

4.4 Alternative Counting

Write all the numbers from 1 to `N` in alternative order, one number from the left side (starting with one) and one number from the right side (starting from `N` down to `1`).

Code

```
var N = 5

// your code here
```

Example 1

Input:

```
var N = 4
```

Output:

```
1
4
2
3
```

Example 2

Input:

```
var N = 9
```

Output:

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

```
5
```

Hint 1

Use two variables to remember the `left` and `right` index that you need to print next.

Hint 2

There's a special case you'll have to handle when `N` is odd.

4.5 Square

Given an integer `N` draw a square of `N x N` asterisks. Look at the examples.

Code

```
var N = 4

// your code here
```

Example 1

Input:

```
var N = 1
```

Output:

```
*
```

Example 2

Input:

```
var N = 2
```

Output:

```
**
**
```

Example 3

Input:

```
var N = 3
```

Output:

```
***
***
***
```

Hint 1

Try printing a single line of `*` first.

Hint 2

You can use `print("")` to print an empty line.

Twist

- draw an empty square
- draw an empty square, but instead of asterisks, use `+` for corners, `-` for the top and bottom sides and `|` for the left and right ones

4.6 Rectangle

Given two integers `N` and `M` draw a rectangle of `N x M` asterisks. Look at the examples.

Code

```
var N = 3
var M = 7

// your code here
```

Example 1

Input:

```
var N = 1
var M = 3
```

Output:

```
***
```

Example 2

Input:

```
var N = 2
var M = 2
```

Output:

```
**
**
```

Example 3

Input:

```
var N = 3
var M = 7
```

Output:

```
*******
*******
*******
```

Hint

You'll need to change the bounds of one of the loops.

Twist

- draw an empty rectangle
- draw an empty rectangle, but instead of asterisks, use `+` for corners, `-` for the top and bottom sides and `|` for the left and right ones

4.7 Triangle

Given an integer `N` draw a triangle of asterisks. The triangle should have `N` lines, the `i`-th line should have `i` asterisks on it.

Code

```
var N = 4

// your code here
```

Example 1

Input:

```
var N = 1
```

Output:

```
*
```

Example 2

Input:

```
var N = 3
```

Output:

```
*
**
```

```
***
```

Example 3

Input:

```
var N = 4
```

Output:

```
*
**
***
****
```

Hint

First you'll want to print a single `*`. Then you'll want to print 2 `*`, then 3 `*`. How many stars will you print at the `i-th` iteration?

Twist

- draw an empty triangle
- draw an empty triangle, but instead of asterisks, use `+` for corners, `-` for the bottom side, `|` for the left side and `\` for the right side

4.8 Pyramid

Given an integer `N` draw a pyramid of asterisks. The pyramid should have `N` lines. On the **i**-th line there should be `N-i` spaces followed by `i*2-1` asterisks.

Code

```
var N = 3

// your code here
```

Example 1

Input:

```
var N = 1
```

Output:

```
*
```

Example 2

Input:

```
var N = 2
```

Output:

```
 *
***
```

Example 3

Input:

```
var N = 3
```

Output:

```
  *
 ***
*****
```

Example 4

Input:

```
var N = 4
```

Output:

```
   *
  ***
 *****
*******
```

Hint 1

How many stars do you have to print at each iteration?

Hint 2

How many spaces do you have to print at each iteration?

Hint 3

What's a general formula for the sequence: `1`, `3`, `5`, `7` ?

4.9 Rhombus

Given an integer `N` draw a rhombus of asterisks, like the ones in the examples.

Code

```
var N = 4

// your code here
```

Example 1

Input:

```
var N = 1
```

Output:

```
*
```

Example 2

Input:

```
var N = 2
```

Output:

```
 *
***
 *
```

Example 3

Input:

```
var N = 3
```

Output:

```
  *
 ***
*****
 ***
  *
```

Example 4

Input:

```
var N = 4
```

Output:

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

Hint 1

Notice that the upper half of the rhombus is the pyramid from the previous exercise.

Hint 2

The second half is the pyramid only inverted and with the last line removed.

4.10 Aztec Pyramid

Given an integer `N` draw a Aztec pyramid of asterisks, like the ones in the examples.

Code

```
var N = 3

// your code here
```

Example 1

Input:

```
var N = 1
```

Output:

```
**
**
```

Example 2

Input:

```
var N = 2
```

Output:

```
  **
  **
******
```

```
******
```

Example 3

Input:

```
var N = 3
```

Output:

```
    **
    **
  ******
  ******
**********
**********
```

Hint 1

You'll have to draw each line twice.

Hint 2

How many stars are on each line?

Hint 3

What's the general term for the sequence `2`, `6`, `10`, `14`, ... ?

4.11 Chess Board

Given an integer `N` draw a chess board of size `N x N`. Each line of the chess board should have spaces and number signs(`#`) alternating. A space represents a white cell and the number sign a black one. The chess board should be bordered using `+`, `-` and `|` like in the examples below.

Code

```
var N = 8

// your code here
```

Example 1

Input:

```
var N = 1
```

Output:

```
+-+
|#|
+-+
```

Example 2

Input:

```
var N = 3
```

Output:

```
+---+
|# #|
| # |
|# #|
+---+
```

Example 3

Input:

```
var N = 5
```

Output:

```
+-----+
|# # #|
| # # |
|# # #|
| # # |
|# # #|
+-----+
```

Example 4

Input:

```
var N = 8
```

Output:

```
+--------+
|# # # # |
| # # # #|
|# # # # |
| # # # #|
|# # # # |
| # # # #|
|# # # # |
| # # # #|
+--------+
```

Hint 1

First consider how to draw the top and bottom border.

Hint 2

How can you alternate between " " and "#" ? Consider the remainder(`%`) when dividing the indices of the loops by `2`.

4.12 Fibonacci

Write a program that prints the first `N` Fibonacci numbers. The first two Fibonacci numbers are `1`, the rest of the elements are the sum of the previous two. The first seven numbers are `1`, `1`, `2`, `3`, `5`, `8` and `13`.

Code

```
var N = 3

// your code here
```

Example 1

Input:

```
var N = 3
```

Output:

```
1
1
2
```

Example 2

Input:

```
var N = 6
```

Output:

```
1
1
2
3
5
8
```

Hint

Use two variables `a = 1` and `b = 0`. At each step `a` should be the *i*-th Fibonacci number, and `b` the *i-1*-th.

4.13 Leap Years

Write a program that prints the next `N` `leap years` starting with `leapYear`. A `leap year` is a year containing an extra day. It has `366 days` instead of the normal `365 days`. The extra day is added in February, which has `29 days` instead of the normal `28 days`. `Leap years` occur every `4` years, `2012` was a leap year and 2016 will be a `leap year`.
Except that every `100` years special rules apply. Years that are divisible by `100` are not `leap years` if they are not divisible by `400`. For example `1900` was not a `leap year`, but `2000` was.

Code

```
var N = 3
```

```
// the current leap year
var leapYear = 2016

// your code here
```

Example 1

Input:

```
var N = 6

// the current leap year
var leapYear = 2016
```

Output:

```
2016
2020
2024
2028
2032
2036
```

Example 2

Input:

```
var N = 3

// the current leap year
var leapYear = 1996
```

Output:

```
1996
2000
2004
```

Hint

Keep in mind that the variable `leapYear` is a leap year to begin with. Given a leap year how can you generate the next leap year ?

4.14 Reverse

You are given a `number`. Print the number with the digits in reversed order.

Code

```
var number = 1234

//your code here
```

Example 1

Input:

```
var number = 12345
```

Output:

```
54321
```

Example 2

Input:

```
var number = 23432
```

Output:

```
23432
```

Example 3

Input:

```
var number = 1000
```

Output:

```
0001
```

Hint

To get the last digit use the `%` operator (the reminder to `10` is the last digit). To get the number without the last digit divide by `10`.

4.15 GCD

You are given two numbers `a` and `b`. Find and print the greatest common divisor of `a` and `b`.
The greatest common divisor of `a` and `b` is the largest number that divides both `a` and `b`.

Code

```
var a = 24
var b = 18

// your code here
```

Example 1

Input:

```
var a = 24
var b = 18
```

Output:

```
6
```

Example 2

Input:

```
var a = 21
var b = 13
```

Output:

```
1
```

Example 3

Input:

```
var a = 12
var b = 36
```

Output:

```
12
```

Hint 1

The smallest divisor of `a` and `b` is `1`. And the greatest value can be at most `min(a, b)`.

Hint 2

Find the minimum of `a` and `b` and store it in `maxDiv`.
Write a for loop that goes from `1` to `maxDiv` and check each number.

4.16 Prime numbers

You are given a `number`. Print `"prime"` if the number is a prime and `"not prime"` otherwise.
A number is a prime if it has **exactly** 2 distinct divisors (1 and itself).

Code

```
var number = 17

// your code here
```

Example 1

Input:

```
var number = 2
```

Output:

```
prime //2 is only divisible by 1 and 2
```

Example 2

Input:

```
var number = 3
```

Output:

```
prime //3 is only divisible by 1 and 3
```

Example 3

Input:

```
var number = 15
```

Output:

```
not prime //15 is divisible by 1,3,5 and 15
```

Example 4

Input:

```
var number = 17
```

Output:

```
prime //17 is only divisible by 1 and 17
```

Example 5

Input:

```
var number = 1
```

Output:

```
not prime //1 is only divisible by 1 (needs exactly 2 divisors to be a prime, only has 1)
```

Hint

Count the number of divisors of the input number.

4.17 Factoring numbers

You are given a `number`. Decompose `number` into prime factor and write it as an expression(see examples).

Code

```
var number = 10

// your code here
```

Example 1

Input:

```
var number = 24
```

Output:

```
24 = 2 * 2 * 2 * 3
```

Example 2

Input:

```
var number = 12
```

Output:

```
12 = 2 * 2 * 3
```

Example 3

Input:

```
var number = 15
```

Output:

```
15 = 3 * 5
```

Example 4

Input:

```
var number = 7
```

Output:

```
7 = 7
```

Example 5

Input:

```
var number = 4
```

Output:

```
4 = 2 * 2
```

Hint

Dividing a number by one of it's factors will result in a smaller number. A number can have a prime factor divisor multiple times, ex: `8 = 2 * 2 * 2`

Twist

Show the powers of each prime factor instead of showing it multiple times.

4.18 Free of squares

Find all numbers free of squares less than or equal to `N`. A number is free of square if it cannot be divided by any square number except `1`.

Code

```
var N = 10

// your code here
```

Example 1

Input:

```
var N = 10
```

Output:

```
1
2
3
5
6
7
10
```

Example 2

Input:

```
var N = 30
```

Output:

```
1
2
3
5
6
7
10
11
13
14
15
17
19
21
22
23
26
29
30
```

Solutions

4.1 Chalkboard

We'll want to print the statement "I will not skip the fundamentals" in a loop `N` times, we can use a `for` or a `while` loop for this.

Solution 1

```
var N = 10

// with a while loop
var times = 0
while times < N {
    print("I will not skip the fundamentals!")
    times = times + 1
}
```

Solution 2

```
var N = 10
// with a for loop
for _ in 1...N {
    print("I will not skip the fundamentals!")
}
```

4.2 Squares

We'll want to iterate from `1` to `N` in a loop. In the solution `cnt` is our loop index(also called `counter`). The square number we want to print for each index is `cnt * cnt`.

```
var N = 10

var cnt = 1

while cnt <= N {
    print(cnt * cnt)

    cnt = cnt + 1
}
```

4.3 Powers of 2

We'll want to intialize a loop variable `power` to 2. We'll repeat a `while` loop in which we print `power` and

double its value while `power <= N`. This will print all the powers of 2 less then or equal to `N`.

```
var N = 10

var power = 2

while power <= N {
    print(power)
    power = power * 2
}
```

4.4 Alternative Counting

We'll use a `while` loop where we simultaneously increase a variable `left`, initialised to `1` and decrease a variable `right`, initialized to `N`. We repeat these steps until `left becomes greater than or equal to right`. We'll also have to handle the case when `left` becomes equal to `right` once our loop has finished, this will only happen for odd `N`.

```
var N = 5

var left = 1
var right = N

while left < right {
    print(left)
    print(right)
    left += 1
    right -= 1
}

if left == right {
    print(left)
}
```

4.5 Square

First consider the problem of printing a single line of `N` asterisks. we can do this using a for loop that makes `N` calls to the `print("*", terminator: "")` statement. After printing a line of asterisks we want to print a new line. Remember that this is done via the `print("")` statement. Now we want to repeat our loop for printing a line of asterisks in another for loop. This gives us 2 nested loops that solve our problem.

```
var N = 4

for i in 1...N {
    for j in 1...N {
        print("*", terminator: "")
    }
    print("")
}
```

4.6 Rectangle

The solution is simmilar to `Problem 4.5`. The only difference is that the inner loop will have `M` iterations instead of `N` (We want to draw `N` lines each with `M` asterisks)

```
var N = 3
var M = 7

for i in 1...N {
    for j in 1...M {
        print("*", terminator: "")
    }
    print("")
}
```

4.7 Triangle

Again we'll want to use 2 nested loops. Our inner loop won't go from 1 to a fixed value now. Instead it will go from 1 to the value of the index in the inner loop such that we progressively print more asterisks per line.

```
var N = 3

for i in 1...N {
    for j in 1...i {
        print("*", terminator: "")
    }
    print("")
}
```

4.8 Pyramid

First note now many asterisks we draw on each line `(1, 3, 5, 7, 9, ...)`, this sequence is given by `2 * i - 1`. For each iteration `i` we'll also want to draw some empty spaces. Notice that the empty space has the form of an inverted triangle. This becomes more apparent if we replace the empty space with `#`.

```
###*###
##***##
#*****#
*******
```

The number of empty spaces we have to draw is equal to `N - i - 1`.

```
var N = 3

for i in 1...N {
    for j in 0..<(N-i) {
        print(" ", terminator: "")
    }

    for j in 1...2*i-1 {
        print("*", terminator: "")
    }
    print("")
}
```

4.9 Rhombus

To draw the rhombus we are going to draw the pyramid + a reversed pyramid.

```
let N = 4

for i in 1...N {
    for j in 0..<(N-i) {
        print(" ", terminator: "")
    }

    for j in 1...2*i-1 {
        print("*", terminator: "")
    }
    print("")
}

if (N > 1) {
    for j in 2...N {
        var i = N - j + 1
        for k in 0..<(N-i) {
            print(" ", terminator: "")
        }

        for k in 1...2*i-1 {
            print("*", terminator: "")
        }
        print("")
    }
}
```

4.10 Aztec Pyramid

To draw the aztec pyramid we are going to start from the pyramid code and double the space and the asterisk print statements(`print(" ") -> print(" ") and print("*") -> print("**")`) - this will double the width of the pyramid. Then we are going to double the inner nested loop to draw each step twice.

```
let N = 3
```

```
for i in 1...N {
    for _ in 1...2 {
        for _ in 0..<(N-i) {
            print("  ", terminator: "")
        }

        for _ in 1...2*i-1 {
            print("**", terminator: "")
        }
        print("")
    }
}
```

4.11 Chess Board

Drawing the chess board requires us to solve two problems: drawing the border and drawing the board. The board can be printed with two nested loops - the coloring will depend on the indexes of each cell. To draw the border we can print the top and bottom separately from the board. The left and right margins will be created by printing a `|` character before and after each line of the board.

```
let N = 8

// prints the top border
print("+", terminator: "")
for _ in 1...N {
    print("-", terminator: "")
}
print("+")

for i in 1...N {
    // prints the left border
    print("|", terminator: "")
    for j in 1...N {
        if i % 2 == j % 2 {
            print("#", terminator: "")
        } else {
            print(" ", terminator: "")
        }
    }
    // prints the right border a a new line
    print("|")
}

// prints the bottom border
print("+", terminator: "")
for _ in 1...N {
    print("-", terminator: "")
}
print("+")
```

4.12 Fibonacci

We notice that the series remains the same if we start with 0, 1 and then add the last two. Use two variables `a = 1` and `b = 0`. At each step `a` should be the *i*-th Fibonacci number, and `b` the *i-1*-th.

```
var N = 10

var a = 1
var b = 0

for _ in 1...N {
    print(a)
    var tmp = a + b
    b = a
    a = tmp
}
```

4.13 Leap Years

We start a while loop which stops after we print `N` leap years. We print the current leap year and jump to the next one - 4 years into the future - except for those special case in which the year is divisible by 100 and not by 400 - then we need to jump 4 more years.

```
var N = 5

// the current leap year
var leapYear = 2016

// the number of leap years that were printed so far
var cnt = 0

// until we print N years
while cnt < N {
    // print the next leap year
    print(leapYear)

    // increase the counter
    cnt += 1

    // go to the next leap year
    leapYear += 4
    if leapYear % 100 == 0 && leapYear % 400 != 0 {
        leapYear += 4
    }
}
```

4.14 Reverse

We can get the last digit of a number if we calculate the reminder to 10. We can remove the last digit from a number if we divide it by 10. Print the last digit, then remove it. If we repeat these steps until the number reaches 0 we will print all the digits in reverse order.

```
var number = 1234

while number > 0 {
    print(number % 10, terminator: "")
    number /= 10
```

```
}
```

4.15 GCD

The smallest divisor of `a` and `b` is 1. And the greatest value can be at most `min(a, b)`, this will be stored in `maxDiv`. Using a loop we check each number from 1 to `maxDiv` as a posible common divisor of `a` and `b`.

```
var a = 24
var b = 18

var maxDiv = a

if b < maxDiv {
    maxDiv = b
}

var gcd = 1

for i in 1...maxDiv {
    if (a % i == 0) && (b % i == 0){
        gcd = i
    }
}

print(gcd) // 6
```

4.16 Prime numbers

We count the number of divisor of `number` and store the result in `numberOfDivisors`. If `numberOfDivisors` if 2 then the number is prime.

```
var number = 17

var numberOfDivisors = 0

for i in 1...number {
    if number % i == 0 {
        numberOfDivisors += 1
    }
}

if numberOfDivisors == 2 {
    print("prime")
} else {
    print("not prime")
}
```

4.17 Factoring numbers

To decompose a number into prime factor we have to repeatedly divide the number by the smallest number that can divide it and continue with the result. For example to factor 12 the smallest number than can divide it is 2. 12 / 2 is 6. 6 is also divisible by 2 the remaining number is 3 which is prime - so 12 = 2 *2* 3.

```
var number = 10
print("\(number) = ", terminator: "")

var isFirst = true

for i in 2...number {
    if number % i == 0 {
        while (number % i == 0) {
            number /= i

            if isFirst {
                isFirst = false
            } else {
                print(" * ", terminator: "")
            }

            print(i, terminator: "")
        }
    }
}
```

Twist

```
var number = 72 // 2^3 * 3^2

print("\(number) = ", terminator: " ")

var isFirst = true

for i in 2...number {
    if number % i == 0 {
        var put = 0
        while (number % i == 0) {
            number /= i
            put += 1
        }
        if isFirst {
            isFirst = false
        } else {
            print(" * ", terminator: " ")
        }
        print("\(i)^\(put)", terminator: " ")
    }
}
```

4.18 Free of squares

To check if a number is free of squares we are going to decompose it into prime factor and count the frequency of each one - if at leat one appeared more than one then the number is not free of squares. We

are going to do this for all number smaller than `N`.

```
var N = 10

print(1)

for i in 2...N {
    var isFree = true

    var a = i

    for j in 2...a {
        if a % j == 0 {
            var put = 0
            while (a % j == 0) {
                a /= j
                put += 1
            }
            if put > 1 {
                isFree = false
            }
        }
    }

    if isFree {
        print(i)
    }
}
```

5. Strings

A string is an ordered collection of characters, such as `"We ❤ Swift"` or `"eat, sleep, code, repeat!"`. In Swift strings are represented by the `String` type which is a collection of values of `Character` type.

Creating strings

You can include predefined `String` values within your code as string literals. A string literal is a fixed sequence of textual characters surrounded by a pair of double quotes ("").

```
let aString = "Hello"
```

Take note of a special case of string, the empty string.

```
var emptyString = "" // empty string literal
```

You can create a new string by concatenating two strings using the `+` operator.

```
let newString = "Hello" + " swift lovers" // "Hello swift lovers"
```

String interpolation can also be used to combine strings. By using the `\(<value>)` syntax inside a string literal.

```
let bat = "BAT"
let man = "MAN"

// "BATMAN" - \(bat) will be replaced with "BAT" and \(man) with "MAN"
let batman = "\(bat)\(man)"

print("\(bat) + \(man) = \(batman)")
// "BAT + MAN = BATMAN"
```

Characters

In some cases you will want to work with the individual characters that make up the string. To do that you can use the **for-in** syntax. A string exposes the list of characters with the `characters` property.

```
var someString = "this string has 29 characters"
```

```
for character in someString.characters {
    // to convert a Character into a String use string interpolation
    // you will need to do this in order to compare characters
    var characterAsString = "\(character)"

    print(characterAsString)
}
```

To see how many character a string has we can use the `count` property on the `characters` property of the string.

```
var string = "This string has 29 characters"
print(string.characters.count) // 29
```

Comparing

You can compare string using the same operators as numbers, but usually you only care about equality.

```
"We ❤ Swift" == "We ❤ Swift" // true
"We ❤ Swift" == "we ❤ swift" // false - string comparison is case sensitive

"We ❤ Swift" != "We ❤ Swift" // false
"We ❤ Swift" != "we ❤ swift" // true
```

Exercises

5.1 Full name

You are given the `firstName` and `lastName` of a user. Create a string variable called `fullName` that contains the full name of the user.

Code

```
var firstName = "Andrei"
var lastName = "Puni"

// your code here
```

Example 1

Input:

```
var firstName = "Andrei"
var lastName = "Puni"
```

Output:

```
"Andrei Puni"
```

Example 2

Input:

```
var firstName = "Steve"
var lastName = "Jobs"
```

Output:

```
"Steve Jobs"
```

Hint

Use string concatenation or use string interpolation.

5.2 Sum

You are given two numbers `a` and `b`. Compute the sum of `a` and `b` and create a string stored in a variable named `formattedSum` that contains the `sum` written like bellow:

```
For a = 2 and b = 5

formattedSum = "2 + 5 = 7"

For a = 12 and b = 19

formattedSum = "12 + 19 = 31"
```

Code

```
var a = 14
var b = 23

// your code here
```

Example 1

Input:

```
var a = 14
var b = 19
```

Expected values:

```
formattedSum = "14 + 19 = 31"
```

Example 2

Input:

```
var a = 12345
var b = 98765
```

Expected values:

```
formattedSum = "12345 + 98765 = 111110"
```

Hint

Use string interpolation.

5.3 Replace

You are given a string stored in the variable `aString`. Create new string named `replacedString` that contains the characters of the original string with all the occurrences of the character `"e"` replaced by `"*"`.

Code

```
var aString = "Replace the letter e with *"

// your code here
```

Example 1

Input:

```
var aString = "Replace the letter e with *"
```

Expected values:

```
replacedString = "R*plac* th* l*tt*r * with *"
```

Example 2

Input:

```
var aString = "Replace the letter e with *"
```

Expected values:

```
replacedString = "R*plac* th* l*tt*r * with *"
```

Example 3

Input:

```
var aString = "eeee eeee"
```

Expected values:

```
replacedString = "**** ****"
```

Example 4

Input:

```
var aString = "Did you know that Kathy is throwing a party tonight?"
```

Expected values:

```
replacedString = "Did you know that Kathy is throwing a party tonight?"
```

Hint

Create `replacedString` step by step by iterating over the characters of `aString`

5.4 Reverse

You are given a string stored in variable `aString`. Create a new string called `reverse` that contains the original string in reverse order. Print the reversed string.

```
"Hello" -> "olleH"
"We ❤ Swift" -> "tfiwS ❤ eW"
```

Code

```
var aString = "this string has 29 characters"

// your code here
```

Example 1

Input:

```
var aString = "Hello"
```

Expected values:

```
reverse = "olleH"
```

Output:

```
"olleH"
```

Example 2

Input:

```
var aString = "We ❤ Swift"
```

Expected values:

```
reverse = "tfiwS ❤ eW"
```

Output:

```
"tfiwS ❤ eW"
```

Example 3

Input:

```
var aString = "this string has 29 characters"
```

Expected values:

```
reverse = "sretcarahc 92 sah gnirts siht"
```

Output:

```
"sretcarahc 92 sah gnirts siht"
```

Hint

Convert each character into a string and join them in reverse order.

5.5 Palindrome

Print `true` if `aString` is a palindrome, and `false` otherwise. A palindrome is a string which reads the same backward or forward.

Code

```
let aString = "anutforajaroftuna"

// your code here
```

Example 1

Input:

```
var aString = "anutforajaroftuna"
```

Output:

```
true
```

Example 2

Input:

```
var aString = "Hello"
```

Output:

```
false
```

Example 3

Input:

```
var aString = "HelloolleH"
```

Output:

```
true
```

Hint

How can reversing a string help here ?

5.6 Words

Code

```
var problem = "split this string into words and print them on separate lines"

// your code here
```

Example 1

Input:

```
var problem ="split this string into words and print them on separate lines"
```

Output:

```
split
this
string
into
words
and
print
them
on
separate
lines
```

Hint

Iterate over the characters in the string. Keep track of the longest word you've encountered so far.

5.7 Long word

Code

```
var problem = "find the longest word in the problem description"

// your code here
```

Example 1

Input:

```
var problem = "find the longest word in the problem description"
```

Output:

```
description
```

Hint

Keep track of the longest word you encounter and also keep track of it's length.

5.8 Magic Time!

Use this magic * operator to solve the next challenge:

```
func *(string: String, scalar: Int) -> String {
    let array = Array(repeating: string, count: scalar)
    return array.joined(separator: "")
}

print("cat " * 3 + "dog " * 2)
// cat cat cat dog dog

var newLine = "\n" * 2

print(newLine)
//
//
```

By using **only one** `print()` statement draw a rectangle of size `N x M` out of asterisks.

Code

```
func *(string: String, scalar: Int) -> String {
    let array = Array(repeating: string, count: scalar)
    return array.joined(separator: "")
}

var N = 5
var M = 10

// your code here
```

Example 1

Input:

```
var N = 5
var M = 10
```

Output:

```
**********
**********
**********
**********
**********
```

Example 2

Input:

```
var N = 2
var M = 2
```

Output:

```
**
**
```

Example 3

Input:

```
var N = 5
var M = 2
```

Output:

```
**
**
**
**
**
```

Hint

This problem can be solved by applying the magic operator exactly `2` times.

Solutions

5.1 Full name

The full name can be obatined in multiple ways. Either you use string concatenation (`+`) between `firstName` , " " and `lastName` (`fullName = firstName + " " + lastName`) or you can use string interpolation: `fullName = "\(firstName) \(lastName)"` .

Solution 1

```
var firstName = "Andrei"
var lastName = "Puni"

var fullName = firstName + " " + lastName
```

Solution 2

```
var firstName = "Andrei"
var lastName = "Puni"

var fullName = "\(firstName) \(lastName)"
```

5.2 Sum

This problem is best solved via string interpolation. We just need to format our interpolated string correctly. The sum can be stored in a variable or computed directly in the interpolared string (`"\(a) + \(b) = \(a + b)"`).

```
var a = 14
var b = 23

var sum = a + b

var formattedSum = "\(a) + \(b) = \(sum)"
```

5.3 Replace

We'll want to construct a new string that is initially empty. We'll iterate over all the characters in our string, if the character is equal to "e" we add "*" to our new string otherwise we add the character.

```
var aString = "Replace the letter e with *"

var replacedString = ""

for character in aString.characters {
    var char = "\(character)"
    if char == "e" {
        replacedString = replacedString + "*"
    } else {
        replacedString = replacedString + char
    }
}
```

5.4 Reverse

First we'll create a new string that is initially empty. We iterate all the characters in our string and add them to our reversed string in reverse order i.e. `character + reverse` instead of `reverse + character`.

```
var aString = "this string has 29 characters"
var reverse = ""

for character in aString.characters {
    var asString = "\(character)"
    reverse = asString + reverse
}

print(reverse)
//sretcarahc 92 sah gnirts siht
```

5.5 Palindrome

We'll have to reverse our string and check whether our string is equal to its reverse.

```
let aString = "anutforajaroftuna"

var reverse = ""

for character in aString.characters {
    var char = "\(character)"
    reverse = char + reverse
}

print(aString == reverse)
```

5.6 Words

We'll start with a variable `word` which is initially empty. Next we'll iterate our string, if we encounter a `word` we print the value from the variable `word` and set its contents to empty. If we encounter something

else we add that character to our `word`. We'll also have to print the word one last time outside the loop.

```
var problem = "split this string into words and print them on separate lines"

var word = ""

for character in problem.characters {
    if character == " " {
        print(word)
        word = ""
    } else {
        word += "\(character)"
    }
}

// don't forget the last word
print(word)

// split
// this
// string
// into
// words
// and
// print
// them
// on
// separate
// lines
```

5.7 Long word

We'll have to keep track of the longest word we've encountered so far and its length. The longest word will be initialised to empty and the length to 0. Each time we encounter a new word we check whether its longer than our current `longestWord`. If it is we update the longest word and the length of the longest word. Keep in mind that we'll also have to progressively compute the length of each word.

```
var problem = "find the longest word in the problem description"

// this will help the algorithm see the last word
problem += " "

var word = ""
var length = 0

var max = 0
var longestWord = ""

for character in problem.characters {
    if character == " " {
        if length > max {
            max = length
            longestWord = word
        }
        word = ""
        length = 0
    } else {
```

```
            word += "\(character)"
            length += 1
        }
    }

    print(longestWord)
```

5.8 Magic Time!

First we create the line we want to print via the `*` operator this can be done via `line = "*" * M + "\n"`. Next we create a rectangle by applying the `*` operator on our `line` variable (`line * N`).

```
func *(string: String, scalar: Int) -> String {
    let array = Array(repeating: string, count: scalar)
    return array.joined(separator: "")
}

var newLine = "\n"

var N = 5
var M = 10

var line = "*" * M
line += newLine

var rectangle: String = line * N

print(rectangle)
// **********
// **********
// **********
// **********
// **********
```

6. Arrays

Introduction

Often when you're dealing with data you don't just have a fixed amount of elements. Take for example a program where you compute the average of multiple grades in a class:

```
var grade1 = 4
var grade2 = 3

var average = Double(grade1 + grade2) / 2.0
print("Average grade: \(average)")
```

What if we wanted the program to also work when we have 3 grades?
We'd have to change our program to work with 3 grades.

```
var grade1 = 4
var grade2 = 3
var grade3 = 5

var average = Double(grade1 + grade2 + grade3) / 3.0
print("Average grade: \(average)")
```

After doing this it will no longer work with 2 grades. What if we wanted our program to work with any number of grades between 1 grade and 10 grades.

It's not practical to write a separate program for each case. We would like to have something like a list of grades. This list would contain any number of grades. This is where arrays come in.

What is an array?

An array is an ordered collection that stores multiple values of the same type. That means that an array of `Int` can only store `Int` values. And you can only insert `Int` values in it.

Declaring Arrays

To declare an array you can use the square brackets syntax(`[Type]`).

```
var arrayOfInts: [Int]
```

You can initialize an array with an array literal. An array literal is a list of values, separated by commas, surrounded by a pair of square brackets:

`[ value , value , ... ]`

```
var arrayOfInts: [Int] = [1, 2, 3]
var arrayOfStrings: [String] = ["We", "❤", "Swift"]
```

` Keep in mind that you can create empty arrays if you don't write any values.

```
var emptyArray: [Int] = []
```

Getting values

To get all the values from an array you can use the `for-in` syntax. This is called iterating through an array.

```
var listOfNumbers = [1, 2, 3, 10, 100] // an array of numbers
var listOfNames = ["Andrei", "Silviu", "Claudiu"] // an array of strings

for number in listOfNumbers {
    print(number)
}
// 1
// 2
// 3
// 10
// 100

for name in listOfNames {
    print("Hello " + name + "!")
}
// Hello Andrei!
// Hello Silviu!
// Hello Claudiu!
```

To get the number of elements in an array you can use the `count` property.

```
var listOfNumbers = [1, 2, 3, 10, 100] // an array of numbers

print(listOfNumbers.count) // 5
```

You can access specific elements from an array using the subscript syntax. To do this pass the index of the value you want to retrieve within square brackets immediately after the name of the array. Also you

can get a subsequence from the array if you pass a range instead of an index.
Element in an array are indexed from 0 to the number of elements minus one. So an array with 3 elements will have elements at index 0, 1 and 2.

```
var listOfNumbers = [1, 2, 3, 10, 100] // an array of numbers

listOfNumbers[0] // 1
listOfNumbers[1] // 2
listOfNumbers[2] // 3
listOfNumbers[3] // 10
listOfNumbers[4] // 100
//listOfNumbers[5]// this gives an error uncomment this line to see it

listOfNumbers[1...2] // [2, 3] this is a subsequence of the original array
```

Adding values

You can add elements to the end of an array using the `append` method.

```
// create a empty array of integers
var numbers: [Int] = []

for i in 1...5 {
    numbers.append(i)
    print(numbers)
    // [1]
    // [1, 2]
    // [1, 2, 3]
    // [1, 2, 3, 4]
    // [1, 2, 3, 4, 5]
}

print(numbers)
// [1, 2, 3, 4, 5]
```

To insert an item into the array at a specified index, call the array's `insert(at:)` method.

```
var numbers: [Int] = [1, 2, 3]

numbers.insert(0, at: 0) // numbers will be [0, 1, 2, 3]
numbers.insert(9, at: 1) // numbers will be [0, 9, 1, 2, 3]
```

You can also append another array using the `+=` operator.

```
var numbers: [Int] = [1, 2, 3]

numbers += [4, 5, 6] // numbers will be [1, 2, 3, 4, 5, 6]

// or just one value
```

```
numbers += [7] // numbers will be [1, 2, 3, 4, 5, 6, 7]
```

Removing Values

To remove an item from a specific index call the `remove(at:)` method.

```
var numbers: [Int] = [1, 2, 3]

numbers.remove(at: 0) // numbers will be [2, 3]
```

Changing values

To change a value use the assignment operator (`=`) after the subscript syntax.

```
var numbers: [Int] = [1, 2, 3]

numbers[0] = 7 // numbers will be [7, 2, 3]
numbers[1] = 5 // numbers will be [7, 5, 3]
numbers[2] = 4 // numbers will be [7, 5, 4]
```

Or you could replace a subsequence of values using range subscripting.

```
var numbers: [Int] = [1, 2, 3, 4, 5, 6]

numbers[2...4] = [0, 0] // numbers will now be [1, 2, 0, 0, 6].
```

Keep in mind that you don't need to replace a sequence with another sequence with the same number of elements. In the example above numbers had 6 elements and after the replacement of the subsequence `2...4` (`[3, 4, 5]`) it had 5.

Type Inference

Thanks to Swift's type inference, you don't have to declare the type of an array if you initialize it with something other than an empty array literal(`[]`).

```
// arrayOfNumbers will be of type [Int]
var arrayOfNumbers = [1, 2, 3]

// arrayOfStrings will be of type [String]
var arrayOfStrings = ["We", "❤", "Swift"]

// arrayOfBools will be of type [Bool]
var arrayOfBools = [true, false, true, true, false]
```

```
// this is the proper way of declaring a empty array of Int - [Int]
var emptyArrayOfInts: [Int] = []

// this will infer into [Int] because the right hand side of the
// assignment has a known type [Int]
var anotherEmptyArray = emptyArrayOfInts
```

Copy Behavior

Swift's Array types are implemented as structures. This means that arrays are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

```
var numbers = [1, 2, 3]
var otherNumbers = numbers // this will create a copy of numbers

// this will append 4 to otherNumbers but not to numbers
otherNumbers.append(4)

// numbers = [1, 2, 3]
// otherNumbers = [1, 2, 3, 4]
```

Mutability

If you create an array and assign it to a variable, the collection that is created will be mutable. This means that you can change (or mutate) the collection after it is created. Changes can be done by adding, removing, or changing items in the collection. Conversely, if you assign an array to a constant, that array is immutable, and its size and contents cannot be changed. In other words if you want to be able to change an array declare it using the `var` keyword, and if you don't want to be able to change it use the `let` keyword.

```
var numbers = [1, 2, 3, 4, 5, 6]

numbers.append(7) // [1, 2, 3, 4, 5, 6, 7]
numbers.remove(at: 0) // [2, 3, 4, 5, 6, 7]

let strings = ["We", "❤", "Swift"]

// the next lines will not compile!
strings.append("!") // this will give an error because strings is immutable
strings.remove(at: 0) // this will give a similar error
```

Exercises

6.1 Max

Print the maximum value from `listOfNumbers`.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
```

Output:

```
100
```

Example 2

Input:

```
var listOfNumbers = [10, 12, 33, 11, 1]
```

Output:

```
33
```

Hint

Assume that the first element of the array is also the largest.

6.2 Odd

Print all the odd numbers from `listOfNumbers`.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
```

Output:

```
1
3
```

Example 2

Input:

```
var listOfNumbers = [10, 12, 33, 11, 1]
```

Output:

```
33
11
1
```

6.3 Sum

Print the sum of all the numbers from `listOfNumbers`.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
```

Output:

```
116
```

Example 2

Input:

```
var listOfNumbers = [10, 12, 33, 11, 1]
```

Output:

```
67
```

Hint

Store the sum in a variable. Keep increasing it.

6.4 Odd Index

Print all the numbers from `listOfNumbers` that are located at odd indexes.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
```

Output:

```
2
10
```

Example 2

Input:

```
var listOfNumbers = [1, 10, 12, 33, 11, 1]
```

Output:

```
10
33
1
```

Hint

Use a while loop and the index subscripts.

6.5 Going back

Print the numbers from `listOfNumbers` in reverse order on separate lines.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
```

Output:

```
100
10
3
2
1
```

Example 2

Input:

```
var listOfNumbers = [12, 33, 11, 1]
```

Output:

```
1
11
33
22
```

Hint

Use a loop that counts from the last index down to the first one.

6.6 Reverse

Reverse the order of the elements in `listOfNumbers` without creating any additional arrays.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3]
```

Expected value:

```
listOfNumbers = [3, 2, 1]
```

Example 2

Input:

```
var listOfNumbers = [12, 33, 11, 1]
```

Expected value:

```
listOfNumbers = [1, 11, 33, 12]
```

Hint 1

Use 2 indices.

Hint 2

At each step advance with the indices one step closer to the middle of the `array`.

6.7 Sorting

Sort the values in `listOfNumbers` in descending order.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]

// your code here
```

6.8 Search

Find out if `x` appears in `listOfNumbers`. Print `yes` if true and `no` otherwise.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]
var x = 10

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var x = 3
```

Output:

```
yes
```

Example 2

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var x = 5
```

Output:

```
no
```

Hint

First assume that the element does not appear in the array. Store that state in a boolean variable.

6.9 Intersection

Print all the elements from `otherNumbers` that appear in `listOfNumbers`. Don't print anything if `listOfNumbers` and `otherNumbers` have no common elements.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]
var otherNumbers = [1, 2, 3, 4, 5, 6]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var otherNumbers = [1, 2, 3, 4, 5, 6]
```

Output:

```
1
2
3
```

Example 2

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var otherNumbers = [5, 2, 3, 10, 13]
```

Output:

```
2
3
10
```

Example 3

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var otherNumbers = [5, 6]
```

Output:

Hint

Use an approach similar to the `Search` problem for each element from `otherNumbers`.

6.10 Divisors

Print all the numbers from `listOfNumbers` that are divisible by at least one number from `divisors`.

Code

```
var listOfNumbers = [1, 2, 3, 10, 100]
var divisors = [2, 5]

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var divisors = [2, 5]
```

Output:

```
2
10
100
```

Example 2

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var divisors = [5, 3]
```

```
```

Output:

```
3
10
100
```

Example 3

Input:

```
var listOfNumbers = [1, 2, 3, 10, 100]
var divisors = [7, 9, 13]
```

Output:

```
```

Hint

Try solving the problem for the case when `listOfNumbers` contains a single element.

6.11 Greatest divisor of all

Find and print the greatest common divisor of all the numbers in `numbers`. A common divisor of a list of numbers is a number that divides all of them.

Code

```
var numbers = [12, 36, 720, 18]

// your code here
```

Example 1

Input:

```
var numbers = [12, 36, 720, 18]
```

Output:

```
6
```

Example 2

Input:

```
var numbers = [3, 12, 36, 18, 7]
```

Output:

```
1
```

Hint

Use an approach similar to the case with only 2 numbers.

6.12 Fibonacci

Generate the first `N` numbers in the fibonacci sequence. Store them in an array named `fibonacci` and print them one on each line.

Approach 1

Use append to add the next numbers

```
var N = 30
var fibonacci = [1, 1]

// your code here
```

Approach 2

Create an array with ones and compute all the numbers

```
let N = 30
var fib = [Int](repeating: 1, count: N)
```

```
// your code here
```

Example 1

Input:

```
var N = 6
```

Expected value:

```
fibonacci = [1, 1, 2, 3, 5, 8]
```

Example 2

Input:

```
var N = 9
```

Expected value:

```
fibonacci = [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

6.13 Divisors

Given a `number` find and store all it's divisors in an array called `divisors`, then print the divisors in ascending order on separate lines.

Code

```
var number = 60
var divisors: [Int] = []

// your code here
```

Example 1

Input:

```
var number = 6
```

Expected value:

```
divisors = [1, 2, 3, 6]
```

Output:

```
1
2
3
6
```

Example 2

Input:

```
var number = 30
```

Expected value:

```
divisors = [1, 2, 3, 5, 6, 10, 15, 30]
```

Output:

```
1
2
3
5
6
10
15
30
```

Hint

Any value between `1` and `number` can be a divisor of `number`.

6.14 Digits

Find and store the digits of `number` from left to right in an array `digits`, and then print the digits on separate lines.

Code

```
var number = 12345
var digits: [Int] = []

// your code here
```

Example 1

Input:

```
var number = 12345
```

Expected value:

```
digits = [1, 2, 3, 4, 5]
```

Output:

```
1
2
3
4
5
```

Example 2

Input:

```
var number = 232121
```

Expected value:

```
digits = [2, 3, 2, 1, 2, 1]
```

Output:

```
2
3
2
1
2
1
```

Hint

Store the digits from right to left if you find it easier. The digits from left to right are the reversed array.

6.15 Unique

Create a list `unique` with all the unique numbers from `listOfNumbers`, and then print the numbers on separate lines.

Code

```
var listOfNumbers = [1, 2, 3, 1, 2, 10, 100]

var unique: [Int] = []

// your code here
```

Example 1

Input:

```
var listOfNumbers = [1, 2, 3, 1, 2, 10, 100]
```

Expected value:

```
unique = [1, 2, 3, 10, 100]
```

Output:

```
1
2
3
10
100
```

Example 2

Input:

```
var listOfNumbers = [2, 3, 1, 1, 1, 2, 2, 2, 10]
```

Expected value:

```
unique = [2, 3, 1, 10]
```

Output:

```
2
3
1
10
```

Solutions

6.1 Max

To get the maximum number from a list of numbers we are going to take each number and remember the biggest so far in temporary variable. After we look ar each of the numbers that variable will have the bigest one of them.

```
var listOfNumbers = [1, 2, 3, 10, 100]

var maxVal = listOfNumbers[0]

for number in listOfNumbers {
    if maxVal < number {
        maxVal = number
    }
}

print(maxVal)
```

6.2 Odd

We'll iterate through our array of numbers and print only the numbers that are odd.

```
var listOfNumbers = [1, 2, 3, 10, 100]

for number in listOfNumbers {
    if number % 2 != 0 {
        print(number)
    }
}
```

6.3 Sum

To calculate the sum of all the number we are going to use a variable initialized with 0 and then add each number to that variable.

```
var listOfNumbers = [1, 2, 3, 10, 100]

var sum = 0

for number in listOfNumbers {
    sum += number
}

print(sum)
```

6.4 Odd Index

We'll want to iterate through our array with an index. We'll initialize the index to 1 and at each step we'll increase the index by 2 this will result in printing only the elements at odd indices in the array.

Solution 1

```
var listOfNumbers = [1, 2, 3, 10, 100]

var i = 1

while i < listOfNumbers.count {
    print(listOfNumbers[i])
    i += 2
}
// 2
// 10
```

Solution 2

```
var listOfNumbers = [1, 2, 3, 10, 100]

for var i = 1; i < listOfNumbers.count; i += 2 {
    print(listOfNumbers[i])
}
```

6.5 Going back

We'll iterate through our array with a loop that counts backwards. Keep in mind that the last element of the array is at index `listNumbers.count - 1` and the first element of the array is at index `0`.

Solution 1

```
var listOfNumbers = [1, 2, 3, 10, 100, 2]

var i = listOfNumbers.count - 1

while i >= 0 {
    print(listOfNumbers[i])
    i -= 1
}
```

Solution 2

```
var listOfNumbers = [1, 2, 3, 10, 100, 2]

for i in 1...listOfNumbers.count {
    print(listOfNumbers[listOfNumbers.count - i])
}
```

Solution 3

```
var listOfNumbers = [1, 2, 3, 10, 100, 2]

for var i = listOfNumbers.count - 1; i >= 0; --i {
    print(listOfNumbers[i])
}
```

6.6 Reverse

We'll iterate the array using 2 indices. One starting at the end of the array. One starting at the beginning of the array. At each iteration we'll swap the elements at these indices.

```
var listOfNumbers = [1, 2, 3, 10, 100]

var firstIndex = 0
var lastIndex = listOfNumbers.count - 1

while firstIndex < lastIndex {
    // swap
    var tmp = listOfNumbers[firstIndex]
    listOfNumbers[firstIndex] = listOfNumbers[lastIndex]
    listOfNumbers[lastIndex] = tmp

    // go to next pair
    firstIndex += 1
    lastIndex -= 1
}
```

6.7 Sorting

Solution 1: Insertion Sort

```
var listOfNumbers = [1, 2, 3, 10, 100]

var nElements = listOfNumbers.count

for fixedIndex in 0..<nElements {
    for i in fixedIndex+1..<nElements {
        if listOfNumbers[fixedIndex] < listOfNumbers[i] {
            var tmp = listOfNumbers[fixedIndex]
            listOfNumbers[fixedIndex] = listOfNumbers[i]
            listOfNumbers[i] = tmp
```

```
        }
    }
}
```

Solution 2: Bubble Sort

```
var listOfNumbers = [1, 2, 3, 10, 100]

var nElements = listOfNumbers.count

var didSwap = true

while didSwap {
    didSwap = false

    for i in 0..<nElements - 1 {
        if listOfNumbers[i] < listOfNumbers[i+1] {
            var tmp = listOfNumbers[i]
            listOfNumbers[i] = listOfNumbers[i+1]
            listOfNumbers[i+1] = tmp
            didSwap = true
        }
    }
}
```

Solution 3: The Swift way

In real life people don't implement sorting anymore, they just call it.

```
array.sort(by: <) // will sort the array in ascending order
array.sort(by: >) // will sort the array in descending order

// you can also use the sorted method to create a sorted copy
let sortedArray = array.sorted(by: <)
```

```
var listOfNumbers = [3, 2, 100, 10, 1]

listOfNumbers.sort(by: <)

print(listOfNumbers)
// [1, 2, 3, 10, 100]

listOfNumbers.sort(by: >)

print(listOfNumbers)
// [100, 10, 3, 2, 1]
```

So to solve the problem we would write:

```
var listOfNumbers = [3, 2, 100, 10, 1]

listOfNumbers.sort(by: <)
```

6.8 Search

First we'll asume that the element does not appear in the array. We'll store `false` in a boolean variable `xAppears`. Next will iterate through all the element in the array. If one of the elements is equal to the element we're searching for we'll update the `xAppears` variable to true. The printed message will depend on `xAppears`.

```
var listOfNumbers = [1, 2, 3, 10, 100]

var x = 10

var xAppears = false

for number in listOfNumbers {
    if number == x {
        xAppears = true
    }
}

if xAppears {
    print("yes")
} else {
    print("no")
}
```

6.9 Intersection

We will first iterate through the array `otherNumbers`, for each `otherNumber` in `otherNumbers` we'll search for that number in our `numbers`, we print the number if we find it.

```
var listOfNumbers = [1, 2, 3, 10, 100]

var otherNumbers = [1, 2, 3, 4, 5, 6]

for otherNumber in otherNumbers {
    for number in listOfNumbers {
        if number == otherNumber {
            print(number)
            break
        }
    }
}
```

6.10 Divisors

We'll iterate through all the numbers in our `listOfNumbers`, for each number we'll iterate through the `divisors` array and check if the number is divisible by an element from the `divisors` array, if the condition is true we break out of the inner loop.

```
var listOfNumbers = [1, 2, 3, 10, 100]
var divisors = [7, 5]

for number in listOfNumbers {
    for divisor in divisors {
        if number % divisor == 0 {
            print(number)
            break
        }
    }
}
```

6.11 Greatest divisor of all

First we'll find the largest number in our numbers array (`maxDiv`), we know that whatever greatest common divisor we're looking for will be less than the largest number in our array. We keep track of the greatest common divisor in a variable `gcd`. Next we iterate through all possible divisors between `1` and `maxDiv`, for each divisor we check if it divides all numbers if it does we update our `gcd` variable.

```
var numbers = [12, 36, 720, 18]

// find the minimum value in numbers
var maxDiv = numbers[0]

for number in numbers {
    if number < maxDiv {
        maxDiv = number
    }
}

var gcd = 1

// find the biggest number that divides all the numbers
for divisor in 1...maxDiv {
    // we assume that divisor divides all numbers
    var dividesAll = true
    for number in numbers {
        // if we find one that does not divide by divisor
        if number % divisor != 0 {
            // we remeber and stop searching
            dividesAll = false
            break
        }
    }

    // if divisor divides all numbers then it's the biggest one so far
    if dividesAll {
        gcd = divisor
    }
}

print(gcd)
```

6.12 Fibonacci

The fibbonaci sequence is defined as `F0 = 1` `F1 = 1` `Fn = Fn - 1 + Fn - 2` for `N > 1`. We already have the first 2 elements of our sequence in the array `fibonacci`. We'll want to iterate from `2` to `N - 1` and add a new element to the `fibbonaci` array at each step using the above formula. Note that our array is indexed starting from 0.

The second approach is similar to the first one but we don't add the new elements to the array instead we alreadyt have the array initialized to the correct dimension and we just overwrite the value at each index from `2` to `N - 1`.

Solution 1

```
var N = 30

var fibonacci = [1, 1]

for i in 2...N - 1 {
    fibonacci.append(fibonacci[i-1] + fibonacci[i-2])
}

for number in fibonacci {
    print(number)
}
```

Solution 2

```
let N = 30
var fib = [Int](repeating: 1, count: N)

for i in 2..<N {
    fib[i] = fib[i-1] + fib[i-2]
}

for number in fib {
    print(number)
}
```

6.13 Divisors

To get all the divisors of a `number` we'll want to check if it's divisible by any divisor between `1` and `number`. We'll write a `for` loop that checks for this and saves the result in the `divisors` array.

```
var number = 60
var divisors: [Int] = []
```

```
for divisor in 1...number {
    if number % divisor == 0 {
        divisors.append(divisor)
    }
}

for divisor in divisors {
    print(divisor)
}
```

6.14 Digits

We'll use a while loop to get all the digits of a number. In the while loop we'll be getting the digits from right to left, to reverse their order we'll use a trick similar to reversing a string, adding each new digit at the beginning of the array.

```
var number = 12345
var digits: [Int] = []

while number > 0 {
    var digit = number % 10

    digits = [digit] + digits

    number /= 10 // 12345 -> 1234 -> 123 -> 12 -> 1
}

for digit in digits {
    print(digit)
}
```

6.15 Unique

We'll initialize our `unique` array to be empty. Next we iterate through all the numberes in `listOfNumbers` for each number we check if its already in our `unique` array, if it is not we add it to the `unique` array. This will result in an array with all the unique numbers from `listOfNumbers`.

```
var listOfNumbers = [1, 2, 3, 1, 2, 10, 100]

var unique: [Int] = []

for number in listOfNumbers {
    var numberIsNew = true

    for otherNumber in unique {
        if number == otherNumber {
            numberIsNew = false
            break
        }
    }

    if numberIsNew {
        unique.append(number)
```

```
        }
    }

    for number in unique {
        print(number)
    }
```

7. Functions

A function is a chunk of code that performs a specific task. Functions have a name that describes their purpose, that name is used to call the function to perform the task when needed. You can provide data to a function by sending parameters to it, and the function can give data back as result.

Let's take a simple example:

```
func isOdd(number: Int) -> Bool {
    if number % 2 == 1 {
        return true
    } else {
        return false
    }
}

isOdd(number: 1) // true
isOdd(number: 2) // false
isOdd(number: 3) // true
```

In the above example `isOdd` is the name of the function, `number` is the parameter and `Bool` is the return type.

Defining a function

When you define a function, you can optionally define one or more named, typed values that the function takes as input (known as parameters), and/or a type of value that the function will pass back as output when it is done (known as its return type).

The general syntax for a function is:

func `name` (`list of parameters`) -> `return type` {
  `statements`
}

Some functions don't return any values. In that case the syntax doesn't have the arrow(->) and the return type.

func `name` (`list of parameters`) {
  `statements`
}

Functions with no parameters with no return value

```
func sayHello() {
    print("Hello!")
}

sayHello() // Hello!
```

Functions with one parameter with no return value

Parameters are followed by their type.

```
func sayHello(to name: String) {
    print("Hello \(name)!")
}

sayHello(to: "Swift") // Hello Swift!
```

Functions with one parameter and return value

To add a return value to a function write `->` after the list of parameter followed by the type of the result. Functions that return a value must do so using the `return` keyword. When calling return inside a function the code execution will stop at that line - similar to the `break` statement inside a loop.

```
func square(number: Int) -> Int {
    return number * number
}

square(number: 1) // 1
square(number: 2) // 4
square(number: 3) // 9
```

Functions with multiples parameters with no return value

To declare multiple parameters use commas to separate them.

```
func count(from: Int, to: Int) {
    for i in from...to {
        print(i)
    }
}

count(from: 5, to: 10)
// 5
// 6
// 7
// 8
// 9
// 10
```

Notice that all parameters have the name in the function call. That is called the external parameter name. All parameters have an implicit external parameter name, the same as the local parameter name.

Functions with multiples parameters and return value

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}

print(sum(1, 2)) // 3
```

Notice that this time neither parameter appeared in the function call. This is because of the `_` character in front of `a` and `b`. It this case `_` means don't give this parameter an external name. Remember this because you are going to use it in the exercises.

External parameter names

Sometimes it's useful to name your parameters differently when you call a function.

For example:

```
func sayHello(name:String) {
    print("Hello " + name + "!")
}

sayHello(name: "Batman")
// Hello Batman!
```

In this case it would have more sense name the parameter `to` because then the function call would read `sayHello(to: "Batman")`. But then the code in the function would make less sense.

To do this you must define *external parameter names* for them. You can write external parameter names before the local name. All parameters have the external parameter name set to the local one by default. You can change it by writing a different name before it.

```
func sayHello(to name:String) {
    print("Hello " + name + "!")
}

sayHello(to: "Batman")
// Hello Batman!
```

You can make a function ignore the external parameter name by writing `_` in front of the parameter name:

```
func double(_ number: Int) -> Int {
    return number * 2
}
```

External parameter names make your code clear. Don't remove them unless you have to.

Default Parameter Values

You can define a default value for any parameter in a function definition. You do this by following the parameter definition with a `=` sign and the value for that parameter. If a parameter has a default value set you can omit that parameter when calling the function. To keep things clean it's recommended that you write all the parameters with default value at the end of the parameter list.

```
func countdown(from: Int, to: Int = 1) {
    for i in (to...from).reversed() {
        print(i)
    }
}

countdown(from: 3)
// 3
// 2
// 1

countdown(from: 5, to: 3)
// 5
// 4
// 3
```

In-Out Parameters

Function parameters are constant by default, that means that you cannot change the value of a parameter inside a function. Trying to change the value of a parameter will result in a compile error.

If you want the function to change the value of a parameter and you want those changes to persist after the function call, define the parameter as an `inout` parameter.

Keep in mind that you can only pass variables as in-out parameters. You cannot pass a constant or a literal value, because they cannot be changed. You have to write an ampersand (`&`) in front of the variable name when calling the function. That will indicate that the variable can be modified by the function.

```
func double(number: inout Int) {
    number = number * 2
}

var n = 10
```

```
double(number: &n)

print(n) // 20
```

Quick tip

If you want to change the value of a parameter and those changes don't need to be reflected outside the function then you can just declare a variable with the same name:

```
func printNumber(after number: Int) {
    var number = number
    number += 1
    print(number)
}

printNumber(after: 2) // 3
```

Exercises

7.1 Min

Write a function named `min2` that takes two `Int` values, `a` and `b`, and returns the smallest one. Use `_` to ignore the external parameter names for both `a` and `b`.

Code

```
// your code here
```

Example 1

Function call:

```
min2(1,2)
```

Output:

```
1
```

Example 2

Function call:

```
min2(10,5)
```

Output:

```
5
```

Hint

You can have multiple `return` statements in one function.

Twist

Use the `min2` function to write `min3`. A function that takes three numbers and returns the one with the minimum value.

7.2 Last Digit

Write a function that takes an `Int` and returns it's last digit. Name the function `lastDigit`. Use `_` to ignore the external parameter name.

Code

```
// your code here
```

Example 1

Function call:

```
lastDigit(12345)
```

Output:

```
5
```

Example 2

Function call:

```
lastDigit(1000)
```

Output:

```
0
```

Example 3

Function call:

```
lastDigit(123)
```

Output:

```
3
```

Hint

Use the modulo(`%`) operator.

7.3 First Numbers

Write a function named `first` that takes an `Int` named `N` and returns an array with the first `N` numbers starting from `1`. Use `_` to ignore the external parameter name.

Code

```
// your code here
```

Example 1

Function call:

```
first(3)
```

Output:

```
[1, 2, 3]
```

Example 2

Function call:

```
first(1)
```

Output:

```
[1]
```

Example 3

Function call:

```
first(10)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Hint

Use the `append` function on arrays to create the required result.

7.4 Countdown

Write a function named `countdown` that takes a number `N`. The function should print the numbers from `N` to `1` with a one second pause in between and then write `GO!` in the end. To make the computer wait for one second call the `sleep` function from the standard library. The `sleep` function takes one parameter, the number of seconds to sleep.

Using the sleep function

In order to use the `sleep` function you will need to import the Foundation framework.

```
import Foundation
// now you can use the sleep function

sleep(1) //will wait for one second before executing the next line
```

Code

```
import Foundation

// your code here
```

Example

Function call:

```
coutdown(3)
```

Output:

```
3
2
1
GO!
```

7.5 Prime Numbers

Implement the following functions. The `divides` function returns `true` if `a` is divisible by `b` and `false` otherwise. The `countDivisors` function should use the `divides` function to return the number of divisors of `number`. The `isPrime` function should use the `countDivisors` function to determine if `number` is prime.

Examples:

```
divides(3, 2) // false - 3 is not divisible by 2
divides(6, 3) // true - 6 is divisible by 3

countDivisors(2) // 2 - 1 and 2
countDivisors(6) // 4 - 1, 2, 3 and 6
countDivisors(12) // 6 - 1, 2, 3, 4, 6 and 12

isPrime(2) // true
isPrime(3) // true
isPrime(10) // false
isPrime(13) // true
```

Code

```
// your code here
```

Example 1

Function call:

```
isPrime(2)
```

Output:

```
true
```

Example 2

Function call:

```
isPrime(3)
```

Output:

```
true
```

Example 3

Function call:

```
isPrime(10)
```

Output:

```
false
```

Example 4

Function call:

```
isPrime(13)
```

Output:

```
true
```

Hint

The `isPrime` function can be implemented in a single line using the `countDivisors` function.

7.6 First Primes

Using `isPrime` write a function named `printFirstPrimes` that takes a parameter named `count` of type `Int` that prints the first `count` prime numbers.

Code

```
// your code here
```

Example 1

Function call:

```
printFirstPrimes(3)
```

Output:

```
2
3
5
```

Example 2

Function call:

```
printFirstPrimes(10)
```

Output:

```
2
3
5
```

```
7
11
13
17
19
23
29
```

Hint

Use the `isPrime` function from the previous exercise.

7.7 Repeat Print

Implement a function named `repeatPrint` that takes a string `message` and a integer `count` as parameters. The function should print the `message` `count` times and then print a newline.

Code

```
// your code here
```

Example 1

Function call:

```
repeatPrint("+", 10)
```

Output:

```
++++++++++
```

Example 2

Function call:

```
repeatPrint(message: "<->", count: 3)
```

Output:

```
<-><-><->
```

Hint

Don't forget about the newline at the end.

7.8 Reverse

Write a function named `reverse` that takes an array of integers named `numbers` as a parameter. The function should return an array with the numbers from `numbers` in reverse order.

Code

```
// your code here
```

Example 1

Function call:

```
reverse([1, 2, 3])
```

Output:

```
[3, 2, 1]
```

Example 2

Function call:

```
reverse([1, 2, 1, 2, 1, 2])
```

Output:

```
[2, 1, 2, 1, 2, 1]
```

7.9 Sum

Write a function named `sum` that takes an array of integers and returns their sum.

Code

```
// your code here
```

Example 1

Function call:

```
sum([1, 2, 3])
```

Output:

```
6
```

Example 2

Function call:

```
sum([1, 1, 1, 1, 1])
```

Output:

```
5
```

7.10 Parse number

Write a function named `parse(digit:)` that takes a string with one character as parameter. The function should return `-1` if the input is not a digit character and the digit otherwise.

```
parse(digit: "1") // 1
parse(digit: "3") // 3
parse(digit: "a") // -1
```

Using the `parse(digit:)` function you can determine if a string of length one is a digit or not. Implement a function named `isNumber` that takes an arbitrary length string and return `true` if the string contains only digits and `false` otherwise. Note that empty strings should not be considered numbers.

```
isNumber("a") // false
isNumber("1") // true
isNumber("1234567890") // true
isNumber("12345abc") // false
isNumber("") // false
```

Using the `isNumber` and `parse(digit:)` functions, write a function named `parse(number:)` that takes a string and returns it's values as an integer or `-1` if the string does not contain only digits.

```
parse(number: "1")     // 1
parse(number: "54321") // 54321
parse(number: "1337")  // 1337
parse(number: "12cd")  // -1
```

Code

```
// your code here
```

Hint

Use a string of digits `let digits = "0123456789"`.

7.11 Time Difference

Write a function named `timeDifference`. It takes as input four numbers that represent two times in a day and returns the difference in minutes between them. The first two parameters `firstHour` and `firstMinute` represent the hour and minute of the first time. The last two `secondHour` and `secondMinute` represent the hour and minute of the second time. All parameters should have external parameter names with the same name as the local ones.

Code

```
// your code here
```

Example 1

Function call:

```
timeDifference(firstHour: 12, firstMinute: 3, secondHour: 13, secondMinute: 10)
```

Output:

```
67
```

Example 2

Function call:

```
timeDifference(firstHour: 8, firstMinute: 10, secondHour: 17, secondMinute: 30)
```

Output:

```
560
```

Hint

You'll have to handle the case when the difference between minutes is less than `0`.

7.12 Correct Pairs

Write a function named `verify` that takes a string `expression` of open and closed parentheses (`(` , `)`) and returns `true` if they are correctly paired and `false` otherwise.

Code

```
// your code here
```

Example 1

Function call:

```
verify(expression: "()")
```

Output:

```
true
```

Example 2

Function call:

```
verify(expression: "((")
```

Output:

```
false
```

Example 3

Function call:

```
verify(expression: "(())")
```

Output:

```
true
```

Example 4

Function call:

```
verify(expression: "()()")
```

Output:

```
true
```

Example 5

Function call:

```
verify(expression: "(((())")
```

Output:

```
false
```

Example 6

Function call:

```
verify(expression: ")(")
```

Output:

```
false
```

Hint 1

Keep track of how many open parentheses you've encountered and how many closed parentheses.

Hint 2

In a correct pairing the number of closed parentheses you encounter can never be greater than the number of open parentheses.

7.13 Mario

Mario uses energy points to walk and jump. He can jump maximum `maxJump` meters up or down. You have the height of each 1 meter portion of a level in the `heights` array. Determine if Mario can finish the level and how much energy he needs to do it. Mario uses 1 energy point to walk one meter and `2 * jumpHeight` energy points to `jumpHeight` meters. Write a function named `levelCost` that takes `heights` and `maxJump` as parameters and returns `-1` if Mario cannot finish the level or the total energy cost that he would need to finish the level.

In the beginning Mario will be on the first 1 meter section of the level and the `heights` array will always have more than one element. All heights have a value greater or equal to 1.

```
levelCost(heights: [1, 1, 2, 2, 5, 2, 1, 1], maxJump: 3) // 19
// 1 point to walk
// 2 to jump from 1 to 2
// 1 point to walk
// 6 to jump from 2 to 5
// 6 to jump from 5 to 2
// 2 to jump from 2 to 1
// 1 point to walk

levelCost(heights: [1, 1, 3, 1, 1], maxJump: 2) // 10
// 1 point to walk
// 4 to jump from 1 to 3
// 4 to jump from 3 to 1
// 1 point to walk

levelCost(heights: [1, 1, 8, 1], maxJump: 5) // -1
// Mario cannot jump from 1 to 8
```

Code

```
// your code here
```

Hint

Think about how you can compute the energy required for a single step.

7.14 Queue

A queue is a data structure that can perform two operations:

- **push** which takes a value and adds it at the end of the queue
- **pop** which returns the value from the start of the queue and removes it from the queue

Your task is to implement the `push` and `pop` operations. The most simple way to represent a queue is using an array. Here are some example operations.

```
// here we define an empty queue
var queue: [Int] = []

// add 1 in the queue
push(1, &queue) // queue = [1]

// add 2 in the queue
push(2, &queue) // queue = [1, 2]

// pop the first element
```

```
pop(&queue) // 1, queue = [2, 3]

// add 3 in the queue
push(3, &queue) // queue = [2, 3]

// pop the first element
pop(&queue) // 2, queue = [3]

// pop the first element
pop(&queue) // 3, queue = []

// pop the first element
pop(&queue) // returns nil because there are no elements in the queue
// queue = []
```

The `push` function should take two parameters, the `number` and the `queue` as an inout parameter.

The `pop` function should take `queue` as an inout parameter and return the first number from the queue after removing it. If the queue is empty it should return nil - the result type should be an optional integer(`Int?`).

Code

```
// your code here
```

Hint

For the `pop` function you'll have to retrieve the first element in the queue.

7.15 Stack

A stack is a data structure that can perform three operations:

- **push** adds a value on the top of the stack
- **top** returns the value from the top of the stack
- **pop** returns the value from the top of the stack and removes it from there

Your task is to implement the `push` , `top` and `pop` operations. The most simple way to represent a stack is using an array. Here are some example operations.

```
var stack: [Int] = []

push(1, &stack) // stack = [1]

push(2, &stack) // stack = [1, 2]

pop(&stack) // 2, stack = [1]

push(3, &stack) // stack = [1, 3]
```

```
pop(&stack) // 3, stack = [1]

pop(&stack) // 1, stack = []

pop(&stack) // returns nil because there are no elements in the stack
// stack = []
```

`push` takes two parameters, the `number` that will be pushed and the `stack` as an inout parameter.

`top` takes one parameter, the `stack`, and returns the value of the top element or nil if the stack is empty - the result type should be and optional integer(`Int?`)

`pop` takes the `stack` as an inout parameter, and returns the value of the top element after it removes it. If the `stack` is empty it should return nil - the result type should be and optional integer(`Int?`)

Code

```
// your code here
```

Hint

You'll have to get the last element from the stack for the `top` operation.

Solutions

7.1 Min

If `a > b` we return `a` from the function otherwise we return `b`.

```
func min2(_ a: Int, _ b: Int) -> Int {
    if a < b {
        return a
    } else {
        return b
    }
}
```

Twist

```
func min3(a: Int, _ b: Int, _ c: Int) -> Int {
    return min2(min2(a, b), c)
}
```

7.2 Last Digit

We just have to return the last digit from the function. The last digit si given by `number % 10`.

```
func lastDigit(_ number: Int) -> Int {
    return number % 10
}
```

7.3 First Numbers

We'll create a new array in which we'll add `number` N times. At the end of the function we'll return that array.

```
func first(_ N: Int) -> [Int] {
    var numbers:[Int] = []

    for number in 1...N {
        numbers.append(number)
    }

    return numbers
}
```

7.4 Countdown

We'll use a `while` loop that counts down from `N` to 1. At each step we'll print the `number` making sure to also call the `sleep` function. After the loop we'll print "GO!"

```
import Foundation

func countdown(_ N: Int) {
    var i = N

    while i > 0 {
        print(i)

        sleep(1)

        i -= 1
    }

    print("GO!")
}
```

7.5 Prime Numbers

The `divides` function returns true if `a % b == 0` and false otherwise, this is equivalent to returning the value of the condition `a % b == 0`

The `countDivisors` function keeps track of the number of encountered divisors in a `cnt` variable. We loop from `1` to `number` and at each step increment `cnt` if `divides(n,i)` returns true.

The `isPrime` function simply returns `true` when the `number` of divisors of the number is equal to `2` using a call to the `countDivisors` function.

```
func divides(_ a: Int, _ b: Int) -> Bool {
    return a % b == 0
}

func countDivisors(_ number: Int) -> Int {
    var cnt = 0
    for i in 1...number {
        if divides(number, i) {
            cnt += 1
        }
    }
    return cnt
}

func isPrime(_ number: Int) -> Bool {
    return countDivisors(number) == 2
}
```

7.6 First Primes

We'll make use of the `isPrime` function from the previous exercise. We'll keep track of how many numbers we already printed in the `printed` variable. We'll also use a variable `i` to keep track of the number we're currently testing. We'll execute a `while` loop until `printed` becomes equal to `count`. At each step we'll check if `i` is prime, we'll print it and increment `printed` if that's the case, otherwise we just increment `i`.

```
func printFirstPrimes(_ count: Int) {
    var i = 2
    var printed = 0
    while printed < count {
        if isPrime(i) {
            print(i)
            ++printed
        }
        ++i
    }
}
```

7.7 Repeat Print

We have to print the `message` `count` times using a `for` loop. After we're done printing we run a final `print()` statement to add a newline.

```
func repeatPrint(message: String, count: Int) {
    for i in 1...count {
        print(message, terminator: "")
    }
    print("")
}
```

7.8 Reverse

The solution to this problem is very similar to `Problem 6.14` the difference is that we implement it as a function.

```
func reverse(_ numbers: [Int]) -> [Int] {
    var reversed: [Int] = []

    for number in numbers {
        reversed.insert(number, at: 0)
    }

    return reversed
}
```

7.9 Sum

We'll keep track of the current sum in `sum` variable initialized to 0. We iterate through our array adding each `number` to our `sum`. Finally we return the `sum`.

```
func sum(_ numbers: [Int]) -> Int {
    var sum = 0

    for number in numbers {
        sum += number
    }

    return sum
}
```

7.10 Parse number

Solution: parse(digit:)

First we check if the given string is a number, if it is not we return `-1`. Next we initialize our result to `0`. For each character in the given string we multiply the result by 10, shifting all digits with 1 position to the left and we add the result of `parseDigit` for the current digit.

```
func parse(digit: String) -> Int {
    let digits = "0123456789"

    var result = 0

    for character in digits.characters {
        var d = "\(character)"

        if d == digit {
            return result
        }

        result += 1
    }

    return -1
}
```

Solution: isNumber

If the string we're given is empty we return `false` otherwise we iterate through all the characters in our string, if any of these characters returns `-1` from our `parseDigit` function we return `false`. If none of them return `-1` from parseDigit it means that all characters in our string are digits and we return `true`

```
func isNumber(_ string: String) -> Bool {
    if string.characters.count == 0 {
```

```
            return false
        }

        for character in string.characters {
            if parse(digit: "\(character)") == -1 {
                return false
            }
        }

        return true
    }
```

Solution: parse(number:)

First we check if the given string is a number, if it is not we return `-1`. Next we initialize our result to `0`. For each character in the given string we multiply the result by 10, shifting all digits with 1 position to the left and we add the result of `parse(digit:)` for the current digit.

```
func parse(number: String) -> Int {
    if isNumber(number) != true {
        return -1
    }

    var result = 0
    for character in number.characters {
        var digit = "\(character)"

        result = result * 10 + parse(digit: digit)
    }

    return result
}
```

7.11 Time Difference

To compute the time difference we have to determine the hour difference and the minute difference between them. If they are in the same hour the minute difference will be negative - in that case we have to add 60 minutes to the time difference and take an hour from the hour one. The final timeDifference in minutes will be `hourDifference * 60 + minuteDifference`.

```
func timeDifference(firstHour: Int,
                    firstMinute: Int,
                    secondHour: Int,
                    secondMinute: Int) -> Int {
    var hourDifference = secondHour - firstHour
    var minuteDifference = secondMinute - firstMinute

    if minuteDifference < 0 {
        hourDifference -= 1
        minuteDifference += 60
    }

    return hourDifference * 60 + minuteDifference
```

```
}
```

7.12 Correct Pairs

We are going to use two counters - `open` and `closed` - that count the number of open and closed parentheses so far. The expression will be valid if at any point the number of open parentheses is greater than or equal to the number of closed ones and in the end the number of closed parentheses is equal to the number of open ones.

```
func verify(expression: String) -> Bool {
    var open = 0
    var closed = 0
    for char in expression.characters {
        var character = "\(char)"
        if character == "(" {
            open += 1
        } else {
            closed += 1
            if closed > open {
                return false
            }
        }
    }
    return open == closed
}
```

7.13 Mario

We are going to go ever each portion of the level remembering the height of the previous one in `lastHeight`. This will help us determine the height difference at each step. Then we can determine if this is a jump (difference > 0) or just a step (difference = 0). If Mario has to jump we have to check if he can jump as high/low - otherwise he cannot complete the level.

```
func levelCost(heights: [Int], maxJump: Int) -> Int {
    var totalEnergy = 0
    var lastHeight = 0

    for height in heights {
        if lastHeight == 0 {
            lastHeight = height
        } else {
            var jumpHeight = lastHeight - height
            if jumpHeight < 0 {
                jumpHeight = -jumpHeight
            }

            if jumpHeight > maxJump {
                return -1
            }

            if jumpHeight == 0 {
                totalEnergy += 1
```

```
            } else {
                totalEnergy += 2 * jumpHeight
            }

            lastHeight = height
        }
    }

    return totalEnergy
}
```

7.14 Queue

```
func push(_ number: Int, _ queue: inout [Int]) {
    queue.append(number)
}

func pop(_ queue: inout [Int]) -> Int? {
    var result = queue.first

    if queue.count > 0 {
        queue.remove(at: 0)
    }

    return result
}
```

7.15 Stack

```
func push(_ number: Int, _ stack: inout [Int]) {
    stack.append(number)
}

func top(_ stack: [Int]) -> Int? {
    if stack.count == 0 {
        return nil
    }
    return stack[stack.count - 1]
}

func pop(_ stack: inout [Int]) -> Int? {
    var result = top(stack)

    if stack.count > 0 {
        stack.remove(at: stack.count - 1)
    }

    return result
}
```

8. Recursion

Recursion is the process of repeating items in a self-similar way.

> picture by Pavlos Mavridis

The same way you can call a function inside of other functions, you can call a function inside of itself. A function that calls itself is called a recursive function. Recursion is important because you can solve some problems by solving similar sub-problems. Recursive solutions usually have less code and are more elegant than their iterative equivalents if the problem you solve is recursive in nature.

Let's take a simple example. To print all the numbers from `1` to `N`, the first thing we have to do is print all the numbers from `1` to `N-1` and then print `N`.

```
func printFirstNumbers(_ N: Int) {
    if N > 1 {
        printFirstNumbers(N - 1)
    }
    print(N)
}

printFirstNumbers(3)
// 1
// 2
// 3
```

Okay ... the example from above works ... but what really happens?

To understand what happens we can take a look at a modified version of the `printFirstNumbers` function. This version will print all the steps it takes.

```
func printFirstNumbers(_ N: Int) {
    print("start printFirstNumbers(\(N))")

    if N > 1 {
        print("printFirstNumbers(\(N)) calls printFirstNumbers(\(N-1))")

        printFirstNumbers(N - 1)
    }

    print("printFirstNumbers(\(N)) will print \(N)")

    print("end printFirstNumbers(\(N))")
}

printFirstNumbers(3)
// start printFirstNumbers(3)
// printFirstNumbers(3) calls printFirstNumbers(2)
// start printFirstNumbers(2)
// printFirstNumbers(2) calls printFirstNumbers(1)
// start printFirstNumbers(1)
// printFirstNumbers(1) will print 1
// end printFirstNumbers(1)
// printFirstNumbers(2) will print 2
// end printFirstNumbers(2)
// printFirstNumbers(3) will print 3
// end printFirstNumbers(3)
```

The computer knows where to continue the execution of `printFirstNumbers(2)` after `printFirstNumbers(1)` finishes by using a data structure known as a call stack. The call stack keeps information about the currently active functions. When `printFirstNumbers(1)` starts executing `printFirstNumbers(3)` and `printFirstNumbers(2)` are still active and they need to resume control right after the if statement.

Notice that `printFirstNumbers(1)` did not call `printFirstNumbers` again. That's known as a base case. You need to have at least one base case inside a recursive function in order to prevent infinite calls - or what is known as stack overflow.

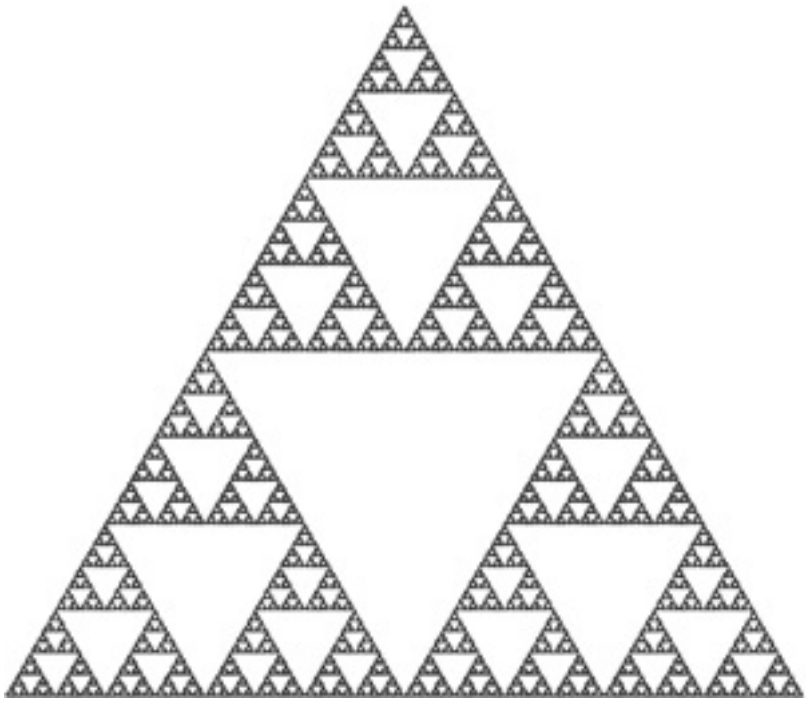
Let's take another example. This time instead of printing the numbers from `1` to `N` let's do it from `N` to `1`.
To count from `N` to `1` all we need to do is print `N` then print all the numbers from `N-1` to `1`.

```
func printFrom(_ N: Int) {
    print(N)
    if N > 1 {
        printFrom(N - 1)
```

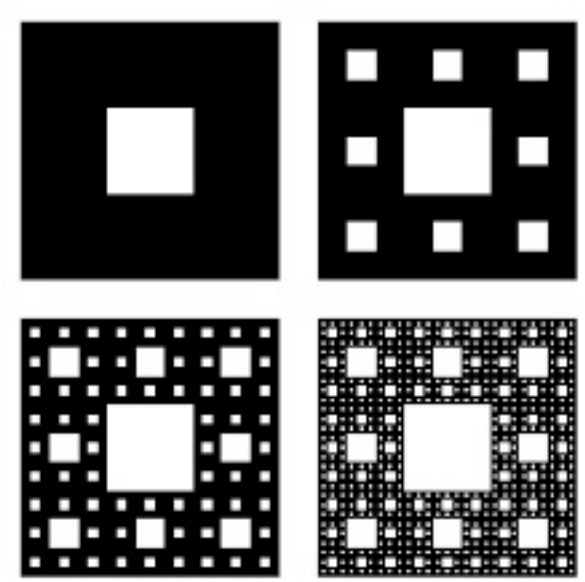
```
    }
}

printFrom(5)
// 5
// 4
// 3
// 2
// 1
```

You can find another example of recursion if you google recursion. The results page will ask you "Did you mean: recursion" which will take you to the same page...

Here are some visual examples of recursion. The Sierpinski triangle and carpet.

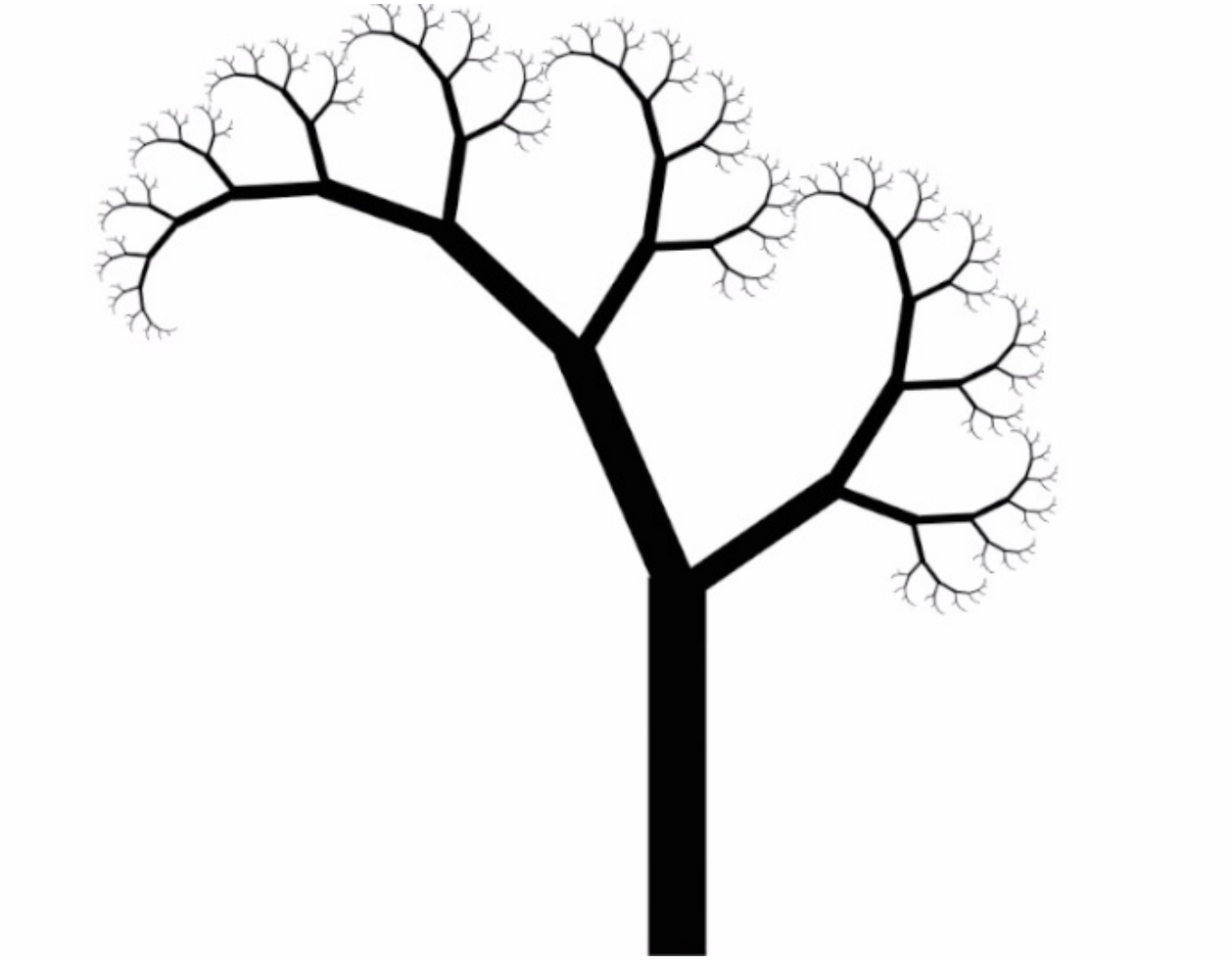


> Sierpinski triangle



> Sierpinski carpet

And here you can find a recursive drawing editor made by Toby Schachman. It's super easy to use - and a lot of fun. If you didn't understand what recursion is all about I highly encourage you to take a few minutes to play with it.



> An example image that could be generated using recursive drawing

Things to remember about recursion:

- you can call a function inside of itself
- you always have at least one base case in order to prevent the function calling itself infinite times

Exercises

8.1 Fibonacci

Implement a recursive function named `fibonacci` that takes a number `N` and returns the `N`-th fibonacci number. The first two fibonacci numbers are `1` and the rest are the sum of the previous two.

Code

```
// your code here
```

Example 1

Function call:

```
fibonacci(3)
```

Function output:

```
2
```

Example 2

Function call:

```
fibonacci(4)
```

Function output:

```
3
```

Example 3

Function call:

```
fibonacci(5)
```

Function output:

```
5
```

Example 4

Function call:

```
fibonacci(6)
```

Function output:

```
8
```

Hint

Remember that if `N > 2` , `fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2)`

8.2 Factorial

The factorial of a non-negative integer `N` , denoted `N!` , is the product of all the positive integer less than or equal to `N` . The value of `0!` is defined as `1` .

```
1! = 1
2! = 1 * 2 = 2
3! = 1 * 2 * 3 = 6
...
7! = 1 * 2 ... * 7 = 5040
```

Write a recursive function named `factorial` that takes an integer `N` and returns it's factorial.

Code

```
// your code here
```

Example 1

Function call:

```
factorial(3)
```

Function output:

```
6
```

Example 2

Function call:

```
factorial(5)
```

Function output:

```
120
```

Example 3

Function call:

```
factorial(10)
```

Function output:

```
3628800
```

Hint

`N! = N * (N - 1)!`

8.3 Digits

Implement a recursive function named `digits` that takes a positive integer `number` and return an array containing it's digits in order.

Code

```
// your code here
```

Example 1

Function call:

```
digits(123)
```

Function output:

```
[1, 2, 3]
```

Example 2

Function call:

```
digits(0)
```

Function output:

```
[0]
```

Example 3

Function call:

```
digits(54321)
```

Function output:

```
[5, 4, 3, 2, 1]
```

Hint

To get the digits of a number you need to get the digits of the number without its last digit (divide by `10`). And then add the last digit to that result.

8.4 Power

Write a recursive function `pow` that takes two numbers `x` and `y` as input and returns `x` to the power `y`.

Code

```
// your code here
```

Example 1

Function call:

```
pow(2, 10)
```

Function output:

```
1024
```

Example 2

Function call:

```
pow(3, 3)
```

Function output:

```
27
```

Example 3

Function call:

```
pow(100, 1)
```

Function output:

```
100
```

Example 4

Function call:

```
pow(10, 0)
```

Function output:

```
1
```

Hint

A simple recursive formula we can use is `x^y = x * x ^ (y - 1)`

8.5 Euclid

Implement the Euclidian algorithm for getting the greatest common divisor of two numbers by using repeated subtractions. The algorithm starts with two numbers and subtracts the smallest one from the other one until one of them becomes zero, the other one is the greatest common divisor of the original number. The `gcd` function takes two numbers as input and returns their greatest common divisor. Implement the algorithm as a recursive function.

Algorithm example:

```
10 2
8 2
6 2
4 2
2 2
2 0 // 2 is the greatest common divisor of 10 and 2

9 6
```

```
3 6
3 3
3 0 // 3 is the greatest common divisor of 9 and 6

35 49
35 14
21 14
7  14
7  7
7  0 // 7 is the greatest common divisor of 35 and 49
```

Code

```
// your code here
```

Example 1

Function call:

```
gcd(2, 10)
```

Function output:

```
2
```

Example 2

Function call:

```
gcd(9, 6)
```

Function output:

```
3
```

Example 3

Function call:

```
gcd(30, 75)
```

Function output:

```
15
```

8.6 Binary Search

Searching a sorted collection is a common task. For example finding a word in a dictionary, or a number in a phone book.

Binary search is one of the fundamental algorithms in computer science. In its most basic form, binary search finds the position of an element, known as the search key, in a sorted array. It does this by repeatedly halving the search interval. Initially the search interval ranges from the first index of the array to the last index. In each step the algorithm compares the middle value from the search interval with the search key. If the middle value is less than the key, then all the values from the first half of the array are lower than the key, that means that the search key cannot be located in the first half, the search will continue in the second half of the array. The same logic will apply if the middle element is greater than the key. If the middle value is equal to the search key then the key has been found.

Let's take a few examples to understand the algorithm, `left` if the first index of the search interval and `right` is the last one:

```
numbers = [1, 2, 4, 5, 7, 8, 9, 12] // 8 elements

------------------------------------------------------------------
key = 4

// Step 1 (left = 0, right = 7)
// the middle element is 5 which is greater than 4
// the search will continue in the first half of the search interval

// Step 2 (left = 0, right = 3)
// the middle element is 2 which is less than 4
// the search will continue in the second half of the search interval

// Step 3 (left = 2, right = 3)
// the middle element is 4 - we found the key!

------------------------------------------------------------------
key =  12

// Step 1 (left = 0, right = 7)
// the middle element is 5 which is less than 12
// the search will continue in the second half of the search interval

// Step 2 (left = 4, right = 7)
// the middle element is 8 which is less than 12
// the search will continue in the second half of the search interval
```

```
// Step 3 (left = 6, right = 7)
// the middle element is 9 which is less than 12
// the search will continue in the second half of the search interval

// Step 4 (left = 7, right = 7)
// the search interval has only one element which is equal to the key

----------------------------------------------------------------------
key = 3

// Step 1 (left = 0, right = 7)
// the middle element is 5 which is greater than 3
// the search will continue in the first half of the search interval

// Step 2 (left = 0, right = 3)
// the middle element is 2 which is less than 3
// the search will continue in the second half of the search interval

// Step 3 (left = 2, right = 3)
// the middle element is 4 which is greater than 3
// the search will continue in the first half of the search interval

// Step 4 (left = 2, right = 2)
// the search interval has only one element which is not equal to the key
// the key could not be found in the array
```

Implement the binary search function using recursion.

Code

```
func binarySearch(_ key: Int,
                  _ numbers: [Int],
                  left: Int = 0,
                  right: Int = -1) -> Bool {
    var right = right
    if right == -1 {
        right = numbers.count - 1
    }

    // your code here

    return false
}
```

Example 1

Function call:

```
binarySearch(2, [1, 2, 4, 5 ,7, 9])
```

Function output:

```
true
```

Example 2

Function call:

```
binarySearch(3, [1, 2, 4, 5 ,7, 9])
```

Function output:

```
false
```

Example 3

Function call:

```
binarySearch(6, [1, 2, 4, 5 ,7, 9])
```

Function output:

```
false
```

Hint

To start off consider the base case, we have to return true when `key` is equal to `numbers[mid]`.

8.7 Towers of Hanoi

There are three pegs labeled `A`, `B` and `C`. On the first peg there are `N` stacked disks, each one with a different diameter from biggest (on bottom) to smallest. Move all the disks from the first peg to the second one while respecting the rules:

- only one disk can be moved at a time
- each move consists of taking the upper disk from one of the stacks and placing it on top of another stack
- no disk may be placed on top of a smaller disk

```
N = 5

     *          |          |
    ***         |          |
   *****        |          |
  *******       |          |
 *********      |          |
---------------------------------
```

We provide code that will help you visualize the algorithm. When you want to move a disk from a peg to another call `move(from:to:)`:

```
moveDisk(from: "A", to: "B")
moveDisk(from: "A", to: "C")
moveDisk(from: "B", to: "C")
```

Output:

```
     *          |          |
    ***         |          |
   *****        |          |
  *******       |          |
 *********      |          |
---------------------------------

moves from A to B

     |          |          |
    ***         |          |
   *****        |          |
  *******       |          |
 *********      *          |
---------------------------------

moves from A to C

     |          |          |
     |          |          |
   *****        |          |
  *******       |          |
 *********      *         ***
---------------------------------

moves from B to C

     |          |          |
     |          |          |
   *****        |          |
  *******       |          *
 *********      |         ***
---------------------------------
```

Download this playground to get started.

Code

```
// your code here
```

Example 1

Input:

```
let N = 2
```

Output:

```
  *    |    |
 ***   |    |
---------------

moves from A to C

  |    |    |
 ***   |    *
---------------

moves from A to B

  |    |    |
  |   ***   *
---------------

moves from C to B

  |    *    |
  |   ***   |
---------------
```

Example 2

Input:

```
let N = 3
```

Output:

```
    *      |      |
```

```
  ***     |      |
 *****    |      |
---------------------

moves from A to B

    |     |      |
  ***     |      |
 *****    *      |
---------------------

moves from A to C

    |     |      |
    |     |      |
 *****    *     ***
---------------------

moves from B to C

    |     |      |
    |     |      *
 *****    |     ***
---------------------

moves from A to B

    |     |      |
    |     |      *
    |    *****   ***
---------------------

moves from C to A

    |     |      |
    |     |      |
    *    *****   ***
---------------------

moves from C to B

    |     |      |
    |     ***    |
    *    *****   |
---------------------

moves from A to B

    |     *      |
    |     ***    |
    |    *****   |
---------------------
```

Hint

To move N disks from a peg `A` to peg `B`, you must first move `N-1` disks from peg `A` to peg `C`, then move the last disk from `A` onto `B` and finally move `N-1` disks from `C` to `B`. In other words to solve the Hanoi(N) problem you must first solve the Hanoi(N-1) problem two times. When you only have to move one disk you can just make the move.

Solutions

8.1 Fibonacci

First consider the base case: If `i <= 2` we know that we have to return `1` (the first 2 fibonacci numbers are both `1`). Otherwise we just return `fibonacci(i - 1) + fibonacci(i - 2)`, which the definition of the fibonacci sequence.

```
func fibonacci(_ i: Int) -> Int {
    if i <= 2 {
        return 1
    } else {
        return fibonacci(i - 1) + fibonacci(i - 2)
    }
}
```

8.2 Factorial

Let's consider the case when `N = 1`, the answer is `1` in that case. Otherwise we compute `N!` as `N * (N - 1)!`. In code this is written as `N * factorial(N - 1)`.

```
func factorial(_ N: Int) -> Int {
    if N == 1 {
        return 1
    } else {
        return N * factorial(N - 1)
    }
}
```

8.3 Digits

We can get the last digit of a number by computing it's remainder to 10. We can remove the last digit by dividing the number by 10. If we take the last digit, remove it and repeat until the number becomes 0 we get the digits in reverse order. We can use recursion to reverse it - first we get the digits of the number without the last then we add the last digit.

```
func digits(_ number: Int) -> [Int] {
    if number >= 10 {
        let firstDigits = digits(number / 10)
        let lastDigit = number % 10
        return firstDigits + [lastDigit]
    } else {
        return [number]
    }
}
```

8.4 Power

Solution 1: tail recursion

Using `x^y = x * x^(y-1)`:

```
func pow(_ x: Int, _ y: Int) -> Int {
    if y == 0 {
        return 1
    } else {
        return x * pow(x, y - 1)
    }
}
```

Solution 2: Exponentiation by squaring

Using exponentiation by squaring:

```
func pow(_ x: Int, _ y: Int) -> Int {
    if y == 0 {
        return 1
    } else if y == 1 {
        return x
    } else {
        // compute x^(y/2)
        let xy2 = pow(x, y / 2)
        // if y is even
        if y % 2 == 0 {
            // x^y is x^(y/2) squared
            return xy2 * xy2
        } else {
            // x^y is x^(y/2) squared times x
            return xy2 * xy2 * x
        }
    }
}
```

8.5 Euclid

```
func gcd(_ a: Int, _ b: Int) -> Int {
    if b == 0 {
        return a
    } else {
        if a > b {
            return gcd(a-b, b)
        } else {
            return gcd(a, b-a)
        }
    }
```

```
}
```

8.6 Binary Search

```
func binarySearch(_ key: Int,
                  _ numbers: [Int],
                  left: Int = 0,
                  right: Int = -1) -> Bool {
    var right = right
    if right == -1 {
        right = numbers.count - 1
    }

    if left < right {
        var mid = (left + right) / 2
        if key < numbers[mid] {
            return binarySearch(key, numbers, left: left, right: mid)
        } else if key > numbers[mid] {
            return binarySearch(key, numbers, left: mid + 1, right: right)
        } else {
            return true
        }
    } else {
        return numbers[left] == key
    }
}
```

8.7 Towers of Hanoi

To move N disks from a peg `A` to peg `B`, you must first move `N-1` disks from peg `A` to peg `C`, then move the last disk from `A` onto `B` and finally move `N-1` disks from `C` to `B`. In other words to solve the Hanoi(N) problem you must first solve the Hanoi(N-1) problem two times. When you only have to move one disk you can just make the move.

```
func hanoi(_ N: Int,
           from firstPeg: String = "A",
           to secondPeg: String = "B",
           using thirdPeg: String = "C") {
    if N == 1 {
        moveDisk(from: firstPeg, to: secondPeg)
    } else {
        hanoi(N - 1, from: firstPeg, to: thirdPeg, using: secondPeg)
        moveDisk(from: firstPeg, to: secondPeg)
        hanoi(N - 1, from: thirdPeg, to: secondPeg, using: firstPeg)
    }
}

hanoi(N)
```

9. Closures

Closures are self contained chunks of code that can be passed around and used in your code. Closures can capture and store references to any constants or variables from the context in which they are defined. This is know as closing over those variables, hence the name closures. Closures are use intensively in the Cocoa frameworks - which are used to develop iOS or Mac applications.

Functions are a special kind of closures. There are three kinds of closures:

- *global functions* - they have a name and cannot capture any values
- *nested functions* - they have a name and can capture values from their enclosing functions
- *closure expressions* - they don't have a name and can capture values from their context

The thing to keep in mind for the moment is that you already have an intuition about closures. They are almost the same as functions but don't necessarily have a name.

```
// a closure that has no parameters and return a String
var hello: () -> (String) = {
    return "Hello!"
}

hello() // Hello!

// a closure that take one Int and return an Int
var double: (Int) -> (Int) = { x in
    return 2 * x
}

double(2) // 4

// you can pass closures in your code, for example to other variables
var alsoDouble = double

alsoDouble(3) // 6
```

Remember the array `sort` method? There is similar one named `sort(by:)` that takes a closure as an parameter:

```
var numbers = [1, 4, 2, 5, 8, 3]

numbers.sort(by: <) // this will sort the array in ascending order
numbers.sort(by: >) // this will sort the array in descending order
```

The `<` and `>` operators are defined as functions, which can be referenced as closures. Here is an example for calling `sort` that uses a closure:

```
var numbers = [1, 4, 2, 5, 8, 3]

numbers.sort(by: { x, y in
    return x < y
})

print(numbers)
// [1, 2, 3, 4, 5, 8]
```

Declaring a closure

The general syntax for declaring closures is:

{ (`parameters`) -> `return type` **in**
`statements`
}

If the closure does not return any value you can omit the arrow (`->`) and the return type. This also applies to the case where the type of the closure can be infered.

{ (`parameters`) **in**
`statements`
}

Closures can use inout parameters, but cannot assign default values to any parameter. Also closure parameters cannot have external names.

Let's look at some examples:

```
var noParameterAndNoReturnValue: () -> () = {
    print("Hello!")
}

var noParameterAndReturnValue: () -> (Int) = {
    return 1000
}

var oneParameterAndReturnValue: (Int) -> (Int) = { x in
    return x % 10
}

var multipleParametersAndReturnValue: (String, String) -> (String) =
    { (first, second) -> String in
    return first + " " + second
}
```

The examples from above don't declare the type of each parameter, if you do so you don't need to state the return type of the closure because it can be inferred.

```
var noParameterAndNoReturnValue = {
    print("Hello!")
}

var noParameterAndReturnValue = { () -> Int in
    return 1000
}

var oneParameterAndReturnValue = { (x: Int) -> Int in
    return x % 10
}

var multipleParametersAndReturnValue =
    { (first: String, second: String) -> String in
    return first + " " + second
}
```

Shorthand Parameter Names

Swift provides shorthand parameter names for closures. You can refer to the parameters as `$0`, `$1`, `$2` and so on. To use shorthand parameter names ignore the first part of the declaration.

```
numbers.sort({ return $0 < $1 })

var double: (Int) -> (Int) = {
    return $0 * 2
}

var sum: (Int, Int) -> (Int) = {
    return $1 + $2
}
```

Capturing Values

In the beginning of the chapter I mentioned that closures can capture values. Let's see what that means:

```
var number = 0

var addOne = {
    number += 1
}

var printNumber = {
    print(number)
}

printNumber() // 0
addOne() // number is 1
printNumber() // 1
addOne() // number is 2
addOne() // number is 3
addOne() // number is 4
printNumber() // 4
```

So a closure can remember the reference of a variable or constant from its context and use it when it's called. In the example above the `number` variable is in the global context so it would have been destroyed only when the program would stop executing. Let's look at another example, in which a closure captures a variable that is not in the global context:

```
func makeIterator(from start: Int, step: Int) -> () -> Int {
    var i = start
    return {
        let currentValue = i
        i += step
        return currentValue
    }
}

var iterator = makeIterator(from: 1, step: 1)

iterator() // 1
iterator() // 2
iterator() // 3

var anotherIterator = makeIterator(from: 1, step: 3)

anotherIterator() // 1
anotherIterator() // 4
anotherIterator() // 7
anotherIterator() // 10
```

Trailing Closure Syntax

If the last parameter of a function is a closure, you can write it after the function call.

```
numbers.sort { $0 < $1 }

func sum(from: Int, to: Int, f: (Int) -> (Int)) -> Int {
    var sum = 0
    for i in from...to {
        sum += f(i)
    }
    return sum
}

sum(from: 1, to: 10) {
    $0
} // the sum of the first 10 numbers

sum(from: 1, to: 10) {
    $0 * $0
} // the sum of the first 10 squares
```

Closures are reference types

Closures are reference types. This means that when you assign a closure to more than one variable they

will refer to the same closure. This is different from value type which make a copy when you assign them to another variable or constant.

```
// a closure that take one Int and return an Int
var double: (Int) -> (Int) = { x in
    return 2 * x
}

double(2) // 4

// you can pass closures in your code, for example to other variables
var alsoDouble = double

alsoDouble(3) // 6
```

Implicit Return Values

Closures that have only one statement will return the result of that statement. To do that, simply omit the `return` keyword.

```
array.sort { $0 < $1 }
```

Higher order functions

A higher order function is a function that does at least one of the following:

- takes a function as input
- outputs a function

Swift has three important higher order functions implemented for arrays: map, filter and reduce.

Map

Map transforms an array using a function.

```
[ x1, x2, ... , xn].map(f) -> [f(x1), f(x2), ... , f(xn)]
```

Let's take as an example the problem of converting an array of numbers to an array of strings.

```
[1, 2, 3] -> ["1", "2", "3"]
```

One way of solving this problem would be to create an empty array of strings, iterate over the original array transforming each element and adding it to the new one.

```
var numbers = [1, 2, 3]

var strings: [String] = []

for number in numbers {
    strings.append("\(number)")
}
```

The other way of solving this problem is by using map:

```
var numbers = [1, 2, 3]

var strings = numbers.map { "\($0)" }
```

`{ "\($0)" }` is the closure we provided to solve this problem. It takes one parameter and converts it into a string using string interpolation.

The closure that we need to give to map take one parameter and will be called once for each of the elements from the array.

Filter

Filter selects the elements of an array which satisfy a certain condition.

For example let's remove all the odd numbers from an array:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8]

var evenNumbers = numbers.filter { $0 % 2 == 0 }
// evenNumbers = [2, 4, 6, 8]
```

Or remove all the even numbers:

```
var oddNumbers = numbers.filter { $0 % 2 == 1 }
// oddNumbers = [1, 3, 5, 7]
```

The closure that we need to give to map takes one parameter and will be called once for each of the elements from the array. It should return a `Bool` value, if it's `true`, the element will be copied into the new array, otherwise no.

Reduce

Reduce combines all the values from an array into a single value.

For example we can reduce an array of numbers to their sum.

```
var numbers = [1, 2, 3, 4, 5]

var sum = numbers.reduce(0) { $0 + $1 } // 15
```

Reduce take two parameters, an initial value and a closure that will be used to combine the elements of the array. The closure provided to reduce takes two parameters, the first one is the partial result and the second one will be an element from the array. The closure will be called for each element once. In the sum example we started the sum from 0 and the closure added the partial sum with each element.

Here is another cool way in which we can use the fact that Swift operators are implemented as functions:

```
var numbers = [1, 2, 3, 4, 5]

var sum = numbers.reduce(0, +) // 15
```

Exercises

9.1 K Times

Write a function named `applyKTimes` that takes an integer `K` and a `closure` and calls the closure `K` times. The closure will not take any parameters and will not have a return value.

Code

```
// your code here
```

Example 1

Function call:

```
applyKTimes(3) {
    print("We Heart Swift")
}
```

Output:

```
We Heart Swift
We Heart Swift
We Heart Swift
```

Example 2

Function call:

```
applyKTimes(2) {
    print("Tic")
    print("Tac")
}
```

Output:

```
Tic
Tac
Tic
```

```
Tac
```

Hint

Remember that you can call a closure just like a regular function.

9.2 Div3

Use `filter` to create an array called `multiples` that contains all the multiples of 3 from `numbers` and then print it.

Code

```
let numbers = [1, 2, 3, 4, 6, 8, 9, 3, 12, 11]

// your code here
```

Example 1

Input:

```
var numbers = [1, 2, 3, 4, 6, 8, 9, 3, 12, 11]
```

Expected values:

```
multiples = [3, 6, 9, 3, 12]
```

Example 2

Input:

```
var numbers = [3, 3, 27, 3, 99]
```

Expected values:

```
multiples = [3, 3, 27, 3, 99]
```

Example 3

Input:

```
var numbers = [2, 4, 8, 16, 32, 128, 1]
```

Expected values:

```
multiples = []
```

Hint

Think of the condition the numbers have to satisfy so that they're divisible by `3`.

9.3 Max

Find the largest number from `numbers` and then print it. Use `reduce` to solve this exercise.

Code

```
let numbers = [4, 7, 1, 9, 6, 5, 6, 9]

// your code here
```

Example

Input:

```
var numbers = [4, 7, 1, 9, 6, 5, 6, 9]
```

Output:

```
9
```

Hint 1

What would be the initial value for our reduce function?

Hint 2

How can you "combine" 2 numbers to get the maximum between them?

9.4 Join

Join all the strings from `strings` into one using `reduce`. Add spaces in between strings. Print your result.

Code

```
let strings = ["We", "Heart", "Swift"]

// your code here
```

Example 1

Input:

```
var strings = ["We", "Heart", "Swift"]
```

Output:

```
"We Heart Swift"
```

Example 2

Input:

```
var strings = ["lonely"]
```

Output:

```
"lonely"
```

Hint 1

What would be the initial value for our reduce function?

Hint 2

How can you combine the strings so that they have spaces between them?

9.5 Sorting

Sort `numbers` in ascending order by the number of divisors. If two numbers have the same number of divisors the order in which they appear in the sorted array does not matter.

Code

```
var numbers = [1, 2, 3, 4, 5, 6]

// your code here
```

Example

Input:

```
var numbers = [1, 2, 3, 4, 5, 6]
```

Expected values:

```
numbers = [1, 2, 3, 5, 4, 6]
// 1 has one divisor
// 2, 3 and 5 have 2
// 4 has 3 divisors
// 6 has 4 divisors

// [1, 5, 2, 3, 4, 6] would also have been a valid solution
```

Hint

You'll have to pass a closure that tells you how the numbers should be compared.

9.6 Chains

Find the sum of the squares of all the odd numbers from `numbers` and then print it. Use `map`, `filter` and `reduce` to solve this problem.

Code

```
var numbers = [1, 2, 3, 4, 5, 6]

// your code here
```

Example 1

Input:

```
var numbers = [1, 2, 3, 4, 5, 6]
```

Output:

```
25 // 1 + 9 + 25 -> 35
```

Example 2

Input:

```
var numbers = [2, 4, 6]
```

Output:

```
0 // none of the numbers are odd
```

Hint

The order in which you apply the `map`, `filter` and `reduce` operations is important.

9.7 For each

Implement a function `forEach(_ array: [Int], _ closure: Int -> ())` that takes an array of integers and a closure and runs the closure for each element of the array.

Example Usage

```
var array = [1,2,3,4]
forEach(array) {
    print($0 + 1)
```

```
}
// This will be printed:
// 2
// 3
// 4
// 5
```

Code

```
// your code here
```

Example 1

Function input:

```
forEach([1, 2, 3, 4]) {
    print($0 + 1)
}
```

Output:

```
2
3
4
5
```

Example 2

Function input:

```
forEach([1, 2, 3, 4]) {
    print($0 * $0)
}
```

Output:

```
1
4
9
16
```

9.8 Combine arrays

Implement a function `combineArrays` that takes 2 arrays and a closure that combines 2 Ints into a single Int. The function combines the two arrays into a single array using the provided closure.
Assume that the 2 arrays have equal length.

Code

```
// your code here
```

Example 1

Function input:

```
var array1 = [1,2,3,4]
var array2 = [5,5,5,3]
combineArrays(array1,array2) {
    $0 * $1
}
```

Function output:

```
[5,10,15,12]
```

Example 2

Function input:

```
var array1 = [5,14,77,12]
var array2 = [1,5,3,13]
combineArrays(array1,array2) {
    return max($0,$1)
}
```

Function output:

```
[5,14,77,13]
```

Hint

You'll have to iterate both arrays at the same time using an index.

Solutions

9.1 K Times

To solve this problem we need to call the passed `closure` `K` times. We can do this in many ways - as seen in the loops chapter.

```
func applyKTimes(_ K: Int, _ closure: () -> ()) {
    for _ in 1...K {
        closure()
    }
}
```

9.2 Div3

To select only the multiples of 3 from `numbers` we can `filter` it using a closure that will return `true` for numbers that are divisible with 3 and `false` otherwise.

```
let numbers = [1, 2, 3, 4, 6, 8, 9, 3, 12, 11]

let multiples = numbers.filter { $0 % 3 == 0 }

print(multiples)
```

9.3 Max

The closure provided to `reduce` takes two parameters, the first one is the partial result and the second one will be an element from the array. The closure will be called for each element once in order. The example shown for the sum of the element from an array used the first parameter to store the partial result of the sum. To get the maximum value the partial result will be the biggest number so far. When handling a new number if it is bigger than the partail maximum then this number will be the new partial maximum otherwise it will remain unchanged.

```
let numbers = [4, 7, 1, 9, 6, 5, 6, 9]

let max = numbers.reduce(numbers[0]) {
    if $0 > $1 {
        return $0
    } else {
        return $1
    }
}

print(max) // 9
```

9.4 Join

This is another example of using `reduce`. We are going to start with an empty string `""`. When handling a new element from `strings` we are going to have two cases: 1) the partial result is an empty string -> it's the first one - then we return it as the partial result 2) the partial result is not empty -> it's not the first one - we have to add a space (`" "`) and the current string to the partial result and return it

```
let strings = ["We", "Heart", "Swift"]

let string = strings.reduce("") {
    if $0 == "" {
        return $1
    } else {
        return $0 + " " + $1
    }
}

print(string)
```

9.5 Sorting

To sort an array with the `sort(by:)` function we need a compare function or closure. To sort `numbers` in ascending order by the number of divisors are are goin to make a closure that compares the number of divisors of the first number with the one of the second. To avoid code duplication - we need to count the number of divisors of two numbers - we can use a nested function inside the closure.

```
var numbers = [1, 2, 3, 4, 5, 6]

numbers.sort(by: { x, y in
    func countDivisors(_ number: Int) -> Int {
        var count = 0
        for i in 1...number {
            if number % i == 0 {
                count += 1
            }
        }
        return count
    }
    return countDivisors(x) < countDivisors(y)
})
```

9.6 Chains

To solve this problem we shoudl solve each step separately. First of all let's get all the odd numbers from `numbers` : this is similar to the multiples of 3 problem. We are goin to use `filter` and a closure that returns `true` if a number is odd. Then we have to square all the numbers. We can do this using `map` . In the end we need to add all the numbers - this can be done with `.reduce(0, combine: +)` .

```
var numbers = [1, 2, 3, 4, 5, 6]

let sum = numbers.filter {
        $0 % 2 == 1 //select all the odd numbers
    }.map {
        $0 * $0 // square them
    }.reduce(0, +) // get their sum

print(sum)
```

9.7 For each

This exercise is similar with the first one from the chapter. All we need to do is iterate over the given array and call the closure with each element.

```
func forEach(_ array: [Int], _ closure: (Int) -> ()) {
    for number in array {
        closure(number)
    }
}
```

9.8 Combine arrays

To combine two arrays we need to iterate through them both at the same time and call the closure with the respective elements. We also need to save the results from each call and return it in the end.

```
func combineArrays(_ array1: [Int],
                   _ array2: [Int],
                   _ closure: (Int,Int) -> Int) -> [Int] {
    var result: [Int] = []
    for i in 0..<array1.count {
        result.append(closure(array1[i],array2[i]))
    }
    return result
}
```

10. Tuples & Enums

Tuples

A tuple is a group of zero or more values represented as one value.

For example `("John", "Smith")` holds the first and last name of a person. You can access the inner values using the dot(`.`) notation followed by the index of the value:

```
var person = ("John", "Smith")

var firstName = person.0 // John
var lastName = person.1 // Smith
```

Named elements

You can name the elements from a tuple and use those names to refer to them. An element name is an identifier followed by a colon(:).

```
var person = (firstName: "John", lastName: "Smith")

var firstName = person.firstName // John
var lastName = person.lastName // Smith
```

Creating a tuple

You can declare a tuple like any other variable or constant. To initialize it you will need a another tuple or a tuple literal. A tuple literal is a list of values separated by commas between a pair of parentheses. You can use the dot notation to change the values from a tuple if it's declared as a variable.

```
var point = (0, 0)

point.0 = 10
point.1 = 15

point // (10, 15)
```

Note: Tuple are value types. When you initialize a variable tuple with another one it will actually create a copy.

```
var origin = (x: 0, y: 0)
```

```
var point = origin
point.x = 3
point.y = 5

print(origin) // (0, 0)
print(point) // (3, 5)
```

Types

The type of a tuple is determined by the values it has. So `("tuple", 1, true)` will be of type `(String, Int, Bool)`. You can have tuples with any combination of zero or more types.

If the tuple has only one element then the type of that tuple is the type of the element. `(Int)` is the same as `Int`. This has a strange implication: in swift every variable or constant is a tuple.

```
var number = 123

print(number) // 123
print(number.0) // 123
print(number.0.0) // 123
print(number.0.0.0) // 123
print(number.0.0.0.0.0.0) // 123
```

Empty tuple

`()` is the empty tuple - it has no elements. It also represents the `Void` type.

Decomposing Tuples

```
var person = (firstName: "John", lastName: "Smith")

var (firstName, lastName) = person

var (onlyFirstName, _) = person
var (_, onlyLastName) = person
```

Note: the `_` means "I don't care about that value"

Multiple assignment

You can use tuples to initialize more than one variable on a single line:

Instead of:

```
var a = 1
var b = 2
```

```
var c = 3
```

you can write:

```
var (a, b, c) = (1, 2, 3)
```

Instead of:

```
a = 1
b = 2
c = 3
```

you can write:

```
(a, b, c) = (1, 2, 3)
```

And yes! One line swap:

```
(a, b) = (b, a)
```

Returning multiple values

You can return multiple values from a function if you set the result type to a tuple. Here is a simple example of a function that returns the quotient and the remainder of the division of `a` by `b`.

```
func divmod(_ a: Int, _ b:Int) -> (Int, Int) {
    return (a / b, a % b)
}

divmod(7, 3) // (2, 1)
divmod(5, 2) // (2, 1)
divmod(12, 4) // (3, 0)
```

Or the named version:

```
func divmod(_ a: Int, _ b:Int) -> (quotient: Int, remainder: Int) {
    return (a / b, a % b)
}
```

```
divmod(7, 3) // (quotient: 2, remainder:1)
divmod(5, 2) // (quotient: 2, remainder:1)
divmod(12, 4) // (quotient: 3, remainder:0)
```

Enums

An enumeration is a data type consisting of a set of named values, called members.

Defining an enumeration

You can define a new enumeration using the `enum` keyword followed by it's name. The member values are introduced using the `case` keyword.

```
enum iOSDeviceType {
    case iPhone
    case iPad
    case iWatch
}

var myDevice = iOSDeviceType.iPhone
```

Dot syntax

If the type of an enumeration is known or can be inferred then you can use the dot syntax for members.

```
// in this case the type is known
var myDevice: iOSDeviceType = .iPhone

// in this case the type can be inferred
if myDevice == .iPhone {
    print("I have an iPhone!")
}
```

Associated Values

Swift enumerations can store associated values of any type, and the value type can be different for each member. For example you might want to store a device model for the iPhone and iPad (like `"mini"` for the iPad, or `"6 Plus"` for the iPhone).

```
enum iOSDeviceType {
    case iPhone(String)
    case iPad(String)
    case iWatch
}
```

You can get the associated values by using a switch statement:

```
var myDevice = iOSDeviceType.iPhone("6")

switch myDevice {
case .iPhone(let model):
    print("iPhone \(model)")
case .iPad(let model):
    print("iPad \(model)")
case .iWatch:
    print("iWatch")
default:
    print("not an iOS device")
}

// iPhone 6
```

Note: Swift does not provide equality operators automatically for enumerations with associated values. You might be tempted to use nested switch statements in order to test equality. Don't forget the tuple pattern!

```
var myDevice = iOSDeviceType.iPhone("6")
var six = iOSDeviceType.iPhone("6")
var sixPlus = iOSDeviceType.iPhone("6 Plus")

// testing equlity with == wont work
// myDevice == six
// myDevice == sixPlus

func sameDevice(_ firstDevice: iOSDeviceType,
        secondDevice: iOSDeviceType) -> Bool {
    switch (firstDevice, secondDevice) {
    case (.iPhone(let a), .iPhone(let b)) where a == b:
        return true
    case (.iPad(let a), .iPad(let b)) where a == b:
        return true
    case (.iWatch, .iWatch):
        return true
    default:
        return false
    }
}

print(sameDevice(myDevice, six)) // true
print(sameDevice(myDevice, sixPlus)) // false
print(sameDevice(myDevice, .iWatch)) // false
```

Raw Values

Enums can have a raw value (a primitive type - Int, String, Character, etc.) associated with each member. The raw value will be of the same type for all members and the value for each member must be unique. When integers are used, they autoincrement if a value is not defined for a member.

```
enum Direction: Int {
    case up = 1
    case down // will have the raw value 2
    case left // will have the raw value 3
    case right // will have the raw value 4
}
```

You can use raw values to create a enumeration value.

```
var direction = Direction(rawValue: 4) // .right

print(direction) // Optional((Enum Value))
```

Note: Because not all raw values have an associated member value the raw value initializer is a failable initializer. The type of `direction` is `Direction?` not `Direction`.

Exercises

10.1 Game

You are working on a game in which your character is exploring a grid-like map. You get the original `location` of the character and the `steps` he will take.
A step `.up` will increase the x coordinate by 1. A step `.down` will decrease the x coordinate by 1. A step `.right` will increase the y coordinate by 1. A step `.left` will decrease the y coordinate by 1.
Print the final `location` of the character after all the steps have been taken.

Code

```
enum Direction {
    case up
    case down
    case left
    case right
}

var location = (x: 0, y: 0)

var steps: [Direction] = [.up, .up, .left, .down, .left]

// your code here
```

Example 1

Input:

```
var location = (x: 0, y: 0)

var steps: [Direction] = [.up, .up, .left, .down, .left]
```

Output:

```
(-2, 1)
```

Example 2

Input:

```
var location = (x: 0, y: 0)
```

```
var steps: [Direction] = [.up, .up, .left, .down, .left, .down, .down,
    .right, .right, .down, .right]
```

Output:

```
(1, -2)
```

Example 3

Input:

```
var location = (x: 5, y: 2)

var steps: [Direction] = [.up, .right, .up, .right, .up,
    .right, .down, .right]
```

Output:

```
(9, 4)
```

Hint 1

Use a switch statement.

Hint 2

Modify the `location` tuple based on what case you're handling in the `switch` statement.

10.2 Min Max

Write a function named `minmax` that takes two integers and returns both the minimum and the maximum values inside a tuple.

Code

```
// your code here
```

Example 1

Function call:

```
minmax(2, 3)
```

Function output:

```
(2, 3)
```

Example 2

Function call:

```
minmax(5, 1)
```

Function output:

```
(1, 5)
```

Example 3

Function call:

```
minmax(3, 3)
```

Function output:

```
(3, 3)
```

Hint 1

A single comparison is enough to determine both the minimum and the maximum value.

10.3 Rock, Paper, Scissors

1) Define an enumeration named `HandShape` with three members: `.rock`, `.paper`, `.scissors`.

2) Define an enumeration named `MatchResult` with three members: `.win`, `.draw`, `.lose`.
3) write a function called `match` that takes two hand shapes and returns a match result. It should return the outcome for the first player (the one with the first hand shape).

Code

```
// your code here
```

Example 1

Function call:

```
match(first: .rock, second: .scissors)
```

Function output:

```
.win
```

Example 2

Function call:

```
match(first: .rock, second: .paper)
```

Function output:

```
.lose
```

Example 3

Function call:

```
match(first: .scissors, second: .scissors)
```

Function output:

```
.draw
```

Hint 1

Handle the case when the hands result in a draw first.

Hint 2

Determine if a win has occurred.

10.4 Fractions

You are given 2 tuples of `a`, `b` type `(Int,Int)` representing fractions. The first value in the tuple represents the numerator, the second value represents the denominator. Create a new tuple `sum` of type `(Int,Int)` that holds the sum of the fractions.

Code

```
var a = (5,8)
var b = (17,9)

// your code here
```

Example 1

Input:

```
var a = (5,8)
var b = (17,9)
```

Expected Value:

```
sum = (181, 72)
```

Example 2

Input:

```
var a = (34,3)
var b = (11,2)
```

```
```

Expected Value:

```
sum = (101, 6)
```

Hint

To add 2 fractions together you have to get them to a common denominator.

10.5 Money

You are given the `CoinType` enumeration which describes different coin values and `moneyArray` which has tuples `(ammount, coinType)`. Print the total value of the coins in the array.

Code

```
enum CoinType: Int {
    case penny = 1
    case nickle = 5
    case dime = 10
    case quarter = 25
}

var moneyArray:[(Int,CoinType)] = [(10,.penny),
                                   (15,.nickle),
                                   (3,.quarter),
                                   (20,.penny),
                                   (3,.dime),
                                   (7,.quarter)]

// your code here
```

Example 1

Input:

```
var moneyArray:[(Int,CoinType)] = [(10,.penny),
    (15,.nickle),
    (3,.quarter),
    (20,.penny),
    (3,.dime),
    (7,.quarter)]
```

Output:

```
```

```
385
```

Example 2

Input:

```
var moneyArray:[(Int,CoinType)] = [
    (2,.penny),
    (3,.quarter)
]
```

Output:

```
77
```

Example 3

Input:

```
var moneyArray:[(Int,CoinType)] = [
    (5, .dime),
    (2, .quarter),
    (1, .nickle)
]
```

Output:

```
105
```

Hint

Remember that `.rawValue` gets the numeric value associated with an enum value.

10.6 Counting Strings

You are given an array of strings stored in the variable `strings`. Create a new array named `countedStrings` containing values of type `(String,Int)`. Each tuple contains a string from the `strings` array followed by an integer indicating how many times it appears in the `strings` array. Each string should only appear once in the `countedStrings` array.

Code

```
var strings = ["tuples", "are", "awesome", "tuples", "are", "cool",
    "tuples", "tuples", "tuples", "shades"]

// your code here
```

Example 1

Input:

```
var strings = ["tuples", "are", "awesome", "tuples", "are", "cool",
    "tuples", "tuples", "tuples", "shades"]
```

Expected Value:

```
countedStrings = [
    "tuples" : 5,
    "are" : 2,
    "awesome" : 1,
    "cool" : 1,
    "shades" : 1
]
```

Example 2

Input:

```
var strings = ["hello", "world", "hello", "swift", "hello", "tuples"]
```

Expected Value:

```
countedStrings = [
    "hello" : 3,
    "world" : 1,
    "swift" : 1,
    "tuples" : 1
]
```

Hint

Keep in mind that you can't change a tuple when iterating an array using the for in syntax. You'll have to

iterate using an index.

Solutions

10.1 Game

We have to iterate over all the steps. Will use a switch statement on each step, depending on the value of the enum we'll update the location tuple's x or y coordinate. If the enum's value is Up or Down location.y is increased or decreased respectivelly. If the enum's value is Right or Left the location.x is increased or decreased respectivelly.

```
enum Direction {
    case up
    case down
    case left
    case right
}

var location = (x: 0, y: 0)

var steps: [Direction] = [.up, .up, .left, .down, .left]

for step in steps {
    switch step {
    case .up:
        location.y += 1
    case .down:
        location.y -= 1
    case .right:
        location.x += 1
    case .left:
        location.x -= 1
    default:
        break
    }
}

print(location)
```

10.2 Min Max

If `a < b` we return the tuple `(a, b)` otherwise we return the tuple `(b, a)`.

```
func minmax(_ a: Int, _ b: Int) -> (Int, Int) {
    if a < b {
        return (a, b)
    } else {
        return (b, a)
    }
}
```

10.3 Rock, Paper, Scissors

First we'll define the `HandShape` and `MatchResult` enums like above. Now to implement the match function we'll treat the draw case and all the win cases otherwise we'll return `.lose`. If the first hand shape is equal to the second hand shape we return a `.draw`. Next we'll return a `.win` if any of the winning conditions of rock scissor paper are met. Otherwise we'll return a `.lose`.

```
enum HandShape {
    case rock
    case paper
    case scissors
}

enum MatchResult {
    case win
    case draw
    case lose
}

func match(first: HandShape, second: HandShape) -> MatchResult {
    if first == second {
        return .draw
    }

    if first == .rock && second == .scissors {
        return .win
    }

    if first == .paper && second == .rock {
        return .win
    }

    if first == .scissors && second == .paper {
        return .win
    }

    return .lose
}
```

10.4 Fractions

The solution involves adding the fractions via the stadard fraction addition formula. `a/b + c/d = (a * d + c * b / b * d`.

```
var a = (5,8)
var b = (17,9)

let numerator = a.0 * b.1 + b.0 * a.1
let denominator = a.1 * b.1
var sum = (numerator, denominator)
```

10.5 Money

We'll have to iterate through our money array. We'll keep track of the total money in a variable. For each `(amount, coinType)` tuple we add the amount multiplied by the `coinType`'s raw value to our total.

```
enum CoinType: Int {
    case penny = 1
    case nickle = 5
    case dime = 10
    case quarter = 25
}

var moneyArray:[(Int,CoinType)] = [(10,.penny),
                                   (15,.nickle),
                                   (3,.quarter),
                                   (20,.penny),
                                   (3,.dime),
                                   (7,.quarter)]

var totalMoney = 0

for (amount, coinType) in moneyArray {
    totalMoney += amount * coinType.rawValue
}

print(totalMoney)
```

10.6 Counting Strings

We'll want to iterate all the strings in our `strings` array. The result will be kept in our `countedStrings` array. At each step of the loop we first check if the string already exists in some tuple of the array, if it does we keep remember that in the boolean variable `alreadyExists` and increment the count of the tuple by `1`. After we finish iterating our `countedStrings` array we check if the `string` was already in the array, if it wasn't we add it with a count of 1.

```
var strings = ["tuples", "are", "awesome", "tuples", "are", "cool",
    "tuples", "tuples", "tuples", "shades"]

var countedStrings: [(String,Int)] = []

for string in strings {
    var alreadyExists = false

    for i in 0..<countedStrings.count {
        if (countedStrings[i].0 == string) {
            countedStrings[i].1 += 1
            alreadyExists = true
        }
    }
    if alreadyExists == false {
        let tuple = (string,1)
        countedStrings.append((string,1))
    }
}
```

11. Dictionaries

A dictionary is an unordered collection that stores multiple values of the same type. Each value from the dictionary is asociated with a unique key. All the keys have the same type.

The type of a dictionary is determined by the type of the keys and the type of the values. A dictionary of type `[String:Int]` has keys of type `String` and values of type `Int`.

Declare Dictionaries

To declare a dictionary you can use the square brackets syntax(`[KeyType:ValueType]`).

```
var dictionary: [String:Int]
```

You can initialize a dictionary with a dictionary literal. A dictionary literal is a list of key-value pairs, separated by commas, surrounded by a pair of square brackets. A key-value pair is a combination of a key and a value separate by a colon(:).

[`key` : `value` , `key` : `value` , `...`]

```
var dictionary: [String:Int] = [
    "one" : 1,
    "two" : 2,
    "three" : 3
]
```

Keep in mind that you can create empty dictionary using the empty dictionary literal (`[:]`).

```
var emptyDictionary: [Int:Int] = [:]
```

Getting values

You can access specific elements from a dictionary using the subscript syntax. To do this pass the key of the value you want to retrieve within square brackets immediately after the name of the dictionary. Because it's possible not to have a value associated with the provided key the subscript will return an optional value of the value type.

To unwrap the value returned by the subscript you can do one of two things: use optional binding or force the value if you know for sure it exists.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2,
    "three" : 3,
    "four" : 4,
    "five" : 5,
    "six" : 6,
    "seven" : 7,
    "eight" : 8,
    "nine" : 9
]

stringsAsInts["zero"] // Optional(0)
stringsAsInts["three"] // Optional(3)
stringsAsInts["ten"] // nil

// Unwaraping the optional using optional binding
if let twoAsInt = stringsAsInts["two"] {
    print(twoAsInt) // 2
}

// Unwaraping the optional using the forced value operator (!)
stringsAsInts["one"]! // 1
```

To get all the values from a dictionary you can use the `for-in` syntax. It's similar to the array for in syntax with the exception that instead of getting only the value in each step you also get the key associated with that value inside of a tuple.

```
var userInfo: [String: String] = [
    "first_name" : "Andrei",
    "last_name" : "Puni",
    "job_title" : "Mad scientist"
]

for (key, value) in userInfo {
    print("\(key): \(value)")
}
```

To get the number of elements (key-value pairs) in a dictionary you can use the `count` property.

```
print(userInfo.count) // 3
```

Updating values

The simplest way to add a value to a dictionary is by using the subscript syntax:

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
```

```
    "two" : 2
]

stringsAsInts["three"] = 3
```

Using the subscript syntax you can change a the value associated with a key:

```
stringsAsInts["three"] = 10
```

You can use the `updateValue(forKey:)` method to update the value associated with a key, if there was no value for that key it will be added. The method will return the old value wrapped in an optional or nil if there was no value before.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2
]

stringsAsInts.updateValue(3, forKey: "three") // nil
stringsAsInts.updateValue(10, forKey: "three") // Optional(3)
```

To remove a value from the dictionary you can use the subscript syntax to set the value to nil, or the `removeValueForKey()` method.

```
stringsAsInts["three"] = nil

stringsAsInts.removeValueForKey("three")
```

Type Inference

Thanks to Swift's type inference, you don't have to declare the type of a dictionary if you initialize it with something other than an empty dictionary literal(`[:]`).

```
// powersOfTwo will have the type [Int:Int]
var powersOfTwo = [
    1 : 2,
    2 : 4,
    3 : 8,
    4 : 16
]

// userInfo will have the type [String:String]
var userInfo = [
    "first_name" : "Silviu",
    "last_name" : "Pop",
```

```
    "job_title" : "evil genius"
]
```

Copy Behavior

Swift's dictionaries are value types. This means that dictionaries are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2
]

var justACopy = stringsAsInts

justACopy["zero"] = 100

print(stringsAsInts) // [zero: 0, one: 1, two: 2]
print(justACopy) // [zero: 100, one: 1, two: 2]
```

Keep in mind that this is not true for Objective-C dictionaries (`NSDictionary` and `NSMutableDictionary`).

Mutability

If you create a dictionary and assign it to a variable, the collection that is created will be mutable. This means that you can change (or mutate) the collection after it is created by adding, removing, or changing items in the collection. Conversely, if you assign a dictionary to a constant, that array or dictionary is immutable, and its size and contents cannot be changed. In other words if you want to be able to change a dictionary declare it using the `var` keyword, and if you don't want to be able to change it use the `let` keyword.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2
]

stringsAsInts["three"] = 3 // [zero: 0, one: 1, two: 2, three: 3]
stringsAsInts["zero"] = nil // [one: 1, two: 2, three: 3]

let powersOfTwo = [
    1 : 2,
    2 : 4,
    3 : 8,
    4 : 16
]

// this will give a runtime error because powersOfTwo is immutable
powersOfTwo[5] = 32
```

```
powersOfTwo.removeValueForKey(1) // this will give a similar error
```

Exercises

11.1 Encode

You are given a dictionary `code` of type `[String:String]` which has values for all lowercase letters. The `code` dictionary represents a way to encode a message. For example if `code["a"] = "z"` and `code["b"] = "x"` the encoded version if `"ababa"` will be `"zxzxz"`. You are also given a `message` which contains only lowercase letters and spaces. Use the `code` dictionary to encode the message and print it.

Code

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var message = "hello world"

// your code here
```

Example 1

Input:

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
```

```
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var message = "hello world"
```

Output:

```
ifmmp xpsme
```

Example 2

Input:

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
```

```
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var message = "wow this problem is hard"
```

Output:

```
xpx uijt qspcmfn jt ibse
```

Hint 1

If a character doesn't have a corresponding encoded character leave it unchanged.

Hint 2

Build the encoded message step by step by getting the corresponding encoded character from the dictionary.

11.2 Decode

You are given a dictionary `code` of type `[String:String]` which has values for all lowercase letters. The `code` dictionary represents a way to encode a message. For example if `code["a"] = "z"` and `code["b"] = "x"` the encoded version if `"ababa"` will be `"zxzxz"`. You are also given a `encodedMessage` which contains only lowercase letters and spaces. Use the `code` dictionary to decode the message and print it.

Code

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
```

```
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var encodedMessage = "uijt nfttbhf jt ibse up sfbe"

// your code here
```

Example

Input:

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var encodedMessage = "uijt nfttbhf jt ibse up sfbe"
```

Output:

```
this message is hard to read
```

Hint

You'll have to invert the code dictionary. Create a new dictionary for this.

11.3 Names

You are given an array of dictionaries. Each dictionary in the array contains exactly 2 keys "firstName" and "lastName". Create an array of strings called `firstNames` that contains only the values for "firstName" from each dictionary.

Code

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]

// your code here
```

Example

Input:

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
```

```
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]
```

Expected values:

```
firstNames = ["Calvin","Garry","Leah","Sonja","Noel"]
```

Hint

Keep in mind that persons is an array of dictionaries, you'll have to process this array to get the required data.

11.4 Full names

You are given an array of dictionaries. Each dictionary in the array contains exactly 2 keys "firstName" and "lastName". Create an array of strings called `fullNames` that contains the values for "firstName" and "lastName" from the dictionary separated by a space.

Code

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]

// your code here
```

Example

Input:

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]
```

Expected values:

```
fullNames = ["Calvin Newton","Garry Mckenzie","Leah Rivera",
    "Sonja Moreno","Noel Bowen"]
```

11.5 Best score

You are given an array of dictionaries. Each dictionary in the array describes the score of a person. Find the person with the best score and print his full name.

Code

```
var people: [[String:Any]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton",
        "score": 13
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie",
        "score": 23
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera",
        "score": 10
    ],
```

```
    [
        "firstName": "Sonja",
        "lastName": "Moreno",
        "score": 3
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen",
        "score": 16
    ]
]

// your code here
```

Example

Input:

```
var people: [[String:Any]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton",
        "score": 13
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie",
        "score": 23
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera",
        "score": 10
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno",
        "score": 3
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen",
        "score": 16
    ]
]
```

Output:

```
Garry Mckenzie
```

Hint

Keep track of the person with the best score that you've encountered.

11.6 Leaderboard

You are given an array of dictionaries. Each dictionary in the array describes the score of a person. Print the leaderboard in the following format:

```
1. full name - score
2. ...
...
```

Code

```
var people: [[String:Any]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton",
        "score": 13
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie",
        "score": 23
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera",
        "score": 10
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno",
        "score": 3
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen",
        "score": 16
    ]
]

// your code here
```

Example

Input:

```
var people: [[String:Any]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton",
        "score": 13
    ],
    [
        "firstName": "Garry",
```

```
        "lastName": "Mckenzie",
        "score": 23
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera",
        "score": 10
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno",
        "score": 3
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen",
        "score": 16
    ]
]
```

Output:

```
1. Garry Mckenzie - 23
2. Noel Bowen - 16
3. Calvin Newton - 13
4. Leah Rivera - 10
5. Sonja Moreno - 3
```

Hint

Sort the list of people using a function that compares two dictionaries.

11.7 Frequency

You are given an array of integers. Find out the frequency of each one.
The frequency of a number is the number of times it appears in the array.
Print the numbers in ascending order followed by their frequency.

Code

```
var numbers = [1, 2, 3, 2, 3, 5, 2, 1, 3, 4, 2, 2, 2]

// your code here
```

Example

Input:

```
var numbers = [1, 2, 3, 2, 3, 5, 2, 1, 3, 4, 2, 2, 2]
```

Output:

```
1 2
2 6
3 3
4 1
5 1
```

Hint 1

Use a dictionary to keep track of the frequency.

Hint 2

Keep track of all the unique numbers of the array

Solutions

11.1 Encode

We'll initialized our `encodedMessage` with the empty string. Next we'll iterate over all the characters in our `message`. If the character has a coresponding entry in our `code` dictionary we add the encoded character to our `encodedMessage` otherwise we add the character as is (the character is not an alphabetical character).

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var message = "hello world"

var encodedMessage = ""

for char in message.characters {
    var character = "\(char)"

    if let encodedChar = code[character] {
        // letter
        encodedMessage += encodedChar
    } else {
        // space
        encodedMessage += character
    }
}

print(encodedMessage)
```

11.2 Decode

We'll initialized our `decodedMessage` with the empty string. Next we'll iterate over all the characters in our `encodedMessage`. If the character has a coresponding entry in our `decode` dictionary we add the decoded character to our `decodedMessage` otherwise we add the character as is (the character is not an alphabetical character).

```
var code = [
    "a" : "b",
    "b" : "c",
    "c" : "d",
    "d" : "e",
    "e" : "f",
    "f" : "g",
    "g" : "h",
    "h" : "i",
    "i" : "j",
    "j" : "k",
    "k" : "l",
    "l" : "m",
    "m" : "n",
    "n" : "o",
    "o" : "p",
    "p" : "q",
    "q" : "r",
    "r" : "s",
    "s" : "t",
    "t" : "u",
    "u" : "v",
    "v" : "w",
    "w" : "x",
    "x" : "y",
    "y" : "z",
    "z" : "a"
]

var encodedMessage = "uijt nfttbhf jt ibse up sfbe"

var decoder: [String:String] = [:]

// reverse the code
for (key, value) in code {
    decoder[value] = key
}

var decodedMessage = ""

for char in encodedMessage.characters {
    var character = "\(char)"

    if let encodedChar = decoder[character] {
        // letter
        decodedMessage += encodedChar
    } else {
        // space
        decodedMessage += character
    }
}

print(decodedMessage)
```

11.3 Names

We'll want to iterate the dictionaries in our `people` array. To obtain the firstName we get the value for key "firstName" from each dictionary.

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]

var firstNames: [String] = []

for person in people {
    if let firstName = person["firstName"] {
        print(firstName)
        firstNames.append(firstName)
    }
}
```

11.4 Full names

We'll want to iterate the dictionaries in our `people` array. To obtain the firstName we get the value for the key "firstName" from each dictionary. To obain the lastName we get the value for the key "lastName" from each dictionary. We combine these to get the fullName.

Solution 1

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
```

```
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]

var fullNames: [String] = []

for person in people {
    if let firstName = person["firstName"] {
        if let lastName = person["lastName"] {
            let fullName = "\(firstName) \(lastName)"
            fullNames.append(fullName)
        }
    }
}
```

Solution 2

```
var people: [[String:String]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton"
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie"
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera"
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno"
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen"
    ]
]

var fullNames: [String] = []

for person in people {
    var fullName = " "
    for (key, value) in person {
        if key == "lastName" {
            fullName += value
        } else {
            fullName = value + fullName
        }
    }
```

```
    fullNames += [fullName]
}

print(fullNames)
```

11.5 Best score

We keep track of the best person in variable `topPerson` and of the best score in the variable `bestScore`. We iterate our array of dictionaries, if we encounter a score that's larger than our current `bestScore` we update our `bestScore` and `topPerson` variables. Lastly we print the result by getting the values for the keys "firstName" and "lastName".

```
var people: [[String:Any]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton",
        "score": 13
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie",
        "score": 23
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera",
        "score": 10
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno",
        "score": 3
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen",
        "score": 16
    ]
]

var topPerson = people[0]
var bestScore = topPerson["score"] as! Int

for person in people {
    if let score = person["score"] as? Int {
        if bestScore < score  {
            bestScore = score
            topPerson = person
        }
    }
}

if let first = topPerson["firstName"] as? String,
   let second = topPerson["lastName"] as? String  {
    print("\(first) \(second)")
}
```

11.6 Leaderboard

First we'll want to sort our people array using a custom comparison function. This function will compare the people's scores and sort them in descending order. After we have the people array sorted in descending order by score we want to iterate over the people array and print the data in the corresponding format.

```
var people: [[String:Any]] = [
    [
        "firstName": "Calvin",
        "lastName": "Newton",
        "score": 13
    ],
    [
        "firstName": "Garry",
        "lastName": "Mckenzie",
        "score": 23
    ],
    [
        "firstName": "Leah",
        "lastName": "Rivera",
        "score": 10
    ],
    [
        "firstName": "Sonja",
        "lastName": "Moreno",
        "score": 3
    ],
    [
        "firstName": "Noel",
        "lastName": "Bowen",
        "score": 16
    ]
]

func compareScores(_ first: [String:Any], second: [String:Any]) -> Bool {
    if let a = first["score"] as? Int {
        if let b = second["score"] as? Int {
            return a > b
        }
    }
    return false
}

people.sort(by: compareScores)

for (index, person) in people.enumerated() {
    if let firstName = person["firstName"] as? String {
        if let lastName = person["lastName"] as? String  {
            if let score = person["score"] as? Int {
                print("\(index + 1). \(firstName) \(lastName) - \(score)")
            }
        }
    }
}
```

11.7 Frequency

We are going to use a dictionary `[Int -> Int]` to store the frequency of each number. When we first find

a number we initialize it's count with 1 and then we increase it each time. We also use an array of [Int] to uniquely keep track of each number we find. To get the numbers in ascending order followed by their frequency we sort the array of unique numbers.

```
var numbers = [1, 2, 3, 2, 3, 5, 2, 1, 3, 4, 2, 2, 2]

var frequency: [Int:Int] = [:]
var uniqueNumbers: [Int] = []

for number in numbers {
    if let oldFr = frequency[number] {
        frequency[number] = oldFr + 1
    } else {
        uniqueNumbers.append(number)
        frequency[number] = 1
    }
}

uniqueNumbers.sort(by: <)

for number in uniqueNumbers {
    print("\(number) \(frequency[number]!)")
}
```