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.