Similar functions

Computing the sum of integers:

```
fun sumInt(a: Int, b: Int): Int {
  var s = 0
  for (i in a .. b)
    s += i
  return s
}
```

Computing the sum of cubes:

```
fun sumCubes(a: Int, b: Int): Int {
  var s = 0
  for (i in a .. b)
    s += i * i * i
  return s
}
```


These are special cases of

$$\sum_{i=a}^{b} f(i)$$

for different choices of the function f.

If mathematics has a notation for this, we should have one, too!

A higher-order function

```
fun sum(a: Int, b: Int, f: (Int) -> Int): Int {
  var s = 0
  for (i in a..b)
    s += f(i)
  return s
}
```

`(Int) -> Int` is the type of a function that maps `Int` to `Int`.

In general, a function that maps arguments of types `A`, `B`, `C` to `R` has type `(A, B, C) -> R`.

But how can we provide the function argument to `sum`?

Function literals

A literal is programming language syntax to create a nameless object.

We use literals all the time. Instead of this:

```
>>> val str: String = "Hello CS109"
>>> val a: Int = 13
>>> println(str); println(a)
```

we use this:

```
>>> println("Hello CS109"); println(13)
```

A literal creates an object (without giving it a name). A function literal creates a function object.

Function literals are also called anonymous functions or lambdas.

Function literals

A function literal or anonymous function creates a function object without giving it a name.

For example: A function that raises an integer to its cube:

```
{ x: Int -> x * x * x }
```

Here, `x: Int` is the parameter of the function, and `x * x * x` is its body.

An anonymous function with several parameters:

```
{ a: Int, b: Int -> a + b }
```

A function literal creates a function object without giving it a name.

Function objects

Function objects are stored on the heap like all other objects. Variables can store a reference to a function object. They can be called like functions:

```
>>> {x : Int -> x * x * x}
(kotlin.Int) -> kotlin.Int
>>> {x : Int -> x * x * x}(3)
27
>>> val f = {x : Int -> x * x * x}
>>> f(3)
27
>>> f(7)
343
>>> f(-30)
-27000
```


Function objects

```
>>> val g = listOf({ x: Int -> x * x },
...                { x: Int -> x * x * x },
...                { x: Int -> x * x * x * x })
>>> g[0](2)
4
>>> g[1](2)
8
>>> g[2](2)
16
```


Summation with function literals

We can now write our summations like this:

```
>>> sum(1, 100, { x: Int -> x } )
5050
>>> sum(1, 100, { x: Int -> x * x * x } )
25502500
```

When the compiler can determine the type of the arguments in the function literal, we can omit them:

```
>>> sum(1, 100, { x -> x } )
5050
>>> sum(1, 100, { x -> x * x * x } )
25502500
```

(This works because the argument `f` of `sum` is a function of type `(Int) -> Int`.)

Syntactic sugar

Kotlin has some more "syntactic sugar" for using function literals in arguments.

If the function literal is the last argument, we can put it outside the parentheses:

```
>>> sum(1, 100) { x -> x }
5050
>>> sum(1, 100) { x -> x * x * x }
25502500
```

If the function literal has only one argument, we can use the magic name `it`:

```
>>> sum(1, 100) { it }
5050
>>> sum(1, 100) { it * it * it }
25502500
```


Higher-Order Functions

Functions like `sum` are called higher-order functions because they take another function object as an argument: A function that works on functions.

Higher-order functions allow us to express ideas such as:

- print a table with a given function
- integrate a function numerically
- find a fixed point of a function.

Collections have higher-order methods

All collections provide many useful higher-order methods that allow you to express in one line what otherwise would have to be a `for`-loop.

```
>>> val words= java.io.File("words.txt").readLines(
>>> words.max()
zymurgy
>>> words.maxBy { it.length }
counterdemonstrations
```

Full literal: `{ s: String -> s.length }`

Any, all, count, and find

These four methods take a predicate argument: a function literal that returns `true` or `false`.

```
>>> words.count { "e" !in it }
37641
>>> words.count { "e" in it && "a" in it &&
      "o" in it && "i" in it && "u" in it }
598
```

```
>>> words.find { "e" in it && "a" in it &&
      "o" in it && "i" in it && "u" in it }
aboideau
>>> words.findLast { "e" in it && "a" in it &&
      "o" in it && "i" in it && "u" in it }
warehousing
```

Any and all

These methods return true or false, and implement the exists and for all quantifier:

```
>>> words.all { "qr" !in it }
true
>>> words.all { "qu" !in it }
false
>>> words.any { "qui" in it }
true
>>> words.all { it.length < 25 }
true
>>> words.any { it.length > 21 }
false
>>> words.any { it.length > 20 }
true
```


filter

One of the most useful methods: Returns a new list with the elements for which the predicate is true.

```
>>> words.filter {"e" in it && "a" in it &&
     "u" in it && "i" in it && "o" in it &&
     "y" in it }
[abstemiously, adventitiously, aeronautically,
ambidextrously, ...
```

Combining with other higher-order methods:

```
>>> words.filter {"e" in it && "a" in it &&
    "u" in it && "i" in it && "o" in it &&
    "y" in it }.minBy { it.length }
autotypies
```


filterNot

Can make the program clearer—it reverses the meaning of the predicate.

```
>>> words.filterNot { it.length <= 20 }
[counterdemonstrations, hyperaggressivenesses,
 microminiaturizations]

>>> words.filterNot { "a" in it || "e" in it ||
    "o" in it || "u" in it || "i" in it }
```


The Sieve of Erathosthenes

```
val n = args[0].toInt()
val sqrtn = Math.sqrt(n.toDouble()).toInt()

var s = (2 .. n).toList()

while (s.first() <= sqrtn) {
  val k = s.first()
  print("$k ")
  s = s.filter { it % k != 0 }
}

println(s.joinToString(separator=" "))
```

Transforming a collection

Another very useful tool: Create a new collection from a given one.

```
>>> (1 .. 10).map { it * it }
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> words.map { it.toUpperCase() }.take(10)
[AA, AAH, AAHED, AAHING, AAHS, AAL, AALII, AALIIS,
AALS, AARDVARK]
```


Sorting

Sorting collections is super-useful.

Lists have `sorted()` and `sortedDescending()` methods that return a new list where the elements have been sorted (by their natural order).

Mutable lists also have `sort()` and `sortDescending` methods that sort the elements inside the list.

`sortedBy`, `sortedByDescending`, `sortBy`, and `sortByDescending` allow you to provide a function object to compute the key for sorting.

```
>>> words.sortedByDescending { it.length }.take(5)
[counterdemonstrations, hyperaggressivenesses,
microminiaturizations, counterdemonstration,
counterdemonstrators]
```


Convert collection to map

`groupBy` takes a function object that computes, for each element of the collection, a key. It returns a map that maps this key to the original elements.

```
>>> val m = words.groupBy { it.length }
>>> m[20]
[counterdemonstration, counterdemonstrators,
hypersensitivenesses, microminiaturization,
representativenesses]
```