Lesson 2 - Functions

Create a main() function

main() is the entry point foe execution for a Kotlin progrem

```
fun main(args: Array<String>) {
    println("Hello, world!")

    // if have arguments
    println("Hello, ${args[0]}")
}
```

> The args in the main() function are optional

Everything is an expression and has a value

```
val temperature = 20
val isHot = if (temperature > 40) true else false
println(isHot)
=> false
```

Sometimes, that value is kotlin.Unit

```
val isUnit = println("This is an expression") // println return an unit object
println(isUnit)
=> This is an expression
   kotlin.Unit
```

Functions in Kotlin

- Declared using the fun keyword
- Can take arguments with either named or default values

Unit returning functions

If a function does not return any useful value, its return type is Unit

```
// Writing :Unit is optional
fun printHello(name: String?): Unit {
    println("Hi there!")
}
```

> `Unit` is a type with only one value `Unit`

Function arguments

Functions may have

- Default parameters
- Required parameters
- Named arguments

Default parameters provide a fallback if no parameter value is passed

```
fun drive(speed: String = "fast") {
    println("driving $speed")
}
drive() ⇒driving fast
drive("slow") ⇒driving slowly
drive(speed = "turtle-like") ⇒driving turtle-like
```

Required parameters

If no default is specified for a parameter, the corresponding argument is required

```
fun tempToday(day: String, temp: Int) {
    println("Today is $day and it's $temp degrees")
}
```

Default versus required parameters

Functions can have a mix of default and required parameters

```
fun reformat(str: String,
             divideByCamelHumps: Boolean,
             wordSeparator: Char,
             normalizeCase: Boolean = true) {  }
reformat("Today is a day like no other day", false, '_')
```

Compact functions

Compact functions, or single-expression functions, make your code more concise and readable

```
fun double(x: Int): Int {
    x * 2
}
```

```
fun double(x: Int): Int = x * 2
```

Lambdas and higher-order functions

- Kotlin functions can be stored in variables and data structures
- They can be passed as arguments to, and returned from, other higher-order functions
- You can use higher-order functions to create new "built-in" functions

Lambda functions - an expression that makes a function that has no names

```
var dirtLevel = 20
val waterFilter = { level: Int -> level / 2 }
// { parameter_and_type -> code_to_execute }
println(waterFilter(dirtLevel))
```

Syntax for function types - closely related to its syntax for lambdas. Declare a variable that holds a function

```
val waterFilter: (Int) -> Int = { level -> level / 2 }
val filteredLevel = waterFilter(20)
```

Higher-order functions - take functions as parameters, or return a function

```
fun encodeMsg(msg: String, encode: (String) -> String): String {
    return encode(msg)
}
```

The body of the code calls the function that was passed as the second argument, and passes the first argument along to it.

```
val enc1: (String) -> String = { input -> input.toUpperCase() }
println(encodeMsg("abc", enc1))
```

Pasing a function reference - Use the `::` operator to pass a named function as an argument to another function

```
fun enc2(input: String): String = input.reversed() // declared using fun
encodeMessage("abc", ::enc2)
```

The ::operator lets Kotlin know that you are passing the function reference as an argument, and not trying to call the function.

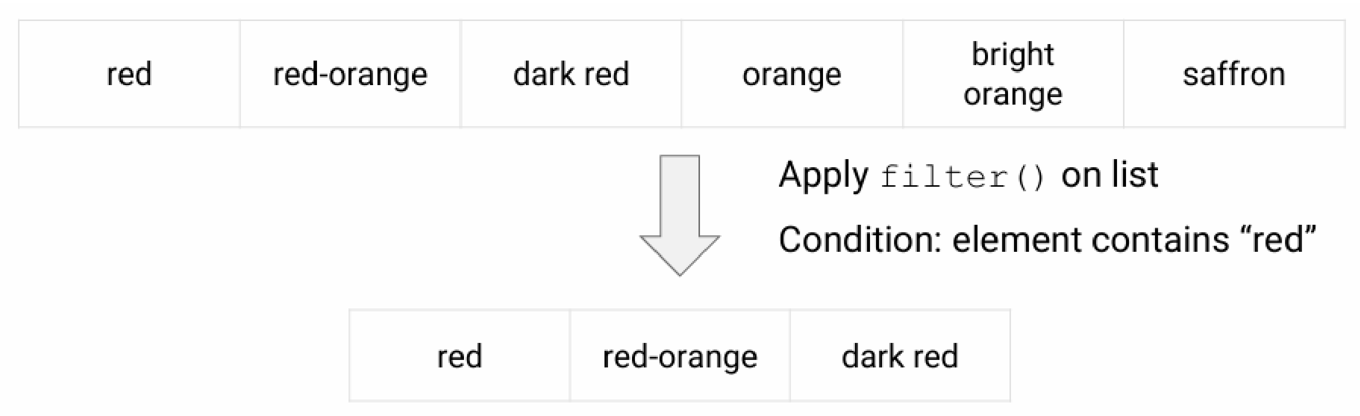
> Kotlin prefers that any parameter that takes a function is the last parameter

```
encodeMessage("acronym", { input -> input.toUpperCase() })
// both work
encodeMsg("acronym") { input -> input.toUpperCase() }
```

Many Kotlin built-in functions are defined using last parameter call syntax

```
inline funrepeat(times: Int, action: (Int) -> Unit)
repeat(3) {
    println("Hello")
}
```

List filters



red	red-orange	dark red	orange	bright orange	saffron

Apply filter() on list

Condition: element contains “red”

red	red-orange	dark red

Iterating through lists

If a function literal has only one parameter, you can omit its declaration and the "->". The parameter is implicitly declared under the name `it`.

```
val ints = listOf(1, 2, 3)
ints.filter { it > 0 }

// equivalent to
ints.filter { n: Int -> n > 0 }
ints.filter { n -> n > 0}
```

If the expression in curly braces `{ }` returns `true`, the item is included

```
valbooks = listOf("nature", "biology","birds")
println(books.filter { it[0] == 'b' })
=> [biology, birds]
```

Eager filters

Eager - occurs regardless of whether the result is ever used

Filters are eager by default. A new list is created each time you use a filter

```
valinstruments = listOf("viola", "cello", "violin")
valeager = instruments.filter { it [0] == 'v'}
println("eager: " + eager)
=> eager: [viola, violin]
```

Lazy filters

Lazy - occurs only if necessary at runtime

Sequences are data structures that use lazy evaluation, and can be used with filters to make them lazy.

```
valinstruments = listOf("viola", "cello", "violin")
valfiltered = instruments.asSequence().filter { it[0] == 'v'}
println("filtered: "+ filtered)
=> filtered: kotlin.sequences.FilteringSequence@386cc1c4
```

Sequences can be turned back into lists using `toList()`.

```
valfiltered = instruments.asSequence().filter { it[0] == 'v'}
valnewList = filtered.toList()
println("new list: " + newList)
=> new list: [viola, violin]
```

Other list transformations

- `map()` performs the same transform on every item and returns the list

  ```
  val numbers = setOf(1, 2, 3) // no dup, immutable
  println(numbers.map { it * 3 })
  => [3, 6, 9]
  ```

- `flatten()` returns a single list of all the elements of nested collections

  ```
  valnumberSets = listOf(setOf(1, 2, 3), setOf(4, 5), setOf(1, 2))
  println(numberSets.flatten())
  => [1, 2, 3, 4, 5, 1, 2]
  ```