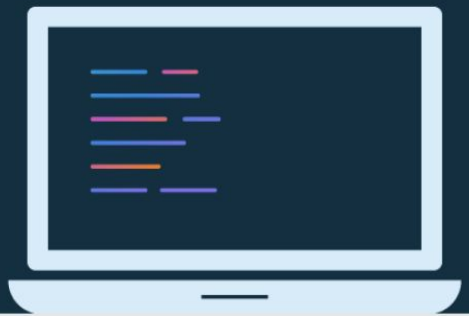
# Lesson 2: Functions



# About this lesson

Lesson 2: Functions

- Programs in Kotlin
- (Almost) Everything has a value
- Functions in Kotlin
- Compact functions
- Lambdas and higher-order functions
- List filters
- Summary

# Programs in Kotlin

# Setting up

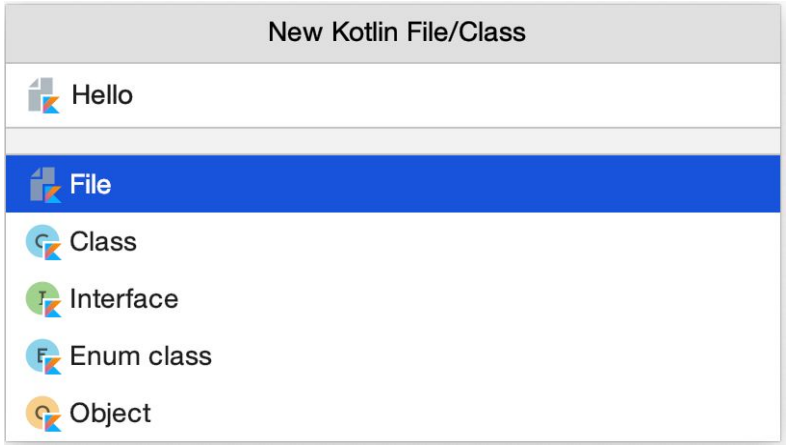
Before you can write code and run programs, you need to:

- Create a file in your project
- Create a `main()` function
- Pass arguments to `main()` (Optional)
- Use any passed arguments in function calls (Optional)
- Run your program

# Create a new Kotlin file

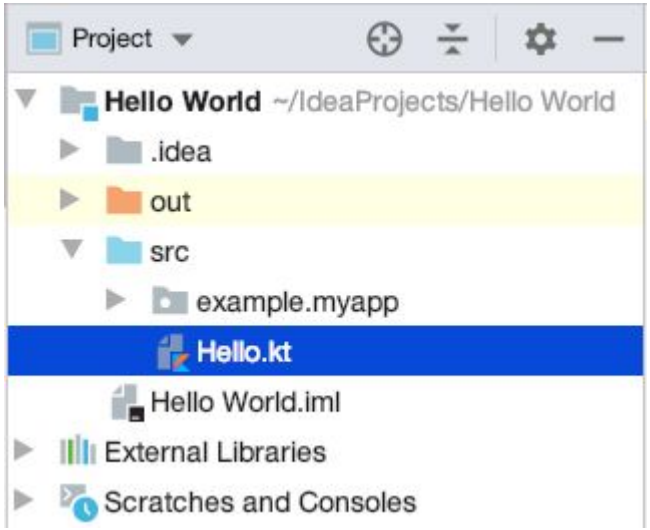
In IntelliJ IDEA's Project pane, under **Hello World**, right-click the `src` folder.

- Select **New > Kotlin File/Class**.
- Select **File**, name the file `Hello,` and press **Enter**.



# Create a Kotlin file

You should now see a file in the `src` folder called `Hello.kt`.



# Create a main() function

`main()` is the entry point for execution for a Kotlin program.

In the `Hello.kt` file:

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

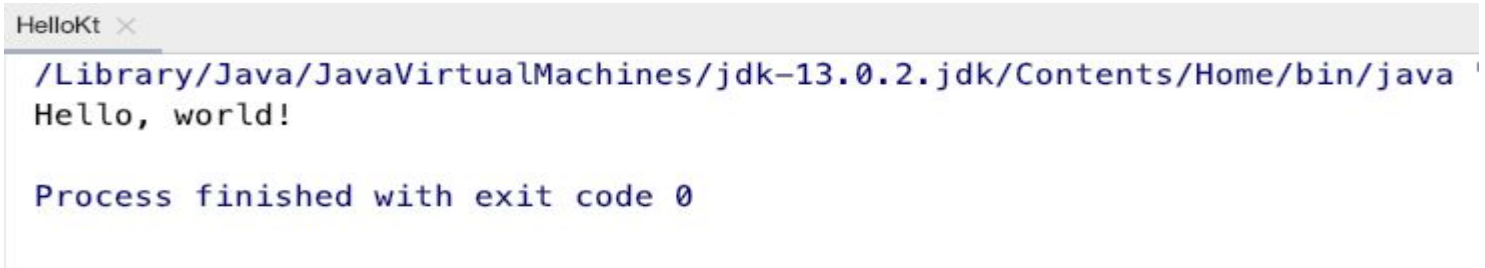
The args in the `main()` function are optional.

# Run your Kotlin program

To run your program, click the Run icon (▶) to the left of the `main()` function.

```
1 ▶ fun main(args: Array<String>) {
2       println("Hello, world!")
3   }
4
```

IntelliJ IDEA runs the program, and displays the results in the console.



# Pass arguments to main()

Select **Run > Edit Configurations** to open the **Run/Debug Configurations** window.

# Use arguments in `main()`

Use `args[0]` to access the first input argument passed to `main()`.

```
fun main(args: Array<String>) {
    println("Hello, ${args[0]}")
}
```

```
⇒ Hello, Kotlin!
```

# (Almost) Everything has a value

# (Almost) Everything is an expression

In Kotlin, almost everything is an expression and has a value. Even an `if` expression has a value.

```
val temperature = 20

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

println(isHot)

⇒ false
```

# Expression values

Sometimes, that value is `kotlin.Unit`.

```
val isUnit = println("This is an expression")

println(isUnit)
```

```
⇒ This is an expression
  kotlin.Unit
```

# Functions in Kotlin

# About functions

- A block of code that performs a specific task
- Breaks a large program into smaller modular chunks
- Declared using the `fun` keyword
- Can take arguments with either named or default values

# Parts of a function

Earlier, you created a simple function that printed "Hello World".

```
fun printHello() {
    println("Hello World")
}

printHello()
```

# Unit returning functions

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

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

`Unit` is a type with only one value: `Unit`.

# Unit returning functions

The `Unit` return type declaration is optional.

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

is equivalent to:

```
fun printHello(name: String?) {
    println("Hi there!")
}
```

# Function arguments

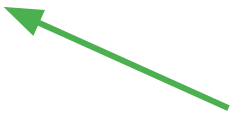
Functions may have:

- Default parameters
- Required parameters
- Named arguments

# Default parameters

Default values provide a fallback if no parameter value is passed.

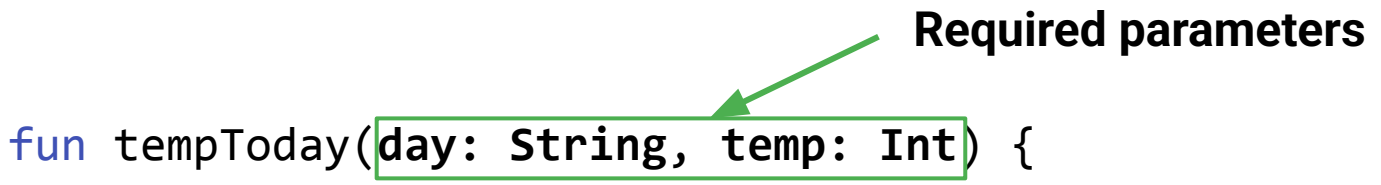
```
fun drive(speed: String = "fast") {
    println("driving $speed")
}

drive() ⇒ driving fast
drive("slowly") ⇒ driving slowly
drive(speed = "turtle-like") ⇒ driving turtle-like
```



**Use "=" after the type to define default values**

# 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){
```


Pass in required arguments.

```
reformat("Today is a day like no other day", false, '_')
```

# Named arguments

To improve readability, use named arguments for required arguments.

```
reformat(str, divideByCamelHumps = false, wordSeparator = '_')
```

It's considered good style to put default arguments after positional arguments, that way callers only have to specify the required arguments.

# Compact functions

# Single-expression functions

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

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

**Complete version**

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

**Compact version**

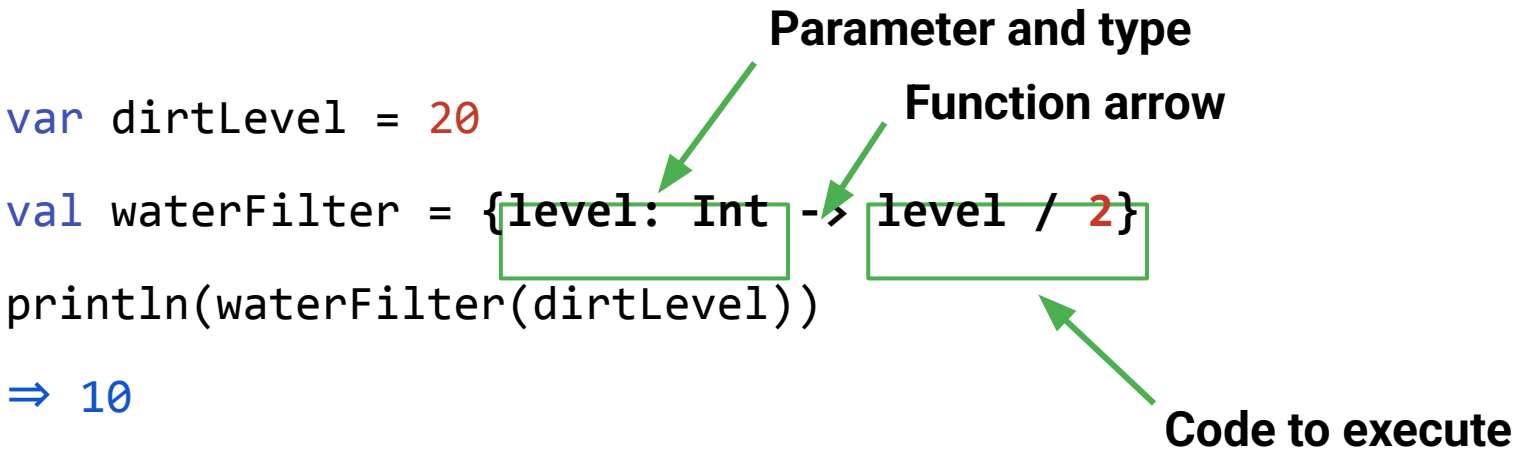
# Lambdas and higher-order functions

# Kotlin functions are first-class

- 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

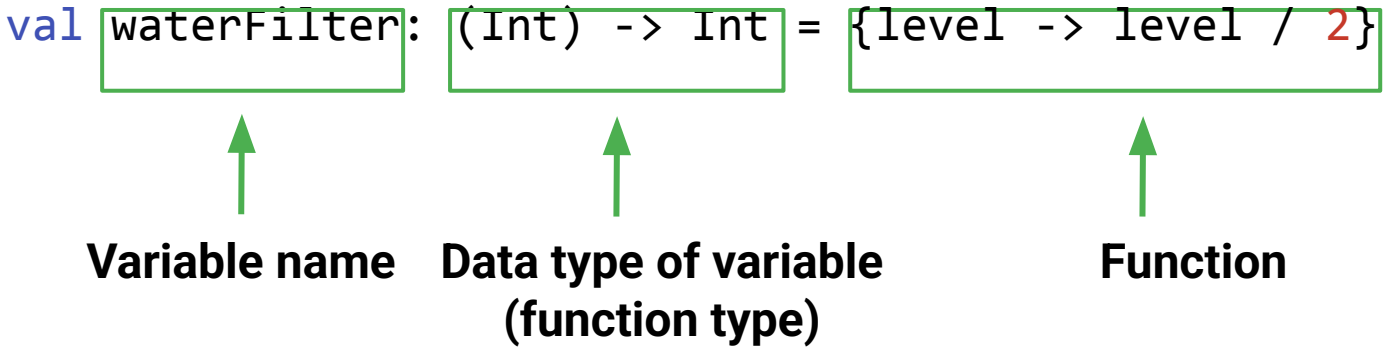
# Lambda functions

A lambda is an expression that makes a function that has no name.



# Syntax for function types

Kotlin's syntax for function types is closely related to its syntax for lambdas. Declare a variable that holds a function.



# Higher-order functions

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.

# Higher-order functions

To call this function, pass it a string and a function.

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

Using a function type separates its implementation from its usage.

https://pl.kotl.in/H3Nhf38bp

# Passing a function reference

Use the `::` operator to pass a named function as an argument to another function.

```
fun enc2(input:String): String = input.reversed()

encodeMessage("abc", ::enc2)
```

**Passing a named function, not a lambda**

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

# Last parameter call syntax

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

```
encodeMessage("acronym", { input -> input.toUpperCase() })
```

You can pass a lambda as a function parameter without putting it inside the parentheses.

```
encodeMsg("acronym") { input -> input.toUpperCase() }
```

# Using higher-order functions

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

```
inline fun repeat(times: Int, action: (Int) -> Unit)
```

```
repeat(3) {
    println("Hello")
}
```

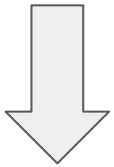
# Kviz

# List filters

# List filters

Get part of a list based on some condition

| 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 }
```

Filter iterates through a collection, where `it` is the value of the element during the iteration. This is equivalent to:

```
ints.filter { n: Int -> n > 0 }   OR   ints.filter { n -> n > 0 }
```

# List filters

The filter condition in curly braces `{}` tests each item as the filter loops through. If the expression returns `true`, the item is included.

```
val books = listOf("nature", "biology", "birds")

println(books.filter { it[0] == 'b' })
```

```
⇒ [biology, birds]
```

# Eager and lazy filters

Evaluation of expressions in lists:

- **Eager:** occurs regardless of whether the result is ever used
- **Lazy:** occurs only if necessary at runtime

Lazy evaluation of lists is useful if you don't need the entire result, or if the list is exceptionally large and multiple copies wouldn't wouldn't fit into RAM.

# Eager filters

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

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

```
⇒ eager: [viola, violin]
```

# Lazy filters

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

```
val instruments = listOf("viola", "cello", "violin")
val filtered = instruments.asSequence().filter { it[0] == 'v'}
println("filtered: " + filtered)
```

```
⇒ filtered: kotlin.sequences.FilteringSequence@386cc1c4
```

# Sequences -> lists

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

```
val filtered = instruments.asSequence().filter { it[0] == 'v'}
val newList = 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)
println(numbers.map { it * 3 })
=> [3, 6, 9]
```

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

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

# Examples

Example1

Example2

Example3

Example4

# Summary

# Summary

In Lesson 2, you learned how to:

- Create a file and a `main()` function in your project, and run a program
- Pass arguments to the `main()` function
- Use the returned value of an expression
- Use default arguments to replace multiple versions of a function
- Use compact functions, to make code more readable
- Use lambdas and higher-order functions
- Use eager and lazy list filters

# Pathway

Practice what you've learned by completing the pathway:

Lesson 2: Functions