

Introduction to Kotlin programming language

Éva Ekler-Antal
eva.ekler.antal@gmail.com

About Kotlin

- Developed mainly by a Russian team working for JetBrains
- Named after Kotlin Island
- Open sourced in 2012
- v1.0 released in 2016
- Google announced support for Kotlin on Android in 2017
- Statically typed language – the types of the variables are known at compile time
- Can be compiled to
 - Java Byte code
 - Javascript
 - Native binaries (embedded devices, iOS)

Contents

1. Why Kotlin?
2. Types and Variables
3. Operators
4. Control Flow Statements
5. Hello, Kotlin! Application
6. Functions
 1. Top Level Functions
 2. Local Functions
 3. Member Functions
 4. Default Arguments
 5. Named arguments
 6. Unit-returning Functions
 7. **Single Expression Functions**
7. **Functions types**
 1. Lambdas
 2. Anonymous Functions
 3. Callable References
 4. Higher Order Functions
 5. Destructuring Declarations
 6. Filters
8. **Classes**
 1. Primary and secondary constructors
 2. **Fields and Properties**
 3. Nested and Inner Classes
9. Packages, Imports
10. Visibility
11. Inheritance
12. Abstract Classes
13. **Interfaces**
14. **Data Classes**
15. **Objects**
16. **Static Methods**
17. **Extension Functions**
18. **Constants**
19. **Infix Functions**

Why is worth to code in Kotlin?

Less code than
using Java

Readable code

Supports both
OO and
functional
programing
styles

Less
NullPointerExceptions
at runtime

No boilerplate
code

You can write
Kotlin code in
Java project as
well

You can
continue to use
your favorite
Java lib

Kotlin Code Editors

- Full support for Kotlin in AndroidStudio, IntelliJ and IntelliJ Community Edition
- REPL (Read Eval Print Loop)
 - Kotlin interpreter
 - Available in Kotlin editors from Tools -> Kotlin -> Kotlin REPL
 - In order to see the evaluated expression, type CTRL + Enter
- If you want to migrate a Java project to Kotlin project, the editors help you with a built in converter tool

Variables

- Two types
 - Changeable, declare it with **var**

```
var a: Int = 10  
a = 20
```

Note: No need for semicolon

```
var a: Int = 10; a = 10
```

Note: Semicolon is used to separate expressions found in one line

- Non-changeable, declare it with **val**

```
val a: Int = 10  
a = 20
```

Compilation error: *Val cannot be reassigned*

Variables – When No Type Specified

```
var a = 10.1
```



The type is inferred by the compiler based on a the context

Types

- Everything is an object -> it is possible to call methods and properties on any variable

```
1.toLong()
```

- Some types (numbers, characters and booleans) are represented at runtime as primitives, but we can see them as ordinary classes

Number Types

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Note: Characters are not numbers like in Java!

Numeric Literals

Type	Example literal
Byte, Short, Int, Long	3, 1_200
Long	3L, 1_200L
Hexadecimal (can be declared as Byte, Short, Int, Long)	0xAF, 0XAF, 0xFF_EC_DE_5E
Binary (can be declared as Byte, Short, Int, Long)	0b0101, 0B0101, 0b01_01
Double	1.23, 1.23e10, 1.2_66
Float	1.2F, 1.2f, 1.2_66f

Casting

- Implicit casting is not allowed (because it is a common source of errors)
 - Ex: you cannot assign a byte value to an int variable

```
val b: Byte = 1
```

```
val c: Int = b
```

Compilation error: *Type mismatch: inferred type is Byte but Int was expected*


- Use explicit casting:

```
val c: Int = b.toInt()
```

Characters

```
var someCharacter: Char = 'a'
```

Character literals go into single quotes



Booleans

```
var someBoolean: Boolean = true  
var otherBoolean = false
```

Two possible values: **true**, **false**



```
someBoolean && otherBoolean || someBoolean && !otherBoolean
```

Built-in boolean operations



Arrays

- Typed arrays

```
var myIntArray = intArrayOf(1, 2, 3)
var myIntArray2 = Array<Int>(3, {it * 1})

println(Arrays.toString(myIntArray))
println(Arrays.toString(myIntArray2))
println(myIntArray[2])
[1, 2, 3]
[0, 1, 2]
3
```

Create an array using utility functions

Create an array with dynamic code using the constructor of the Array class

it is the index of the actual element

Elements can be accessed using the [] operator

- Mixed arrays

```
var mixedArray = arrayOf(1, "something")
println(Arrays.toString(mixedArray))
[1, something]
```

Note: We used the `Arrays.toString` from Java lib

Strings

Strings are represented by the String class

```
var someString: String = String(charArrayOf('m', 'y', ' ', 's', 't', 'r', 'i', 'n', 'g'))
```

Each string is an array of characters

```
var someOtherString = "my string"
```

A convenient way to create string is using string literals

String Templates to Build Strings

- String templates start with \$ sign
- They are evaluated and their value are concatenated in the specified part of the string
- Templates may contain

A single variable

```
val nrOfProgrammers = 10  
print("There are $nrOfProgrammers in the room")
```

There are 10 in the room

An expression

```
val nrOfProgrammers = 10  
val nrOfManagers = 12  
print("There are ${nrOfProgrammers + nrOfManagers} man in the room")
```

There are 22 man in the room

String Literal Types

Escaped strings

- May contain escape characters:
`\t, \b, \n, \r, \', \", \\` and `\$`
- Delimiter: double quote
- Ex:

```
var escapedString = "Hello!\nThis is me!"
```

- May contain templates

Raw strings

- Contain newlines and any other (non-escape) characters
- Delimiter: triple quote

Ex:

```
var rawString = """Hello!
    |This is me!
    """.trimMargin()
```


- Use `trimMargin` to trim leading whitespace characters before the `|` from every line and to remove the first and the last blank lines

```
print(rawString)
Hello!
This is me!
```

- May contain templates

Equals Operator

```
var s1 = "hello"  
var s2 = "hello"  
println(s1 == s2) //outputs: true
```



The “==” operator can be used for any type of object, because it is translated into a call of equals when objects on both sides are not null

Nullability

- By default the value of a variable cannot be null

```
var a: Int = null
```

Compilation error: *Null can not be a value of a non-null type Int*

- The '?' operator indicates that your variable can be null

```
var a: Int? = null
```

- More example

- List of strings where the elements can be null

```
var x: List<String?> = listOf(null, null, null)
```

- Nullable list

```
var x: List<String>? = null
```

- Nullable list with nullable elements

```
var x: List<String?>? = null  
x = listOf(null, null, null)
```

Not Null Assertion Operator

- What happens when we try to call a method on a nullable variable?

```
var someNullableValue: Int? = 7  
someNullableValue.toByteArray()
```

Compilation error: *Only safe (?) or non-null asserted (!!)* calls are allowed on a nullable receiver of type *Int?*

Let's use the **null assertion operator**

```
var someNullableValue: Int? = 7  
someNullableValue!!.toByteArray()
```

Say "double bang"

When we call a function on a null object (using the "double bang"), a `KotlinNullPointerException` will be thrown at runtime

Null Testing Operator

- What happens when we try to call a method on a nullable variable?

```
var someNullableValue: Int? = 7  
someNullableValue.toByteArray()
```

Compilation error: *Only safe (?) or non-null asserted (!!)* calls are allowed on a nullable receiver of type *Int*?

Let's use the **null testing operator**

```
var someNullableValue: Int? = 7  
someNullableValue?.toByteArray()
```

Say "if not null"

`someNullableValue?.toByteArray()` is not invoked if `someNullableValue` is null (Do not confuse with the double bang operator)

The Elvis Operator

- `?:` is called “Elvis” operator, because it looks like the styled hair of Elvis Presley
- if the expression of the left is not null, then uses it, otherwise evaluates the expression on the right

```
fun foo(car: Car) {  
    var extras: MutableList<String> = car.defaultExtras ?: mutableListOf()  
    //some code  
}
```

```
fun bar(car: Car) {  
    var manufacturer: String = car.manufacturer ?: throw IllegalArgumentException("Manufacturer  
cannot be null")  
    //some code  
}
```

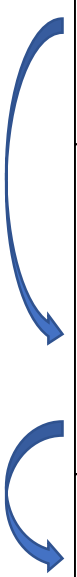
If Statement – in Classical Way

- Can be used as in Java

```
val nrOfPizza = 4
var isEnoughPizza = false
if (nrOfPizza > 10) isEnoughPizza = true
if (isEnoughPizza)
    print("We have enough pizza!")
else
    print("We don't have enough pizza!")
```

If Statement – as Expression

- In Kotlin almost everything has a value (ex exceptions: while and for loops have no return value)
- This means you can use a value of an “if” statement for example



```
val nrOfPizza = 4
var isEnoughPizza = false
if (nrOfPizza > 10) isEnoughPizza = true
if (isEnoughPizza)
    print("We have enough pizza!")
else
    print("We don't have enough pizza!")
```

```
val nrOfPizza = 4
var isEnoughPizza = if (nrOfPizza > 10) true else false
if (isEnoughPizza)
    print("We have enough pizza!")
else
    print("We don't have enough pizza!")
```

```
val nrOfPizza = 4
print("We ${if (nrOfPizza > 10) "have" else "don't have"} enough pizza ")
```


If Statement - Ranges in Conditions

```
val nrOfProgrammers = 10

if (nrOfProgrammers in 1..5)
    print("Too few!")
else
    print("Enough")
```

Enough

Loops

- Basic loop

```
var myArray = arrayOf(10, 20, 30)
for (element in myArray) {
    print("$element ")
}
10 20 30
```

- Loop over the index along with the elements

```
var myArray = arrayOf(10, 20, 30)
for ((index, element) in myArray.withIndex()) {
    println("$element at position $index")
}
10 at position 0
20 at position 1
30 at position 2
```

Loops over Ranges

```
for (i in 'd'..'k') print(i)  
defghijk
```

```
for (i in 0 .. 10) print(i)  
012345678910
```

```
for (i in 10 downTo 1) print(i)  
10987654321
```

```
for (i in 1..10 step 2) print(i)  
13579
```

The **when** Statement

- The **when** automatically breaks, saving the boilerplate code for you

```
val nrOfProgrammers = 10
when (nrOfProgrammers) {
    0 -> print("No meetup")
    in 10..50 -> print("Ok")
    100 -> print ("Full")
    else -> print("What??")
}
Ok
```

The **when** Statement – as Expression

- The ‘when’ statement also has a value: the value of the last expression it was picked

```
fun getFortuneCookie(birthday: Int): String{
    val sentences = arrayOf("You will have a great day!",
        "Things will go well for you today.",
        "Enjoy a wonderful day of success.",
        "Take it easy and enjoy life!")

    return when (birthday){
        in 1993..1998 -> "Hey!!!"
        1990,1992 -> "Not lucky"
        else -> {
            var fortuneIndex = birthday.rem(sentences.size)
            sentences[fortuneIndex]
        }
    }
}
```

The **when** Statement – as Replacement of **if-else** Chain

```
fun whatShouldIDoToday(mood: String, weather: String, temperature: Int): String {  
    return when {  
        mood == "sad" && weather == "rainy" -> "Watch a good film"  
        weather == "sunny" && temperature > 20 -> "Go and play in the garden!"  
        temperature < 5 -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

Note: we can use it without argument

While Loops

- They work as usual

```
var nrOfPizza = 10
while (nrOfPizza > 0) {
    nrOfPizza--
}

var nrOfPizzaOrdered = 0
do {
    nrOfPizzaOrdered++
} while (nrOfPizzaOrdered < 10)
```

Hello, Kotlin! Application

HelloKotlin.kt

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

Note: no need for an enclosing class

Note: starting from Kotlin 1.3 the parameter of the main function can be omitted

Functions

Parameter notation: <name1>: <type1>, <name2>: <type2>, ...

- Functions can be declared using the **fun** keyword

- Example with no return value

```
fun printMessage(message: String) {  
    println(message)  
}
```

- Example with return value

```
fun generateRandomNumber(bound: Int): Int {  
    return Random().nextInt(bound)  
}
```

The type of the return value comes after a colon

- Entry point of Kotlin apps (main function)

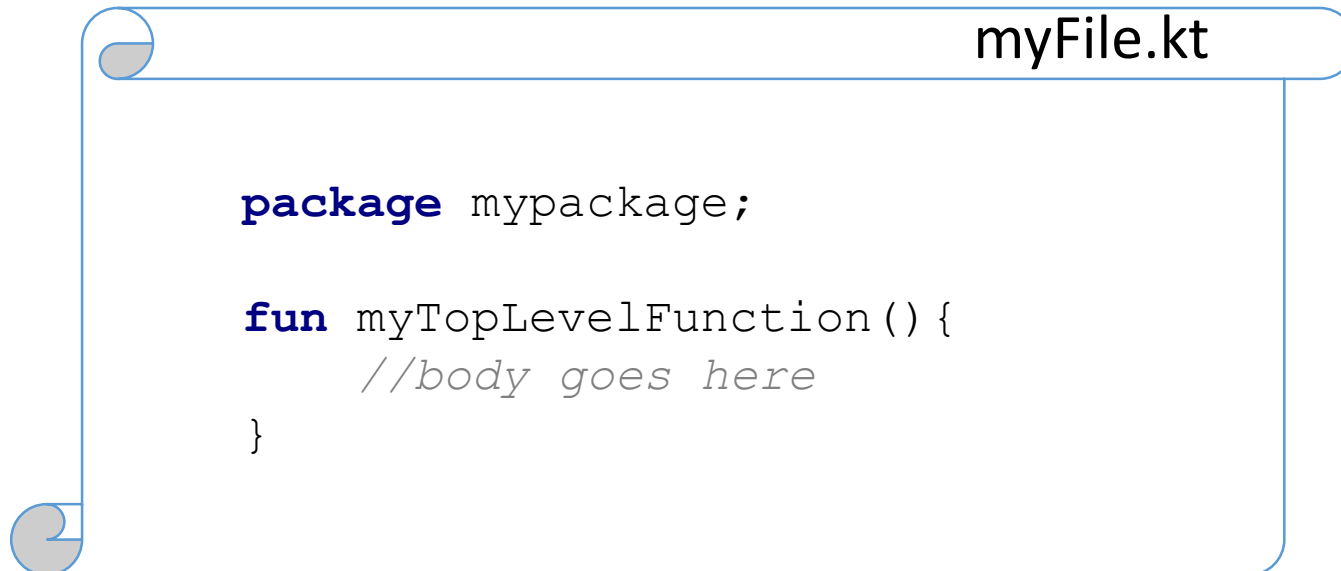
```
fun main(args: Array<String>) {  
    printMessage("Hello, Kotlin!")  
    println("Your random number is: ${generateRandomNumber(50)}")  
}
```

```
Hello, Kotlin!  
Your random number is: 42
```

You can invoke a function even from a string template

Functions – top level functions

- Top level functions
 - Are declared outside a class directly in a file
 - Can be called from main, from other top level functions and from class member functions and init blocks



```
package mypackage;

fun myTopLevelFunction() {
    //body goes here
}
```

Functions – local functions

- Local functions
 - Ex: Function inside another function

```
fun foo () {  
  var a = 4  
  fun bar () {  
    a++  
  }  
}
```

They can access the variables of the outer function

Functions – member functions

- Member functions
 - Defined inside a class
 - Can be called on a class instance using the . operator

```
class Car {  
    fun start() {  
        // ...  
    }  
}
```

```
var car: Car = Car()  
car.start()
```

or

```
Car().start()
```

Functions – default arguments

- The parameters can have default values that are used when the parameter is omitted

```
fun generateRandomNumber(bound: Int = 100): Int {  
    return Random().nextInt(bound)  
}
```

Parameter omitted



```
fun main(args: Array<String>) {  
    println("Your random number is: ${generateRandomNumber(50)}")  
    println("Your second random number is: ${generateRandomNumber()}")  
}
```

Your random number is: 3

Your second random number is: 36

Functions – named arguments

```
fun printRandomNumber(bound: Int = 100, message: String = "Your random number is") {  
    var randomNumber = Random().nextInt(bound)  
    println("$message $randomNumber")  
}
```

```
fun main(args: Array<String>) {  
    printRandomNumber()  
    printRandomNumber(40, "Let's see...")  
    printRandomNumber(bound = 20, message = "Hey...")  
    printRandomNumber(message = "Hm...")  
}
```

Calling the function with
named arguments for
better readability

```
Your random number is 82  
Let's see... 23  
Hey... 18  
Hm... 81
```

Note: when you omit some
default parameter, the
others must be named at
the call

Unit-returning functions

- In Kotlin every function returns something (there is no **void** like in Java)
- If a function returns no useful value, its return type is `Unit`
- The `Unit` type has one value: `Unit`

In case of Unit-returning functions, the type of the return value can be omitted

```
fun printMessage(message: String): Unit {  
    println(message)  
    return Unit //or return  
}
```


The return statement can be also omitted

Single Expression Functions

- If a function returns a single expression, the curly braces can be omitted and the body of the function can be written after = sign
 - The return type can also be omitted if it can be inferred by the compiler

Single expression functions - example (step 1)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        mood == "sad" && weather == "rainy" -> "Watch a good film"  
        weather == "sunny" && temperature > 20 -> "Go and play in the garden!"  
        temperature < 5 -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```



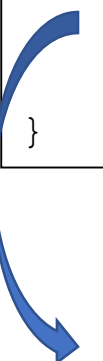
Good code, but let's Kotlinize it!



```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    val isSadAndRainy = mood == "sad" && weather == "rainy"  
    val isSunnyAndWarm = weather == "sunny" && temperature > 20  
    val isCold = temperature < 5  
    return when {  
        isSadAndRainy -> "Watch a good film"  
        isSunnyAndWarm -> "Go and play in the garden!"  
        isCold -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

Single expression functions – example (step 2)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    val isSadAndRainy = mood == "sad" && weather == "rainy"  
    val isSunnyAndWarm = weather == "sunny" && temperature > 20  
    val isCold = temperature < 5  
    return when {  
        isSadAndRainy -> "Watch a good film"  
        isSunnyAndWarm -> "Go and play in the garden!"  
        isCold -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

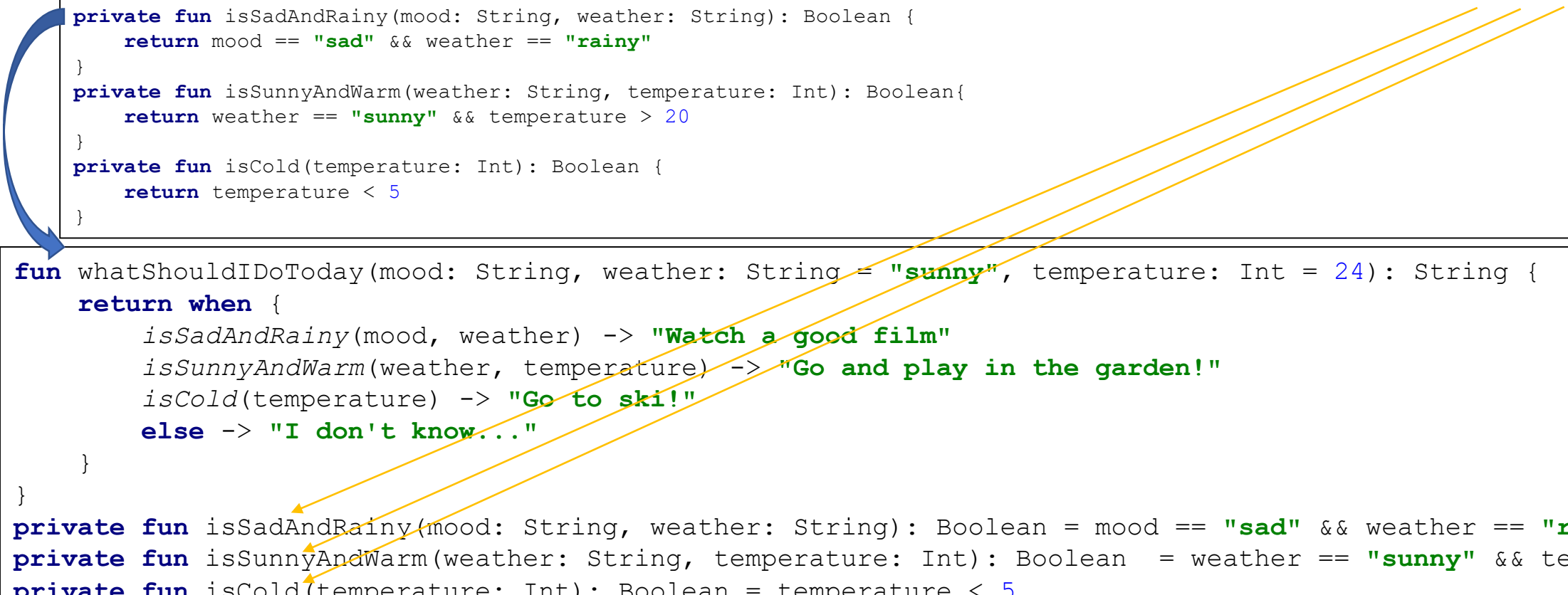


```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
private fun isSadAndRainy(mood: String, weather: String): Boolean {  
    return mood == "sad" && weather == "rainy"  
}  
  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean {  
    return weather == "sunny" && temperature > 20  
}  
  
private fun isCold(temperature: Int): Boolean {  
    return temperature < 5  
}
```

Single expression functions – example (step 3)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
private fun isSadAndRainy(mood: String, weather: String): Boolean {  
    return mood == "sad" && weather == "rainy"  
}  
  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean {  
    return weather == "sunny" && temperature > 20  
}  
  
private fun isCold(temperature: Int): Boolean {  
    return temperature < 5  
}
```

Single expression functions



```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
private fun isSadAndRainy(mood: String, weather: String): Boolean = mood == "sad" && weather == "rainy"  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean = weather == "sunny" && temperature > 20  
private fun isCold(temperature: Int): Boolean = temperature < 5
```

Single expression functions – example (step 4)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
  return when {  
    isSadAndRainy(mood, weather) -> "Watch a good film"  
    isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
    isCold(temperature) -> "Go to ski!"  
    else -> "I don't know..."  
  }  
}
```

Return type can be omitted because it can be inferred by the compiler

```
private fun isSadAndRainy(mood: String, weather: String): Boolean = mood == "sad" && weather == "rainy"  
private fun isSunnyAndWarm(weather: String, temperature: Int): Boolean = weather == "sunny" && temperature > 20  
private fun isCold(temperature: Int): Boolean = temperature < 5
```

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
  return when {  
    isSadAndRainy(mood, weather) -> "Watch a good film"  
    isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
    isCold(temperature) -> "Go to ski!"  
    else -> "I don't know..."  
  }  
}
```

```
private fun isSadAndRainy(mood: String, weather: String) = mood == "sad" && weather == "rainy"  
private fun isSunnyAndWarm(weather: String, temperature: Int) = weather == "sunny" && temperature > 20  
private fun isCold(temperature: Int) = temperature < 5
```

Single expression functions – example (step 5)

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int = 24): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}
```

Default value of parameter
can be provided by a single
expression function

```
fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int =  
    getDefaultTemperature()): String {  
    return when {  
        isSadAndRainy(mood, weather) -> "Watch a good film"  
        isSunnyAndWarm(weather, temperature) -> "Go and play in the garden!"  
        isCold(temperature) -> "Go to ski!"  
        else -> "I don't know..."  
    }  
}  
  
fun getDefaultTemperature() = 24
```

Functions are First-class Citizen

- All functions in Kotlin are first-class, so they can be
 1. Stored in variables
 2. Passed as function arguments
 3. Returned from functions

-> functions need type

Function Types - Examples

```
(Int, String) -> String
```

a function type that has an Int param and a String param, and returns a String

```
(Double) -> Unit
```

a function type that has Double param, and returns nothing useful (Unit)

```
() -> Unit
```

a function type that takes no param, and returns nothing useful (Unit)

```
val foo: () -> Int
```

A variable of which type is a function that has no params, and returns an Int

Instantiating a Function Type

- Three ways
 1. Using a function literal
 1. Lambda
 2. Anonymous function
 2. Using a callable reference to an existing declaration
 1. Reference to a function
 2. Reference to a property
 3. Reference to a constructor
 3. Using instances of a custom class that implements a function type as an interface (NOT COVERED IN THIS PRESENTATION)

Instantiating Function Types Using a Lambda Expression

- Lambdas are function literals (function literals are functions that are not declared but passed immediately as an expression)

```
val square: (Int) -> (Int) = {number: Int -> number * number}
```

The braces around the return type can be omitted

Optional parameter type annotation

Lambda expression (always surrounded by curly braces)

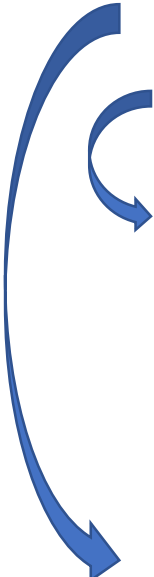
The parameters go before the arrow. If there is no parameter, the arrow can be omitted

The body goes after the arrow. The last expression in the body is treated as the return value

Calling the function type:

```
print(square(4)) // outputs 16
```

Lambda Expressions – Examples



```
val square: (Int) -> (Int) = {number: Int -> number * number}
```

The type of the parameter can be omitted

```
val square: (Int) -> (Int) = {number -> number * number}
```

The type of the variable (of type function) can be omitted if it can be inferred

```
val square = {number: Int -> number * number}
```

Most of the cases using the function type leads to more understandable code because parameter types and return type are expressed explicitly!

Function Types – More Examples

```
val bar: ((Int) -> String) -> Double
```

A variable of type function that has a function param (which takes an Int and returns a String), and returns a Double

```
() -> (Int) -> Unit
```

a function type that takes no param, and returns a function that takes an Int and returns Unit. It is equivalent to:

```
() -> ((Int) -> Unit)
```

and not the same as

```
((Int) -> Unit) -> Unit
```

```
((Float) -> Int)?
```

a function type that has a Float param, and returns an Int, and might be null

The arrow notation is right-associative!

Instantiating Function Types Using an Anonymous Function

- Anonymous functions are like regular functions with no name

```
val square: (Int) -> Int = fun (number: Int): Int { return number * number }
```

Optional parameter type annotation

Anonymous function

The parameters go into brackets just in case of regular functions

The body can be a block or a single expression. In case of block body, the **return** statement must be present.

Calling the function type:

```
print(square(4)) // outputs 16
```

Anonymous Functions - Examples

```
val square: (Int) -> Int = fun(number: Int): Int { return number * number }
```

The type of the parameter can be omitted

```
val square: (Int) -> Int = fun(number): Int { return number * number }
```

The type of the variable (of type function) can be omitted if it can be inferred

```
val square = fun(number: Int): Int { return number * number }
```

Anonymous functions are often more readable than lambdas, because their return type is always expressed explicitly

Lambda vs Anonymous Function

lambda – long version

```
val square: (Int) -> Int = {number: Int -> number * number}
```

Anonymous function – long version

```
val square: (Int) -> Int = fun(number: Int): Int { return number * number }
```

lambda – short version

```
val square = {number: Int -> number * number}
```

Anonymous function – short version

```
val square = fun(number: Int): Int { return number * number }
```

Instantiating Function Types Using a Callable Reference

1. Using a reference to a top-level, local, member or extension function

```
fun calculateSquare(number: Int): Int {  
    return number * number  
}
```

Top-level function

```
val square: (Int) -> Int = ::calculateSquare
```

Reference to a top-level function

```
val absoluteValue: (Int) -> Int = Math::abs
```

Reference to a member function

```
class Foo{  
    var someProperty: Int = 10  
}
```

2. Using a reference to a top-level, member or extension property

A property in class Foo

```
val foo = Foo()
```

```
val someFunctionReturningInt: () -> Int = foo::someProperty
```

Reference to a property of type Int

3. Using a reference to a constructor

```
class Foo
```

```
val someFunctionReturningFoo: () -> Foo = ::Foo
```

Reference to the constructor of Foo

Higher Order Functions

FYI

```
public fun thread: Thread(  
    start: Boolean = true,  
    isDaemon: Boolean = false,  
    contextClassLoader: ClassLoader? = null,  
    name: String? = null,  
    priority: Int = -1,  
    block: () -> Unit  
) { ... }
```

- Higher order functions are functions that take function(s) as parameters, or return function

```
fun downloadProducts(downloadFinished: (List<String>) -> Unit) {  
    thread(block = {  
        val products: List<String> = emptyList()  
  
        //fire http query, add elements to products list  
  
        downloadFinished(products);  
    })  
}
```

Usage:

```
downloadProducts({ products -> print(products) })
```


Higher Order Functions – Kotlinize the Example

```
fun downloadProducts(downloadFinished: (List<String>) -> Unit) {  
    thread(block = {  
        val products: List<String> = emptyList()  
        //fire http query, add elements to products list  
        downloadFinished(products);  
    })  
}  
downloadProducts({ products -> print(products) })
```

If a lambda is the last argument, it can be passed outside the parenthesis

```
fun downloadProducts(downloadFinished: (List<String>) -> Unit) {  
    thread() {  
        val products: List<String> = emptyList()  
        //fire http query, add elements to products list  
        downloadFinished(products);  
    }  
}  
  
downloadProducts() { products -> print(products) }
```

Higher Order Functions – Kotlinize the Example

```
fun downloadProducts(downloadFinished: (List<String>) -> Unit) {  
    thread() {  
        val products: List<String> = emptyList()  
        //fire http query, add elements to products list  
        downloadFinished(products);  
    }  
}
```

```
downloadProducts() { products -> print(products) }
```

If a lambda is passed outside the parenthesis and there are no other passed params, the parenthesis can be omitted

```
fun downloadProducts(downloadFinished: (List<String>) -> Unit) {  
    thread {  
        val products: List<String> = emptyList()  
        //fire http query, add elements to products list  
        downloadFinished(products);  
    }  
}
```

```
downloadProducts { products -> print(products) }
```

Higher Order Functions – Example for Returning a Function

createLogger
returns a function

```
fun createLogger(basicMessage: String) = fun(logMessage: String) {  
    print("$basicMessage $logMessage")  
}  
val logger = createLogger("Some basic message:")  
logger("Some log message") //outputs Some basic message: Some log message
```

More on Anonymous Functions

Are like regular functions with no name

Their parameters must be always specified in parenthesis (unlike in case of lambdas)

Their body can be a block. In that case the return type must be specified (except in case of Unit)

```
val square: (Int) -> Int = fun(number: Int): Int { return number * number }
```

Their body can be a single expression

```
val square: (Int) -> Int = fun(number) = number * number
```

Their parameters' type can be omitted in case they can be inferred


Their return type can be omitted in case of expression body

Lambdas – Implicit Name of the Single Parameter

```
val square: (Int) -> Int = { number -> number * number }
```



```
val square: (Int) -> Int = { it * it }
```



It is allowed not to declare the single parameter of a lambda, it will be implicitly declared with the name **it**

Returning two values from a function – destructuring declarations

The 'Kotlin way' for that is to use data class since they automatically declare componentN() functions needed for destructuring declarations

data class

```
data class HttpQueryResponse(val statusCode: Int, val responseBody: String)

fun requestData(): HttpQueryResponse {
    val statusCode: Int?
    val responseBody: String?

    // request data and set the statusCode and responseBody variables

    return HttpQueryResponse(statusCode, responseBody)
}

val (statusCode, responseBody) = requestData()
```

The return value is decomposed, destructured

Lambdas – Destructuring in Lambdas

- If a lambda has a parameter of the `Pair` type or `Map.Entry`, or any other type that has the appropriate `componentN()` functions, you can introduce several new parameters instead of one by putting them in parentheses

```
fun updateWithNewData(updateUI: (HttpQueryResponse) -> Unit) {  
    val data: HttpQueryResponse = requestData()  
    // some other code  
    updateUI(data)  
}
```

```
updateWithNewData { (code, resp): HttpQueryResponse ->  
    {  
        val someTextview = getTextview()  
        someTextview.setText(resp)  
    }  
}
```

Destructured
HttpQueryResponse as
lambda parameter

FYI

```
data class HttpQueryResponse(val statusCode: Int, val responseBody: String)
```

Lambdas – Underscores for unused variables

```
fun updateWithNewData(updateUI: (HttpQueryResponse) -> Unit) {  
    val data: HttpQueryResponse = requestData()  
    // some other code  
    updateUI(data)  
}
```


```
updateWithNewData { (_, resp): HttpQueryResponse ->  
    {  
        val someTextview = getTextview()  
        someTextview.setText(resp)  
    }  
}
```

Unused lambda parameters can
be replaced by _

Lambdas – Destructuring in Lambdas – More Examples

```
val someMap = mutableMapOf(Pair(1, "A"), Pair(2, "B"))
```

FYI `forEach(action: (Map.Entry<K, V>) → Unit): Unit`



```
someMap.forEach{entry -> println(entry)} //outputs: 1=A 2=B
```

```
someMap.forEach{(key, value) -> println("key: $key, value: $value ")}  
//outputs: key: 1, value: A key: 2, value: B
```

FYI `map(transform: (Map.Entry<K, V>) → R): List<R>`



```
val valuesOfMap: List<String> = someMap.map { entry -> entry.value }
```

```
val valuesOfMap: List<String> = someMap.map { (key, value) -> value }
```

```
val valuesOfMap: MutableCollection<String> = someMap.values
```

Hint: use the **values** property instead



Lambdas – Examples with Collections

```
val someList: List<Int> = listOf(1, 12, 3, 46, 8, 4, 55, 66)
```

Task: Let's select the even numbers, sort them, multiply by 2, and print them out

FYI `filter(predicate: (T) -> Boolean): List<T>`

FYI `sortedBy`: Returns a list of all elements sorted according to natural sort order of the value returned by specified **[selector]** function

```
someList.filter({ item: Int -> item % 2 == 0 }).sortedBy({ item: Int ->
item }).map({ item: Int -> item * 2 }).forEach({ item -> print("$item ") })
// outputs: 8 16 24 92 132
```



We can place the lambda params outside the brackets, brackets not needed, and we can use the implicit `it` param

```
someList.filter{ it % 2 == 0 }.sortedBy{ it }.map{ it * 2 }.forEach{
print("$it ") } // outputs: 8 16 24 92 132
```

Lambdas – More Examples with Collections

Consider the Product class:

```
data class Product(var name: String, var category: String, var price: Double)
```

Let's start with a list of products:

```
var products: List<Product> = mutableListOf(Product("bread", "food", 10.0), Product("milk",  
"food", 9.0), Product("t-shirt", "clothes", 20.0))
```

Task1: retrieve all products of type food sorted by price

```
val foodListByPrice: List<Product> = products.filter { it.category == "food" }.sortedBy {  
it.price }
```

Task2: Print out the products grouped by category

```
products.groupBy { it.category }.forEach { key, value -> print("key: $key, value: $value") }
```

Eager and Lazy Filtering

- Are provided by the standard library
- Filtering can be eager and lazy as well
- Eager filtering example

```
val numbers = listOf(2, 3, 7, 8)
val mappedNumbers = numbers.map { print(it + 1); it + 1 }
3489
```

- Lazy filtering example

```
val numbers = listOf(2, 3, 7, 8)
val mappedNumbers = numbers.asSequence().map { print(it + 1); it + 1 }
//nothing printed out
print(mappedNumbers.toList())
3489[3, 4, 8, 9]
```

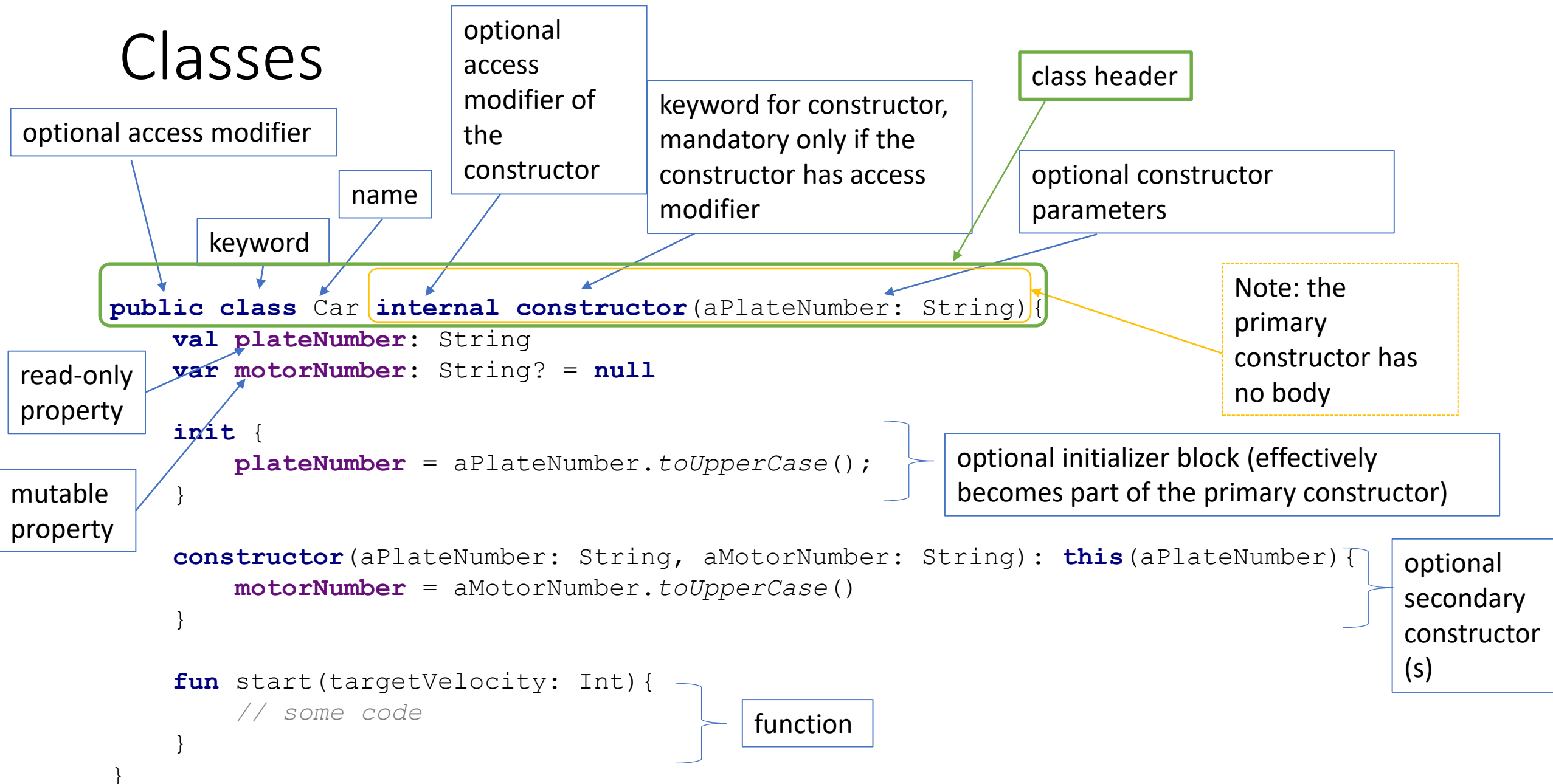
Classes

- Shortest valid class declaration

```
class Car
```

- A class may contain
 - Constructors
 - Properties
 - Init blocks
 - Functions
 - Inner and nested classes
 - Object declarations

Classes



Classes – Equivalent Java Code

```
public class Car internal constructor(aPlateNumber: String){
    val plateNumber: String
    var motorNumber: String? = null

    init {
        plateNumber = aPlateNumber.toUpperCase();
    }

    constructor(aPlateNumber: String, aMotorNumber: String):
    this(aPlateNumber){
        motorNumber = aMotorNumber.toUpperCase()
    }

    fun start(targetVelocity: Int){
        // some code
    }
}
```

Kotlin code

```
public final class Car {
    @NotNull
    private final String plateNumber;

    @NotNull
    public final String getPlateNumber() {
        return this.plateNumber;
    }

    @Nullable
    private String motorNumber;

    @Nullable
    public final String getMotorNumber() {
        return this.motorNumber;
    }

    public final void setMotorNumber(@Nullable
String var1) {
        this.motorNumber = var1;
    }

    public Car(@NotNull String aPlateNumber) {
        super();
        this.plateNumber =
aPlateNumber.toUpperCase();
    }

    public Car(@NotNull String aPlateNumber,
@NotNull String aMotorNumber) {
        this(aPlateNumber);
        this.motorNumber =
aMotorNumber.toUpperCase();
    }

    public final void start(int targetVelocity) {
        // some code
    }
}
```

Corresponding Java code

Constructors

- A class may have one primary constructor and one or more secondary constructors
- The primary constructor is part of the header and has no body

The Primary Constructor - Example

Constructor param can be used in init blocks and property initializers

```
class FibonacciSequence(givenNrOfElements: Int, firstElement: Int = 1) {
```

```
    private val nrOfElements: Int = givenNrOfElements  
    var elements: MutableList<Int> = mutableListOf()
```

```
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2..nrOfElements - 1) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
}
```

The primary constructor cannot contain any code, the initializer code should go into **init blocks**

The init blocks become part of the primary constructor. Init blocks are executed in their declaration order.

```
fun main(args: Array<String>) {  
    val fibonacciSequence = FibonacciSequence(10, 0)  
    print(fibonacciSequence.elements)  
}
```

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

The Primary Constructor – Some Refactor

```
class FibonacciSequence(givenNrOfElements: Int, firstElement: Int = 1) {  
  
    private val nrOfElements: Int = givenNrOfElements  
    var elements: MutableList<Int> = mutableListOf()  
  
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2 until nrOfElements) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
}  
  
fun main(args: Array<String>) {  
    val fibonacciSequence = FibonacciSequence(givenNrOfElements = 10, firstElement = 0)  
    print(fibonacciSequence.elements)  
}
```

Let's use `until` instead of `range` (good choice when you have range ending in `n-1`)

Let's use the variable names at the call to be more expressive and well readable

The Primary Constructor – More Refactor

We can declare and initialize the properties of the class in the primary constructor with **val** or **var** keywords

```
class FibonacciSequence(givenNrOfElements: Int, firstElement: Int = 1) {  
  
    private val nrOfElements: Int = givenNrOfElements  
    var elements: MutableList<Int> = mutableListOf()  
  
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2 until nrOfElements) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
  
    fun main(args: Array<String>) {  
        val fibonacciSequence =  
        FibonacciSequence(givenNrOfElements = 10, firstElement = 0)  
        print(fibonacciSequence.elements)  
    }  
}
```

Note: constructor parameters can have default values

```
class FibonacciSequence(private val nrOfElements: Int,  
firstElement: Int = 1) {  
  
    var elements: MutableList<Int> = mutableListOf()  
  
    init {  
        if (nrOfElements > 0) elements.add(firstElement)  
        if (nrOfElements > 1) elements.add(1)  
        for (i in 2 until nrOfElements) {  
            elements.add(elements[i - 2] + elements[i - 1])  
        }  
    }  
  
    fun main(args: Array<String>) {  
        val fibonacciSequence = FibonacciSequence(nrOfElements = 10,  
firstElement = 0)  
        print(fibonacciSequence.elements)  
    }  
}
```

Correcting the name

Note: we could move the declaration (together with its default value) of the **elements** into the primary constructor

Generated Parameterless Constructor

- If all of the parameters of the primary constructor have default values, the compiler will generate an additional parameterless constructor which will use the default values

```
class Student(name: String = "")  
  
var noNameStudent = Student()  
var student = Student("Aurora Johnson")
```

```
class Student(name: String)  
  
var noNameStudent = Student()  
var student = Student("Aurora Johnson")
```

Compilation error, because there is no parameterless constructor generated

Secondary Constructors - Example

Delegation to the primary constructor

```
class Student(private var firstName: String, private var lastName: String) {  
  
    var id: Int? = null  
    var address: String? = null  
  
    constructor(firstName: String, lastName: String, id: Int) : this(firstName, lastName) {  
        this.id = id  
    }  
  
    constructor(firstName: String, lastName: String, id: Int, address: String) :  
this(firstName, lastName, id) {  
        this.address = address  
    }  
  
}
```

Delegation to the other secondary constructor

Secondary Constructors

- A class may have one or more secondary constructor declared with the **constructor** keyword
- If the class has a primary constructor, each secondary constructor should delegate to it using the **this** keyword (if not directly, then indirectly through another secondary constructor)
 - The init blocks are also executed (through the primary constructor)
 - If there is no primary constructor, the init blocks are also executed because the delegation still happens implicitly
- Important: properties cannot be declared in the parameter list of the secondary constructor

```
class Student(var firstName: String, var lastName: String){  
    constructor(firstName: String, lastName: String, var id: Int): this(firstName, lastName){  
        ...  
    }  
}
```

Secondary Constructors – Let's Kotlinize

```
class Student(private var firstName: String, private var lastName: String) {  
    var id: Int? = null  
    var address: String? = null  
    constructor(firstName: String, lastName: String, id: Int) : this(firstName, lastName) {  
        this.id = id  
    }  
    constructor(firstName: String, lastName: String, id: Int, address: String) : this(firstName, lastName, id) {  
        this.address = address  
    }  
}
```

In most of the cases no need for secondary constructors, a more elegant way is to use default values in the primary constructor

```
class Student(var firstName: String, var lastName: String, var id: Int? = null, var address: String? = null)
```

Usage:

```
var student = Student("Aurora", "Smith")  
var student2 = Student("Aurora", "Smith", 678, "Budapest, Kossuth street 1")  
var student3 = Student("Aurora", "Smith", address = "Budapest, Kossuth street 1")  
var student4 = Student(firstName = "Aurora", lastName = "Smith", id = 678, address = "Budapest, Kossuth street 1")
```

No Constructor

- In case a non-abstract class has neither primary nor secondary constructor, a public primary parameterless constructor will be generated
 - To modify it's visibility declare an empty primary constructor with non-default visibility (default visibility is **public**)

```
class MyClass private constructor () {  
    ...  
}
```


Fields and Properties

- Unlike in Java, declaring directly fields in classes is not possible
- We have properties with setters and getters instead
- When a property needs a backing field, Kotlin automatically generates it

Mutable Properties

We can declare them either in the primary constructor or in the body of the class

Declaration
in class body

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]  
    [<getter>]  
    [<setter>]
```

Example

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
    var name: String  
    get() = "$manufacturer $type"  
    set(value: String) {  
        val splittedName = value.split(" ".toRegex())  
        manufacturer = splittedName[0]  
        type = splittedName[1]  
    }  
  
    private var currentFuelLevel: Int = 0  
}
```

Mutable properties

custom getter

custom setter

the parameter of the setter is
called value by convention (you
can use another name)

The type of the parameter can be
omitted

default getter and setter is
automatically generated

Properties – Invocation of Accessors

```
print(car.name)
```

← When we read a property its getter is called

```
car.name = "Nissan Leaf"
```

← When we change the value of a property its setter is called

- Note: Getter visibility must be always the same as property visibility

Backing Fields

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
  
    var name: String  
        get() = "$manufacturer $type"  
        set(value: String) {  
            val splittedName = value.split(" ".toRegex())  
            manufacturer = splittedName[0]  
            type = splittedName[1]  
        }  
  
    private var currentFuelLevel: Int = 0  
        get() = field  
        set(value) {  
            field = value  
        }  
}
```

← Default getters and setter looks like this

← The backing field can be accessed with the **field** identifier inside the implementation of the accessors

Backing field for properties are automatically generated in the following cases

1. At least one accessor uses the default implementation (at least one accessor has no custom implementation)
2. A custom accessor references the **field** identifier

Backing Fields - Example

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {  
  
    var name: String  
        get() = "$manufacturer $type"  
        set(value: String) {  
            val splittedName = value.split(" ").toRegex()  
            manufacturer = splittedName[0]  
            type = splittedName[1]  
        }  
  
    private var currentFuelLevel: Int = 0  
        set(value) {  
            if (value <= 100)  
                field = value  
            else  
                field = 100  
        }  
}
```

Let's say the **currentFuelLevel** cannot be bigger than 100%

It has default getter

Immutable (Read-Only) Properties

We can declare them either in the primary constructor or in the body of the class

Read-only property declaration in class body

```
val <propertyName>[: <PropertyType>] [= <property_initializer>]  
[<getter>]
```

No setter allowed

```
class Car(val numberPlate: String, var manufacturer: String, var type: String) {
```

```
    var name: String  
        get() = "$manufacturer $type"  
        set(value: String) {  
            val splittedName = value.split(" ".toRegex())  
            manufacturer = splittedName[0]  
            type = splittedName[1]  
        }  
}
```

```
    private var currentFuelLevel: Int = 0  
        set(value) {  
            if (value <= 100)  
                field = value  
            else  
                field = 100  
        }  
}
```

Immutable property

```
    val isFuelLow: Boolean  
        get() = currentFuelLevel < 10  
}
```

No backing field generated as the **field** identifier is not used

Nested Classes

```
class Outer{  
    var someProp = 2;  

```

```
    class Nested{  
        fun someFun () {  
            someProp++  
        }  
    }  
}
```

Classes can be nested into each other

Compilation error: *Unresolved reference someProp*
(Members of the outer class can **NOT** be reached)

```
Outer.Nested().someFun()
```

Instantiating the nested class and invoking its method

Inner Classes

```
class Outer{  
    var someProp = 2;  
  
    inner class Nested{  
        fun someFun() {  
            someProp++  
            this@Outer.someProp++  
        }  
    }  
}  
  
Outer().Nested().someFun()
```

If a nested class is marked as **inner** it can access the members of its outer class

No compilation error

(Members of the outer class can be reached)

The instance of the outer class can be accessed using a qualified (labeled) **this@Outer**

In order to instantiate the inner class we need an instance of the outer class

No difference between the two lines

Packages

- We organize Kotlin files into packages, just like in Java

```
package mydomain.myapp  
  
private fun someTopLevelFunction() {}  
  
class SomeTopLevelClass
```

A source file may start with a package declaration

The full name of `someTopLevelFunction` is `mydomain.myapp.someTopLevelFunction`

The full name of `SomeTopLevelClass` is `mydomain.myapp.SomeTopLevelClass`

- If the package is not specified in the file, the contents belong to "default" package that has no name

Imports

- We can import
 - Classes
 - Top level properties
 - Functions and properties declared in object declarations
 - Enum constants

Note: There is no static-import like in Java

```
import somepackage.SomeTopLevelClass
import somepackage.SomeTopLevelClass.SomeInnerClass
import somepackage.someTopLevelFunction
import someotherpackage.someTopLevelFunction as someAlias
import somepackage.*
```

Default Imports

- The following packages are imported by default in each Kotlin file
 - `kotlin.*`
 - `kotlin.annotation.*`
 - `kotlin.collections.*`
 - `kotlin.comparisons.*` (since 1.1)
 - `kotlin.io.*`
 - `kotlin.ranges.*`
 - `kotlin.sequences.*`
 - `kotlin.text.*`
 - `java.lang.*`
 - `kotlin.jvm.*`

Visibility

- Classes, objects, interfaces, constructors, functions, properties (and their setters) can have visibility modifiers
 - Getters always have the same visibility as the property
- Visibility modifiers
 - Public (visibility from everywhere)
 - Private (visibility inside the file of declaration)
 - Internal (visibility from the same module)
 - Protected (not visible for top level declarations)
- Default visibility is public (even if you don't specify it explicitly)
- In order to use a visible top level declaration from another package, you should import it

Top level declarations and their visibility

- Can be declared on the top level (for example in a .kt file directly in a package):
 - Functions
 - Properties
 - Classes
 - Objects
 - Interfaces
- Visibility of top level declarations
 - Public (visibility from everywhere)
 - Private (visibility inside the file of declaration)
 - Internal (visibility from the same module)
 - Protected (can't be used for top level declarations)

```
package topleveldeclarations

private fun someTopLevelFunction() {}

internal var someProperty: String = "hello"
    private set

class SomeTopLevelClass

internal object SomeTopLevelObject

interface SomeTopLevelInterface
```

Visibility of the members of classes and interfaces

- Available access modifiers
 - Private – visible inside the class
 - Protected – visible inside the class and subclasses
 - Internal – anyone in the same module who sees the class sees its' internal members
 - Public – anyone from anywhere who sees the class sees its' public members
- **NOTE:** if you override a protected member without specifying an access modifier, the override member will be also protected
- **NOTE** for Java users: outer class does not see private members of its inner classes in Kotlin

Modules

- A module is a set of Kotlin files compiled together
 - An IntelliJ IDEA module
 - A Maven project
 - A Gradle source set (with the exception that the test source can access the internal declarations of main)
 - A set of files compiled with one invocations of the Ant task

Inheritance

By default classes are NOT subclassable, to make it so use the **open** keyword

By default member functions can NOT be overridden, to make it so use the **open** keyword

Use the **override** keyword to override a member

The Button class is the subclass of the View class, the View is the superclass of the Button class

If the derived class has a primary constructor, the constructor of the base class must be called

Calling the `onClick()` of the superclass

```
open class View(var id: Int) {
    open fun onClick() {
        //some code
    }
}

class Button(id: Int, var color: Int): View(id) {
    private val silver = 0xC0C0C0

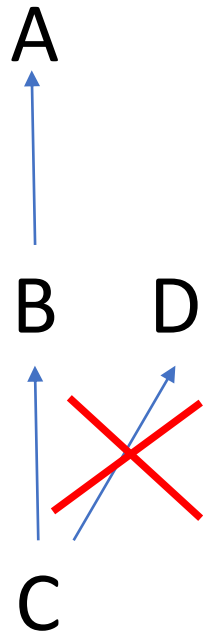
    override fun onClick() {
        super.onClick()
        color = silver
    }
}
```

```
var loginButton: View = Button(1, 0xAAAAAA)
loginButton.click()
```

Example usage

Inheritance – One Direct Superclass

- A class may have multiple indirect superclasses, but only one direct superclass



C derives from B, B derives from A, so A is an indirect superclass of C

Inheritance – Prohibit Overriding

The `click` is
open both in
View and
Button

Use the **`final`**
keyword to
prohibit overriding
an **`open`** member

Compilation error:
*click in Button is
final and cannot
be overridden*

```
open class View(var id: Int) {  
    open fun click() {  
        //some code  
    }  
}  
  
open class Button(id: Int) : View(id) {  
    final override fun click() {  
        // some code  
    }  
}  
  
open class ImageButton(id: Int) : Button(id) {  
    override fun click() {  
    }  
}
```

Inheritance – when a Subclass has no Primary Constructor

```
open class View(var id: Int) {  
    open fun onClick() {  
        //some code  
    }  
}
```

```
open class Button : View {
```

```
    var color: Int
```

```
    private val defaultColor: Int = 0x000000
```

```
    constructor(id: Int): super(id){  
        this.color = defaultColor  
    }
```

```
    constructor(id: Int, color: Int): this(id){  
        this.color = color  
    }
```

```
}
```

The superclass has to be initialized using the **super** keyword to call the constructor of the base class

Or the superclass has to be initialized using the **this** keyword to delegate to other constructor that calls the constructor of the base class

Inheritance – Overriding Properties

Properties can be overridden, even a **val** to a **var** (Note: a **var** cannot be overridden to **val**)

Properties can be also overridden by overriding their getter

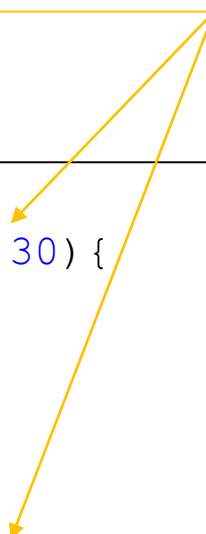
Note: a getter can be a single-expression function or a function with body as well

```
open class View(var id: Int) {  
    open val color: Int = 0x000000  
  
    open val tooltipText: String  
        get() = "This is a tooltip text of view: $id"  
}  
  
open class Button(id: Int, override var color: Int = 0xFFFFFFFF, var text: String) : View(id) {  
    override val tooltipText: String  
        get() {  
            return text  
        }  
}
```

Inheritance - Default Parameter Values of Overridden Methods

Overridden methods use the same default values as the base method, they are not allowed to provide any default values to their parameters

```
open class View(var id: Int) {  
    open fun rotateWithAngle(angle: Int = 30) {  
        //some code  
    }  
}  
  
open class Button(id: Int) : View(id) {  
    override fun rotateWithAngle(angle: Int) {  
        //some other code  
    }  
}
```



Abstract Classes

If a class contains abstract member(s) it must be marked as **abstract**

```
abstract class View(var id: Int) {  
    abstract val color: Int  
    abstract fun click()  
}  
  
class Button(id: Int) : View(id) {  
    override val color: Int = 0x000000  
  
    override fun click() {  
        // some code  
    }  
}
```

abstract members do not have implementation and the **open** keyword is not necessary

abstract member implementations

Abstract Classes – Overriding a non-abstract member with an abstract one

```
open class View(var id: Int) {  
    open fun click() {  
        //some code  
    }  
}  
  
abstract open class Button(id: Int) : View(id) {  
    override abstract fun click()  
}
```

It is possible to override a non-abstract member with an abstract one!

Interfaces – What can they contain?

We declare an interface with the **interface** keyword

An interface may contain properties that don't use backing field (not even generated ones)

An interface may contain abstract properties (the **abstract** keyword is not necessary)

Yet another example for a property that does not have a backing field

An interface may contain abstract functions (the **abstract** keyword is not necessary)

An interface may contain functions with implementation

```
interface HttpRequest {  
    val baseUrl: String  
    get() = "https://myapp.com/api"  
  
    var apiUrl: String  
  
    val fullUrl: String  
    get() = baseUrl + apiUrl  
  
    fun send()  
  
    fun onError() {  
        //some code  
    }  
}
```


Interfaces – Implementations

The abstract members must be implemented

```
interface HttpRequest {  
    val baseUrl: String  
    get() = "https://myapp.com/api"  
  
    var apiUrl: String  
  
    val fullUrl: String  
    get() = baseUrl + apiUrl  
  
    fun send()  
  
    fun onError() {  
        //some code  
    }  
}
```

```
class LoginHttpRequest : HttpRequest {  
    override var apiUrl: String = "/login"  
  
    override fun send() {  
        // some code  
    }  
}
```

```
fun foo(request: HttpRequest) {  
    //some code  
}  
  
fun bar() {  
    val loginHttpRequest = LoginHttpRequest()  
    foo(loginHttpRequest)  
}
```

Example usage

Inheritance – Implementing Multiple Interfaces

- A class may implement more interfaces

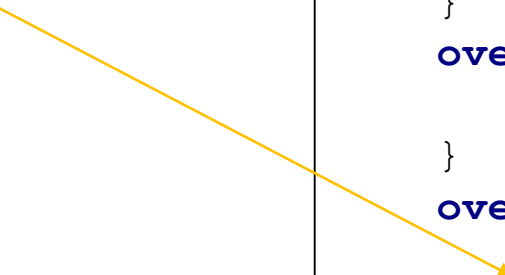


C implements A and B

```
interface A {  
    fun a()  
}  
  
interface B {  
    fun b()  
}  
  
class C : A, B {  
    override fun a() {  
        //some code  
    }  
  
    override fun b() {  
        //some code  
    }  
}
```

Overriding Rules in Case of Inherited Members with the Same Name

When a class wants to call one of the implementation of some member from their supertypes with the same name, it must specify the supertype's name along with the member's name



```
open class A {  
    open fun a() {}  
    open fun foo() {print("A.foo")}  
}  
  
interface B {  
    fun b()  
    fun foo() {print("B.foo")}  
}  
  
class C : A(), B {  
    override fun a() {  
        //some code  
    }  
    override fun b() {  
        //some code  
    }  
    override fun foo() {  
        super<A>.foo()  
        //some code  
    }  
}
```

Interfaces Inheritance

Interfaces may extend each other

The child interface may implement to inherited members

All (directly or indirectly) inherited members without implementation must be implemented by the child class

```
interface A {  
    fun a()  
    fun aa()  
    fun aaa()  
}  
  
interface B : A {  
    fun b()  
    override fun aa() {  
        //some code  
    }  
}  
  
class C : B {  
    override fun a() {  
        //some code  
    }  
  
    override fun aaa() {  
        //some code  
    }  
  
    override fun b() {  
        //some code  
    }  
}
```

Abstract classes vs interfaces

Abstrac class

- Can have constructor
- Can contain implemented methods
- Cannot be instantiated
- Can contain properties with backing field (so it can store changing state)

Interface

- Cannot have constructor
- Can contain implemented methods
- Cannot instantiated
- Can contain properties without backing fields (it cannot store changing state)

Implementing a Java Interface

Let's have a Java interface in a Java file

```
public interface MyJavaInterface {  
    void interfaceMethod(Integer someParameter);  
}
```

Let's have a Java method using that interface in another Java file

```
public class MyInterfaceUser {  
    void someMethod(MyJavaInterface myJavaInterface) {  
        //some code  
    }  
}
```

Let's have a Kotlin variable of type MyInterfaceUser in a Kotlin file

```
var myInterfaceUser: MyInterfaceUser = MyInterfaceUser()
```

Let's call the interface's method and pass an implementation of the interface using a lambda

```
myInterfaceUser.someMethod({ someParameter: Int -> print("hey") })
```

For better readability we can specify explicitly the type of the implemented interface

```
myInterfaceUser.someMethod(MyJavaInterface { someParameter: Int -> print("hey") })
```

Or we can implement the interface as an **object expression** (similar to anonymous inner classes in Java)

```
myInterfaceUser.someMethod(object : MyJavaInterface {  
    override fun interfaceMethod(someParameter: Int?) {  
        //some code  
    }  
})
```

Data Classes

```
data class Product(var name: String, var price: Double)
```

- Their main purpose is to hold data
- The compiler automatically generates the following members using the properties declared in the primary constructor only
 - `equals()` and `hashCode()`
 - `toString()` (ex: **Product(name=bread, price=10.0)**)
 - `componentN()` functions corresponding to the properties in their declaration order (useful for destructuring declarations)
 - `copy()` function

```
var bread = Product("bread", 10.0)  
var homeMadeBread = bread.copy(name = "home made bread")
```

- built-in data classes: `Pair`, `Triple`

Object Declarations - Classes with Single Instance

To declare a singleton, use an **object declaration**

```
object User {  
    var username: String? = null  
    var password: String? = null  
  
    fun login(username: String, pwd: String) {  
        this.username = username  
        this.password = pwd  
    }  
  
    fun logout() {  
        username = null  
        password = null  
    }  
}
```

In order to refer an object, use its name

Note: There will be only one instance of the User 'class' in the JVM

```
fun someFunction() {  
    User.login("eva", "1234")  
}  
  
fun someOtherFunction() {  
    getPersonalizedAds(User.username)  
}
```

Example usage

Objects – They can have supertypes

```
interface Person{
    val name: String?
}

object User: Person{
    var username: String? = null
    var password: String? = null
    override val name: String? = username

    fun login(username: String, pwd: String) {
        this.username = username
        this.password = pwd
    }

    fun logout() {
        username = null
        password = null
    }
}
```

Objects can have supertypes (can implement interface or extend a class)

Objects – Notes

- No way to make another instance of the objects
- Their initialization is thread-safe
- Object declarations are not expressions, so they cannot be on the right hand side of a declaration (ex: cannot be assigned to a variable)

Object Declarations vs Object Expressions

Object Declaration

- They are initialized **lazily**, when accessed for the first time

```
object SomeSingleton {  
    val someProperty = 1  
}
```

Object Expression

- They are executed (and initialized) **immediately**, where they are used

```
fun bar() {  
    foo(object: SomeInterface {  
        override fun someInterfaceMethod() {  
            // ...  
        }  
    })  
}
```

We can create an object of an anonymous class

```
fun bar() {  
    val someObject = object {  
        val city = "Budapest"  
        val country = "Hungary"  
    }  
    print("$someObject.city is in ${someObject.country}")  
}
```

We can create just an object

Companion Objects

```
class SomeClass{  
    companion object SomeComapnionObject{  
        fun foo() {  
            // ...  
        }  
    }  
}
```

Valid usages:

```
val foo1 = SomeClass.SomeComapnionObject.foo()  
val foo2 = SomeClass.foo()
```

- a companion object is initialized when the corresponding class is loaded (resolved), matching the semantics of a Java static initializer
- at runtime they are instance members of real objects

Static Methods

- Kotlin represents package-level functions as static methods
- Kotlin can also generate static methods for functions defined in named objects or companion objects if you annotate those functions as **@JvmStatic**

```
class SomeClass{  
    companion object SomeComapnionObject{  
        @JvmStatic  
        fun foo(){  
            // ...  
        }  
    }  
}
```

If you use this annotation, the compiler will generate both a static method in the enclosing class of the object and an instance method in the object itself

- **@JvmStatic** annotation can also be applied on a property of an object or a companion object making its getter and setter methods be static members in that object or the class containing the companion object

Extension Functions

- They allow adding new functionality to any (even JDK) class without subclassing from the class

```
val someList: List<Int> = listOf(1, 12, 3, 46, 8)
```

```
fun List<Int>.prettyPrint() {  
    println("#####this#####")  
}
```

```
someList.prettyPrint() // outputs: #####[1, 12, 3, 46, 8]#####
```

Extension function to List<Int>

The **this** points to the receiver object (in this case someList)

Extension Functions – Generic Extension Functions

- Let's make the previous extension function generic, so that it will be callable on any List<T>

```
fun <T> List<T>.prettyPrint() {  
    println("#####$this#####")  
}
```

```
val someStringList = listOf<String>("one", "two")
```

```
someStringList.prettyPrint() //outputs: #####[one, two]#####
```

Extension function to List<T>

Extension Functions – More Examples

- A useful built-in extension function

```
val someStringList = listOf<String>("one", "two")  
  
someStringList.get(0).also { print("The receiver is: $it") } // outputs: The receiver is:  
one
```

Can be invoked on any object

FYI

```
public inline fun <T> T.also(block: (T) -> Unit): T {  
    contract {  
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
    }  
    block(this)  
    return this  
}
```


Compile-Time Constants

- Read-only properties with value known at compile time can be marked as compile time constants using the `const` keyword
- Requirements
 - Top level property or member of an object (not class!)
 - Initialized with a String or primitive type value
 - No custom getter

```
const val DEFAULT_LOG_LEVEL = "Info"

object SomeObject{
    const val DEFAULT_LOG_LEVEL = "Info"
}
```

Note: no compile time constants allowed in classes!

Infix Functions

- Declared with the **`infix`** keyword
- They must satisfy the following:
 - They are member functions or extensions
 - They have a single parameter
 - Their parameter is not marked with **`vararg`** and has no default value

The infix functions can be called without using the `.` and parenthesis

```
class Student() {  
  
    private var competencies: MutableList<String> = mutableListOf()  
  
    init {  
        this.addCompetence("Kotlin") //compiles  
        addCompetence("Android") //compiles  
        this addCompetence "Java" //compiles  
        addCompetence "Scala" //compilation error, missing receiver  
    }  
  
    infix fun addCompetence(competence: String) {  
        competencies.add(competence)  
    }  
}
```

```
fun main(args: Array<String>) {  
    var student = Student()  
    student.addCompetence("Javascript") //compiles  
    student addCompetence "HTML" //compiles  
}
```

Kotlin Documentation

- Official documentation: <http://kotlinlang.org/docs/reference/>

Bibliography

- <https://kotlinlang.org>
- [About first-class functions](#)

Explore More

- Read about
 - Type Aliases
 - Operator Overloading
 - Inline Functions
 - Enums
 - Sealed Classes
 - Delegation
 - Coroutines
- Subscribe to Android Weekly mails
- More on Types: <https://www.kotlinderdevelopment.com/typical-kotlin/>