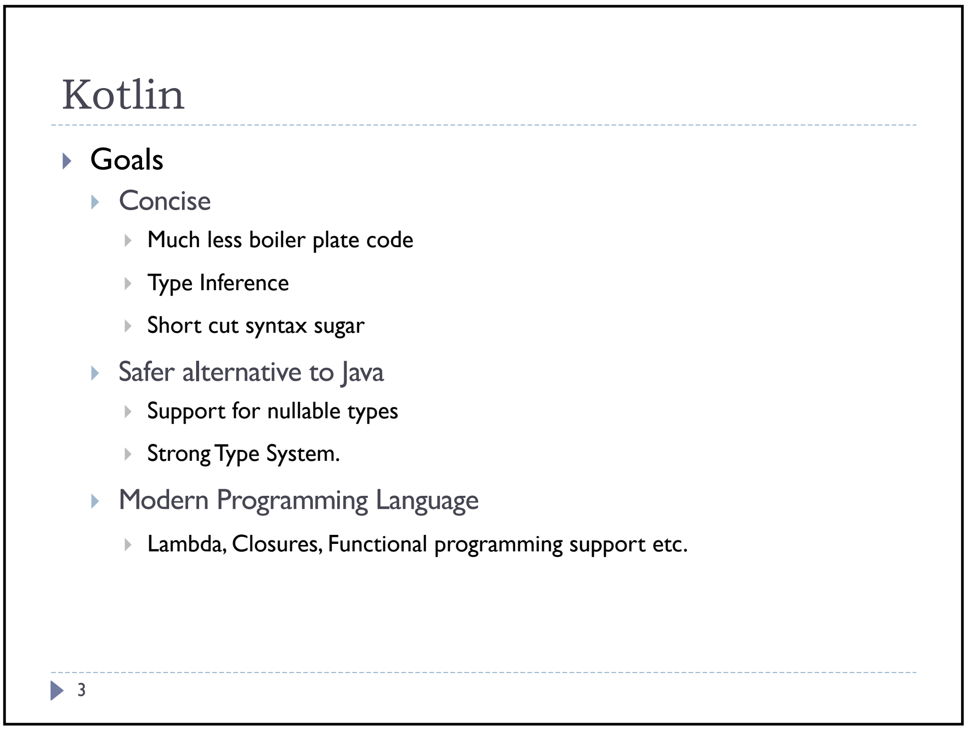
Kotlin

amit.gulati@gmail.com

Kotlin

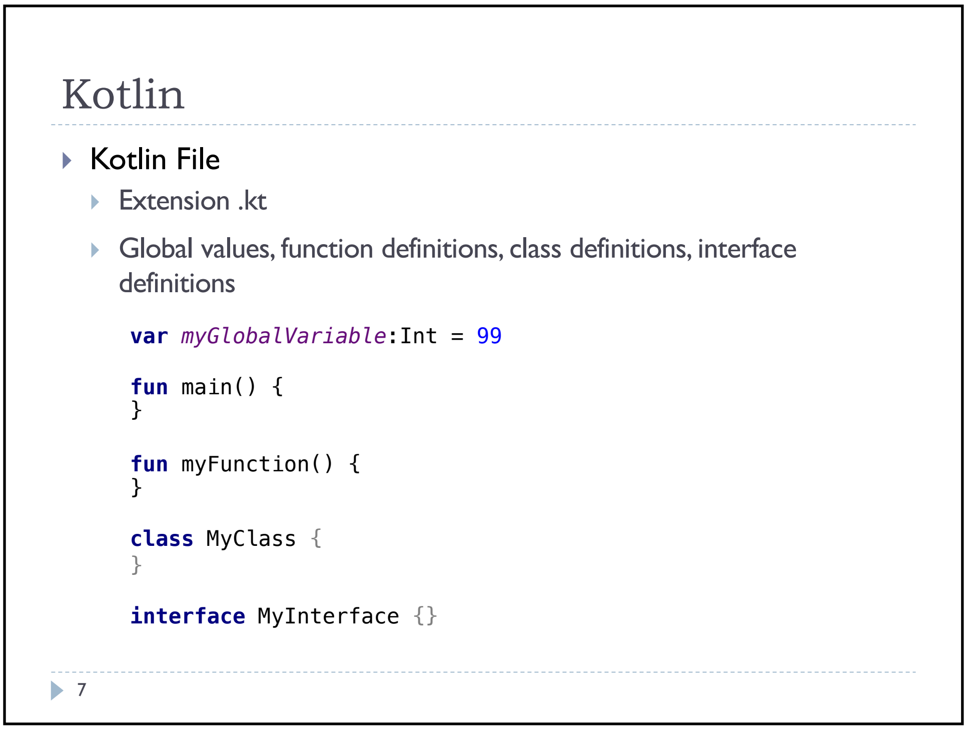
- What is Kotlin?
  - Programming Language by Jetbrains
  - 1.0 Officially announced in 2016
  - Open Source
  - Inspired by Java, Scala, Groovy, C# etc.
  - Fully InterOp with Java



Kotlin

- Goals
  - Concise
    - Much less boiler plate code
    - Type Inference
    - Short cut syntax sugar
  - Safer alternative to Java
    - Support for nullable types
    - Strong Type System.
  - Modern Programming Language
    - Lambda, Closures, Functional programming support etc.

3

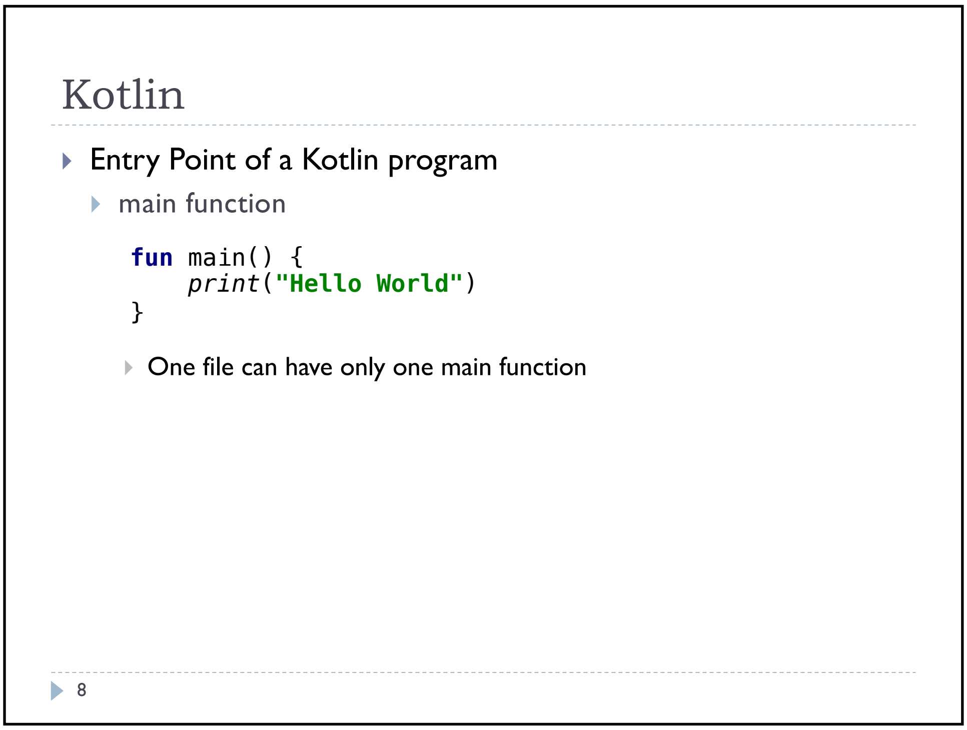


Kotlin

- Kotlin File
  - Extension .kt
  - Global values, function definitions, class definitions, interface definitions

```
var myGlobalVariable:Int = 99

fun main() {
}

fun myFunction() {
}

class MyClass {
}

interface MyInterface {}
```

7

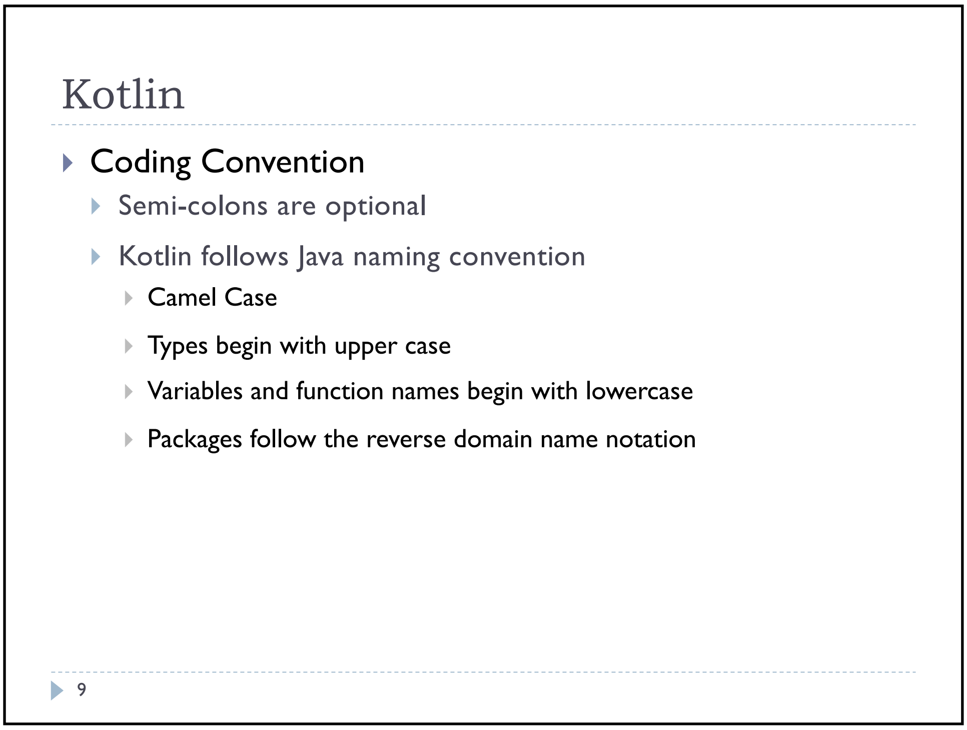


Kotlin

- Entry Point of a Kotlin program
  - main function

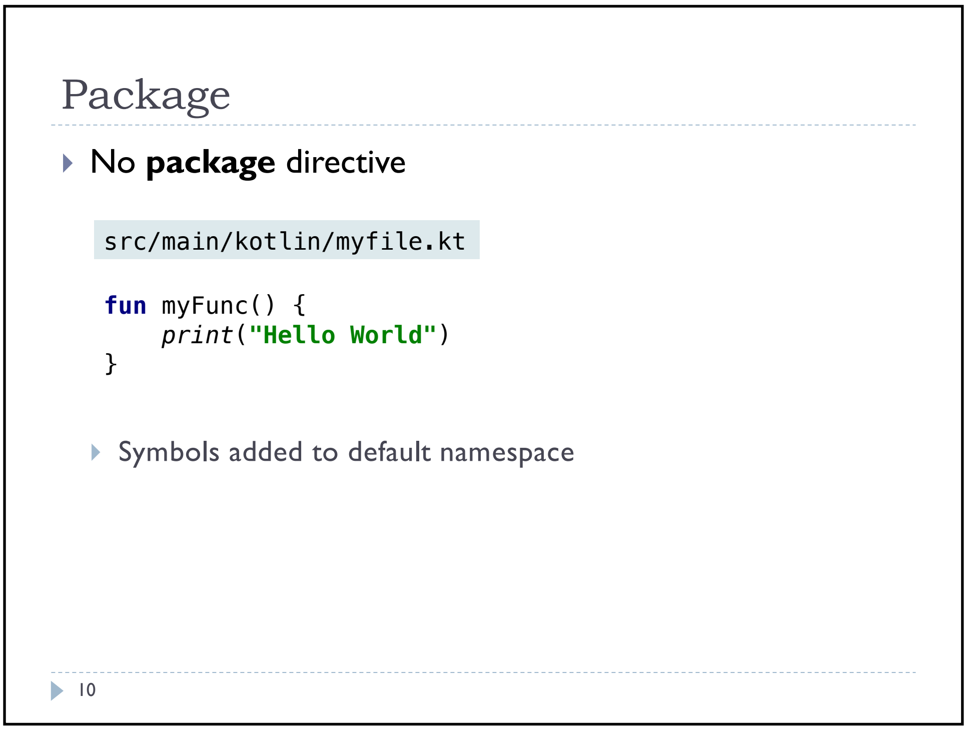
```
fun main() {
    print("Hello World")
}
```

    - One file can have only one main function



Kotlin

- Coding Convention
  - Semi-colons are optional
  - Kotlin follows Java naming convention
    - Camel Case
    - Types begin with upper case
    - Variables and function names begin with lowercase
    - Packages follow the reverse domain name notation



Package

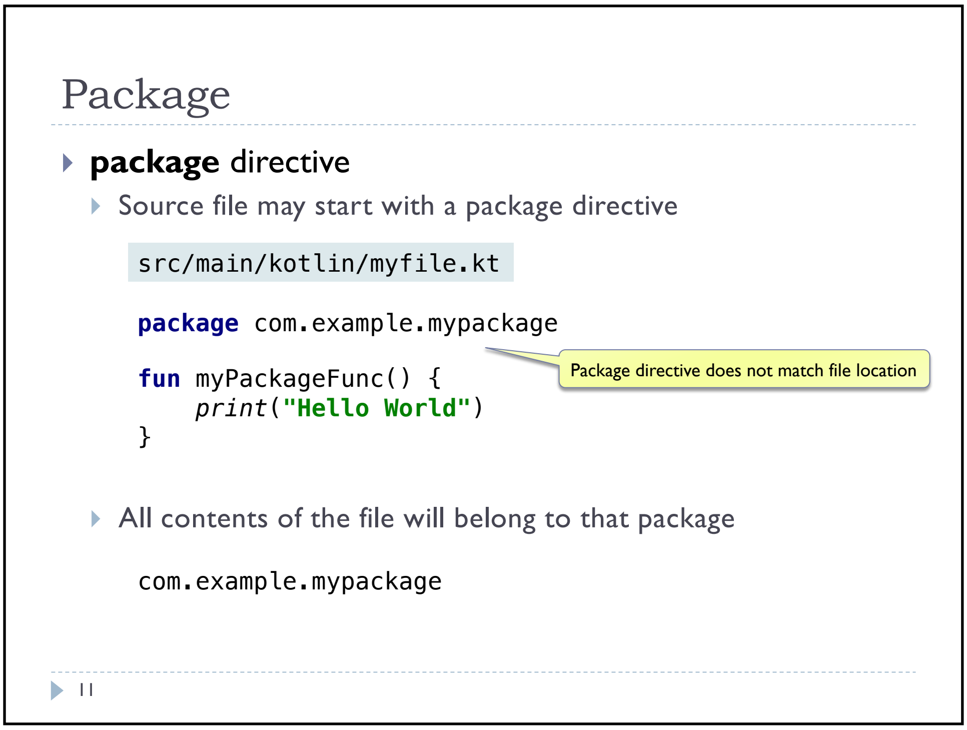
- No **package** directive

```
src/main/kotlin/myfile.kt
```

```
fun myFunc() {
    print("Hello World")
}
```

  - Symbols added to default namespace

10



Package

- **package** directive
  - Source file may start with a package directive

```
src/main/kotlin/myfile.kt
```

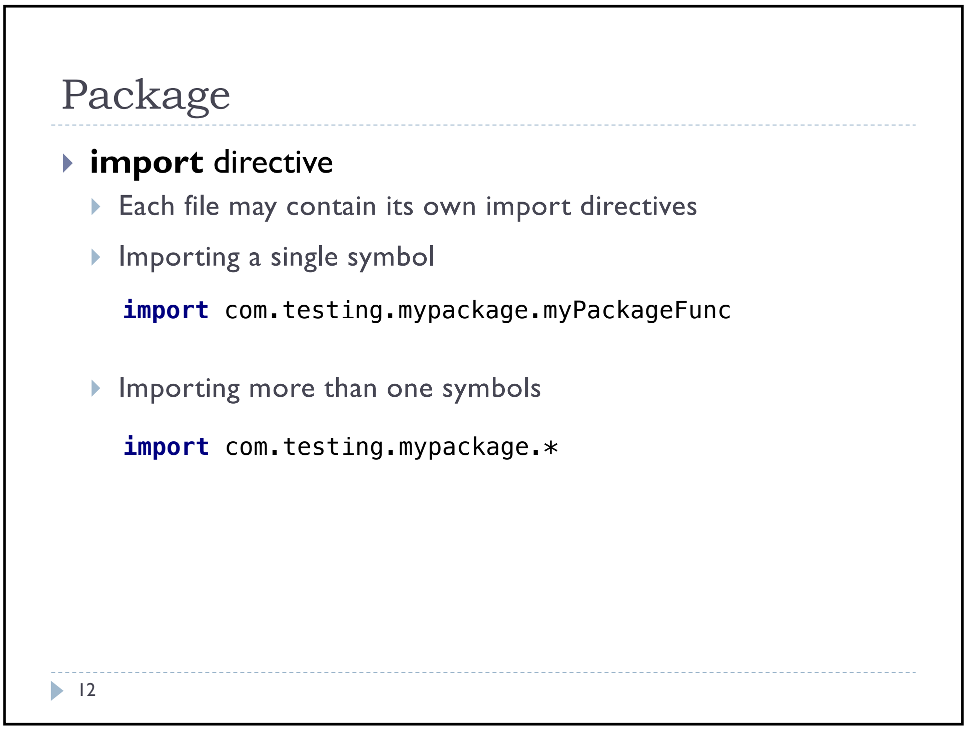
```
package com.example.mypackage

fun myPackageFunc() {
    print("Hello World")
}
```

Package directive does not match file location

  - All contents of the file will belong to that package

```
com.example.mypackage
```

11



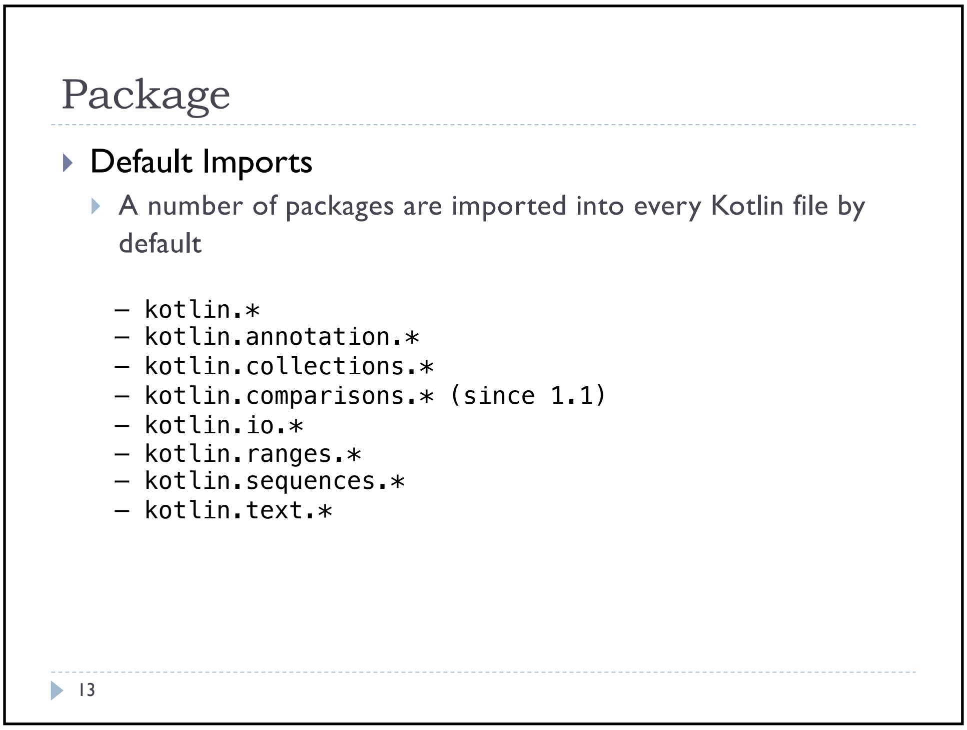
Package

- **import** directive
  - Each file may contain its own import directives
  - Importing a single symbol

```
import com.testing.mypackage.myPackageFunc
```

  - Importing more than one symbols

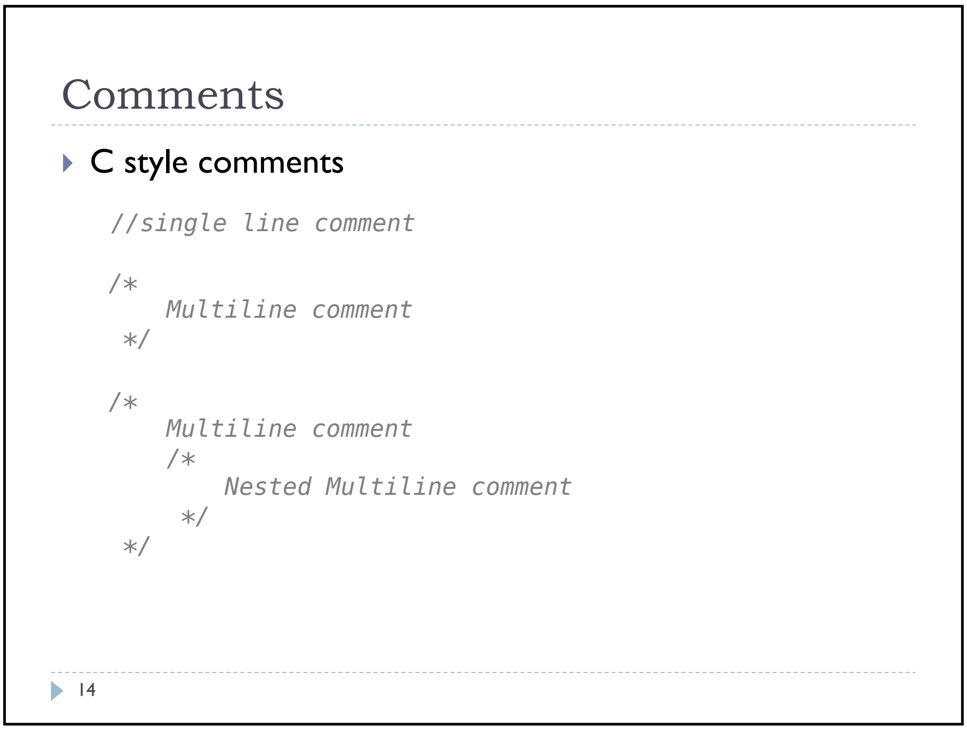
```
import com.testing.mypackage.*
```



Package

- Default Imports
  - A number of packages are imported into every Kotlin file by default

```
– kotlin.*
– kotlin.annotation.*
– kotlin.collections.*
– kotlin.comparisons.* (since 1.1)
– kotlin.io.*
– kotlin.ranges.*
– kotlin.sequences.*
– kotlin.text.*
```

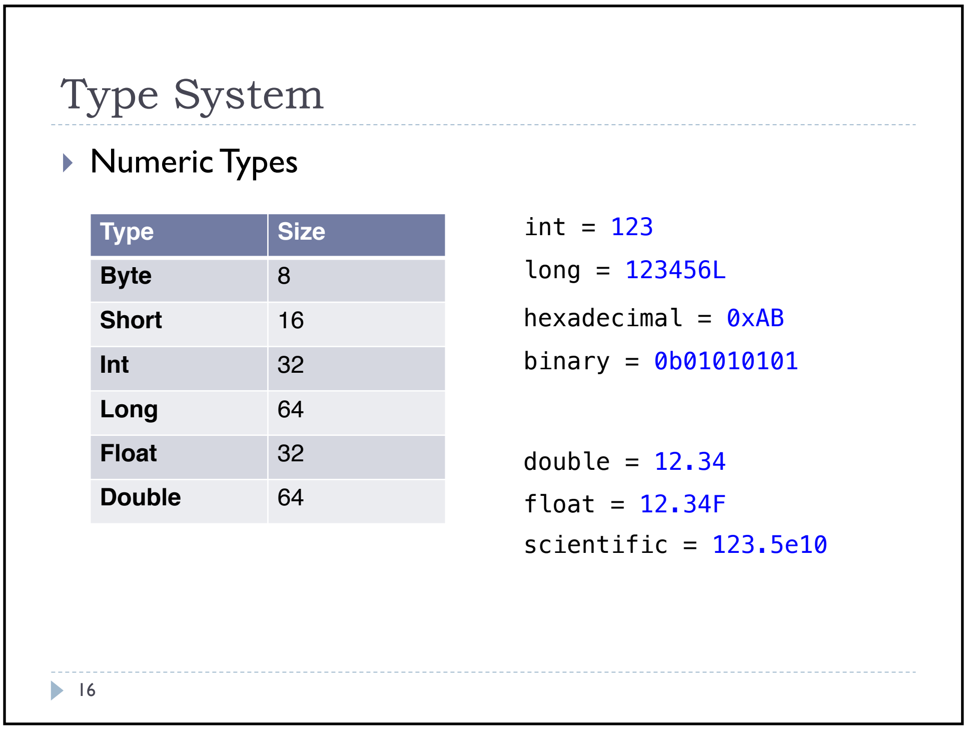


Comments

- C style comments

```
//single line comment

/*
    Multiline comment
 */

/*
    Multiline comment
    /*
        Nested Multiline comment
     */
 */
```

14



Type System

- Numeric Types

Type	Size
Byte	8
Short	16
Int	32
Long	64
Float	32
Double	64

```
int = 123
long = 123456L
hexadecimal = 0xAB
binary = 0b01010101


double = 12.34
float = 12.34F
scientific = 123.5e10
```

16

Type System

- Boolean Type

```
true
false
```

  - Result of logical expressions

```
x == y , x < y, x > y
```

  - Conjunction (&&), and disjunction (!!) operations

```
x < y && x < z
x == y || y == z
```

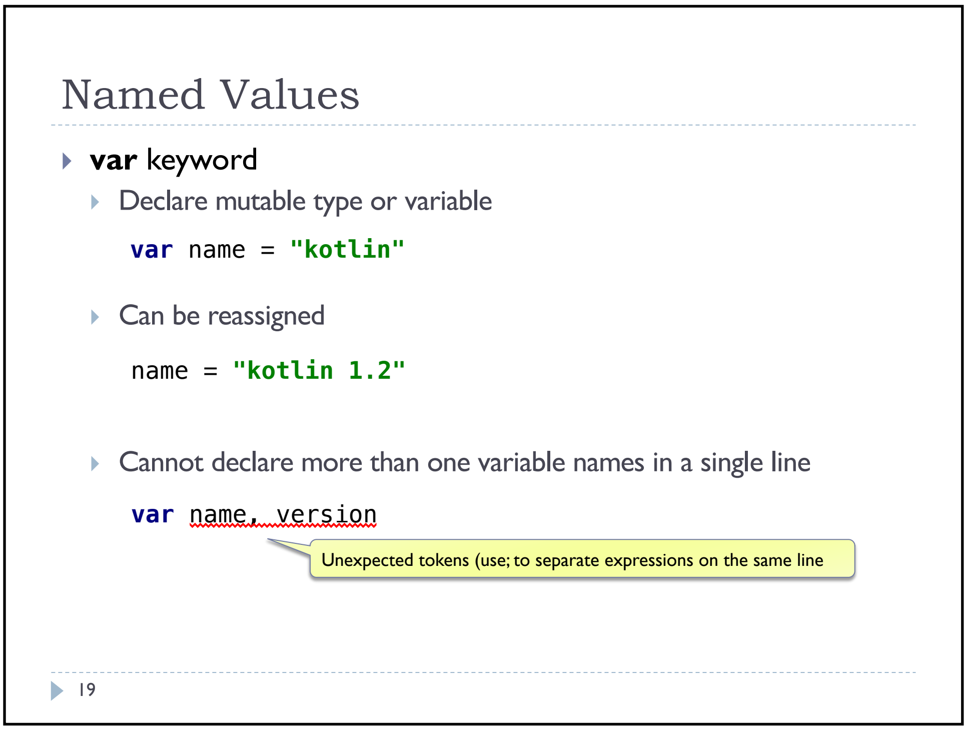
Type System

- Character Type
  - Character literals use single quotes

```
‘A’, ‘B’, ‘C’, ‘D’, ‘E’
```

- String Type
  - Ordered Collection of Characters enclosed in double quotes

```
"Hello, world!\n"
```



Named Values

- **var** keyword
  - Declare mutable type or variable

```
var name = "kotlin"
```

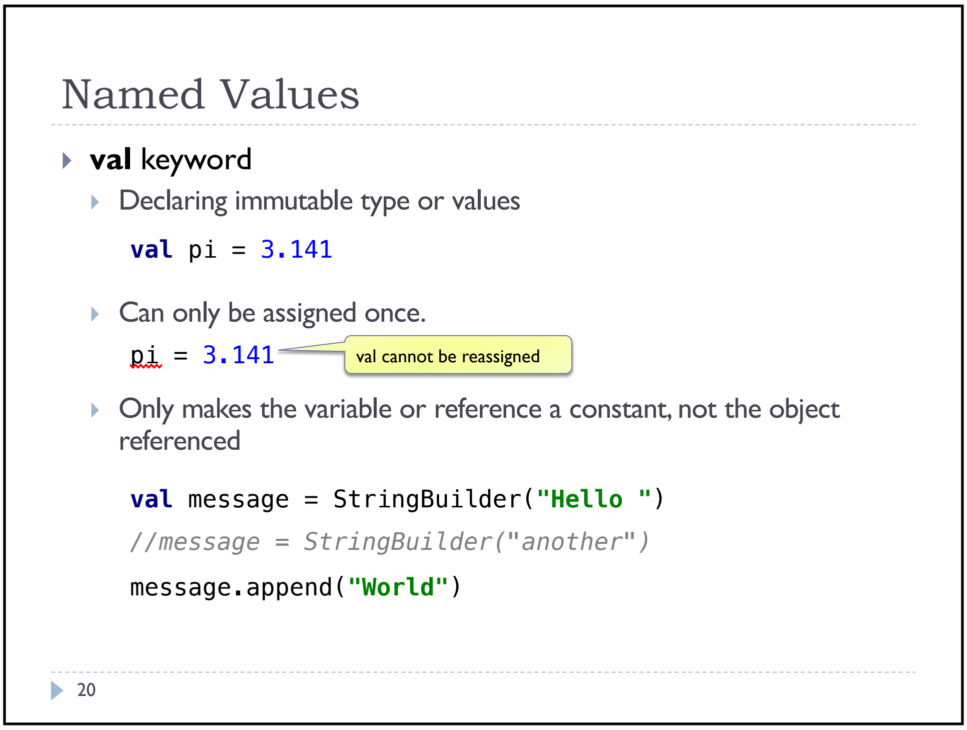
  - Can be reassigned

```
name = "kotlin 1.2"
```

  - Cannot declare more than one variable names in a single line

```
var name, version
```

Unexpected tokens (use; to separate expressions on the same line



Named Values

- **val** keyword
  - Declaring immutable type or values

```
val pi = 3.141
```

  - Can only be assigned once.

```
pi = 3.141
```

val cannot be reassigned

  - Only makes the variable or reference a constant, not the object referenced

```
val message = StringBuilder("Hello ")
//message = StringBuilder("another")
message.append("World")
```

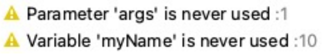
Named Values

- Naming Convention
  - Can contain almost any character, including Unicode characters
  - Cannot contain whitespace characters, mathematical symbols, arrows, etc.

Named Values

- Warnings

```
fun main(args: Array<String>) {
    var myName = "Amit Gulati"
}
```



⚠ Parameter 'args' is never used :1
⚠ Variable 'myName' is never used :10

```
fun main() {
    val myName = "Amit Gulati"
    println(myName)
}
```

Named Values with Types

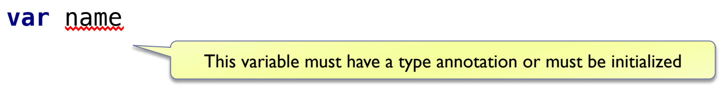
- Type System
  - Kotlin is a strongly and statically typed language
  - Named values must have a type
    - Type cannot change at runtime
  - Type for named values can be provided in 2 ways
    - Type Inference
    - Type Annotation (Explicit Type Definition)

Named Values with Types

- Type Inference
  - Kotlin compiler is able to infer the type of a variable

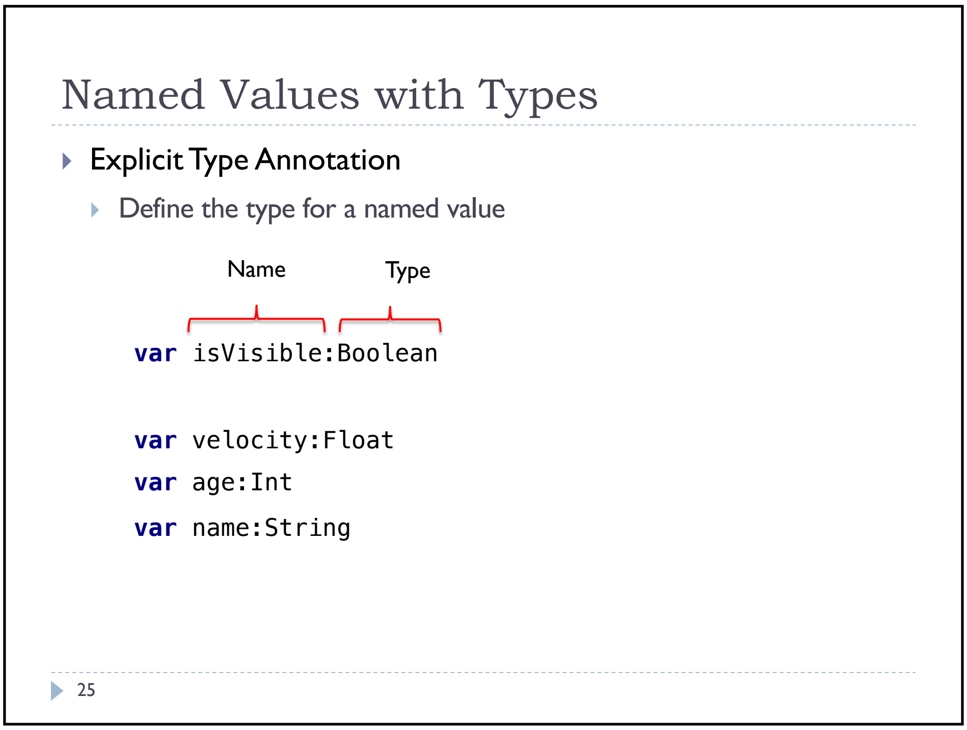
```
val greet = "hello"

println(greet)
println(greet::class)
println(greet.javaClass)
```

  - Value must be assigned for the compiler to infer the type of variable



```
var name
```

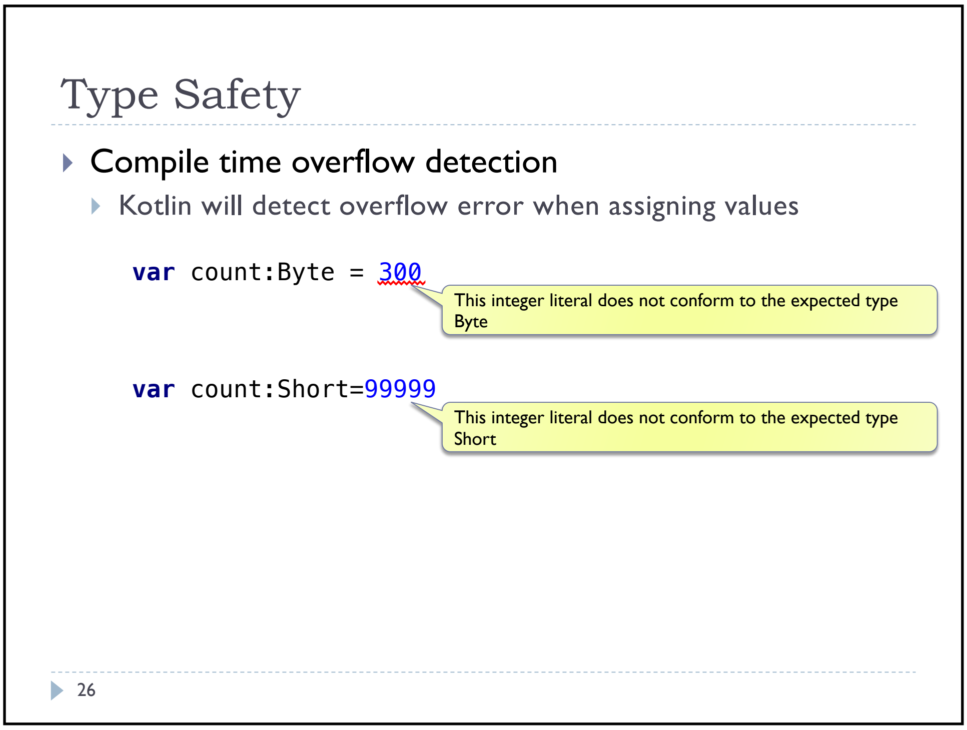
This variable must have a type annotation or must be initialized



Named Values with Types

- Explicit Type Annotation
  - Define the type for a named value

```
var isVisible:Boolean

var velocity:Float
var age:Int
var name:String
```



Type Safety

- Compile time overflow detection
  - Kotlin will detect overflow error when assigning values

```
var count:Byte = 300
```

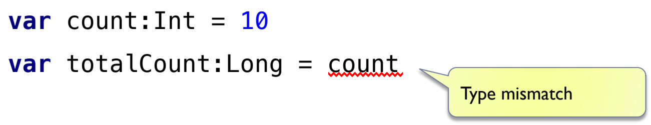
This integer literal does not conform to the expected type Byte

```
var count:Short=99999
```

This integer literal does not conform to the expected type Short

Type Safety

- Implicit Type Conversion
  - Kotlin does not support implicit type conversion



```
var count:Int = 10
var totalCount:Long = count
```

Type mismatch

  - Kotlin requires explicit type casting
    - Every variable type contains methods to convert it to other types

```
toLong()      toShort()     toChar()
toInt()       toFloat()
toByte()      toDouble()
```

Type Equality

- Two ways of Equality
  - == operator (Structural Equality)

```
val number1 = 100.6
val number2 = 100.6
println(number1 == number2)

true


val string1 = StringBuffer("Kotlin").toString()
val string2 = StringBuffer("Kotlin").toString()
println(string1 == string2)

true
```

Type Equality

- Two ways of Equality
  - === operator (Referential Equality)
    - Compares references

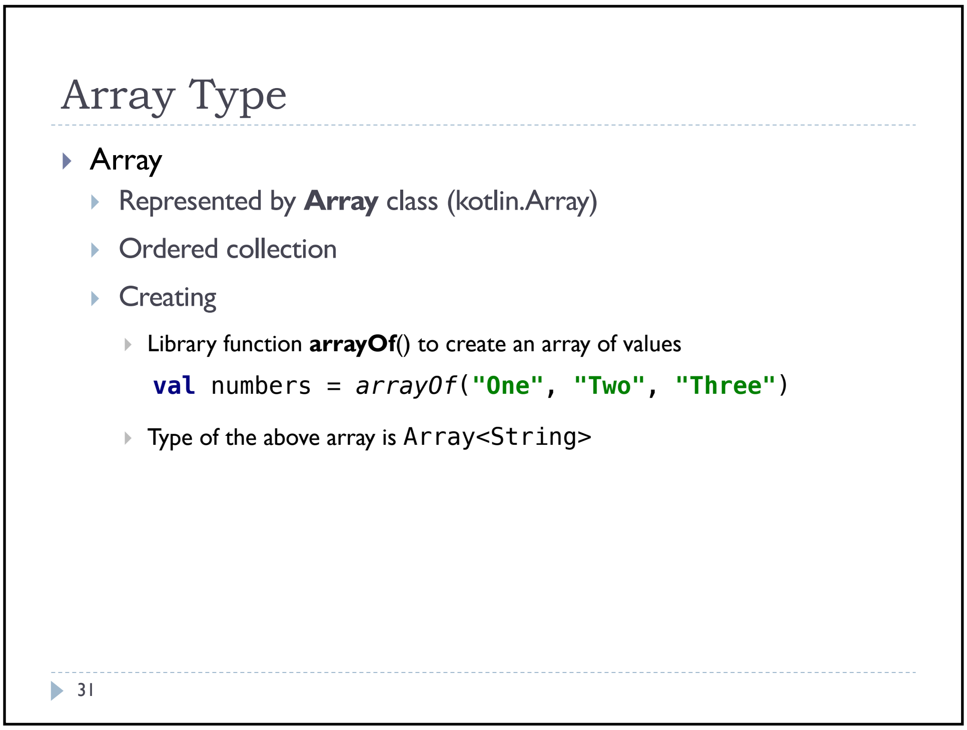
```
val string1 = StringBuffer("Kotlin").toString()
val string2 = string1
println(string1 === string2)

true
```

```
val string1 = StringBuffer("Kotlin").toString()
val string2 = StringBuffer("Kotlin").toString()
println(string1 === string2)

false
```

String formatting

- String Templates
  - Create String value from a mix of constants, variables, literals, and expressions
  - $ symbol is used to create a Template expression

```
val side = 100
print("Area of Square with side = $side is ${side * side}")
```

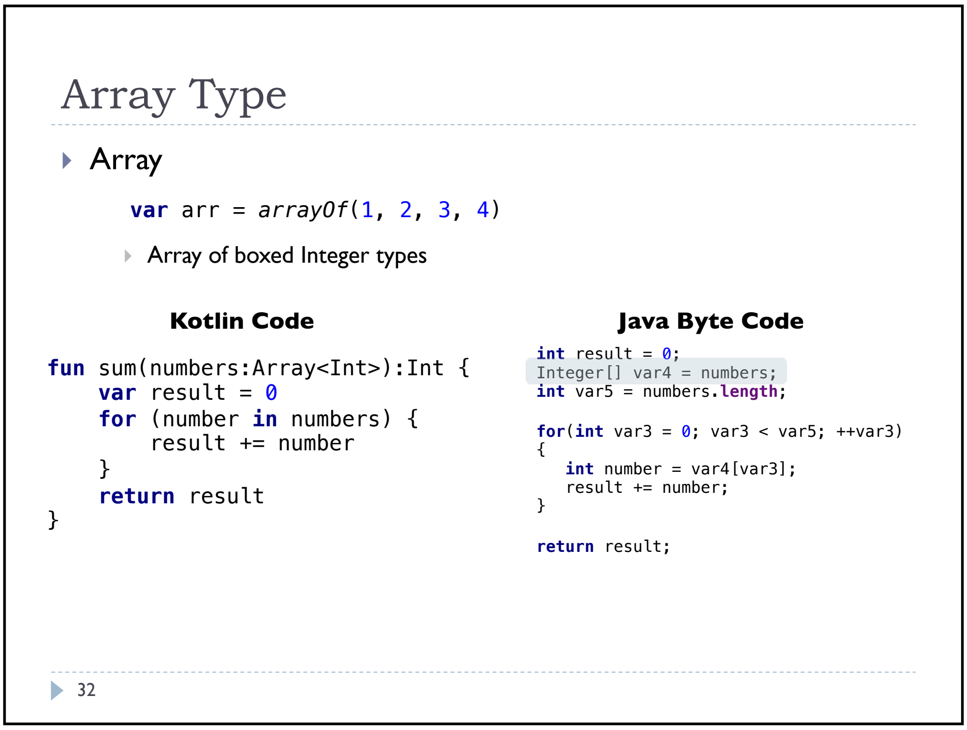


Array Type

- Array
  - Represented by **Array** class (kotlin.Array)
  - Ordered collection
  - Creating
    - Library function **arrayOf**() to create an array of values

```
val numbers = arrayOf("One", "Two", "Three")
```

    - Type of the above array is `Array<String>`



Array Type

- Array

```
var arr = arrayOf(1, 2, 3, 4)
```

  - Array of boxed Integer types

Kotlin Code

```
fun sum(numbers:Array<Int>):Int {
    var result = 0
    for (number in numbers) {
        result += number
    }
    return result
}
```

Java Byte Code

```
int result = 0;
Integer[] var4 = numbers;
int var5 = numbers.length;

for(int var3 = 0; var3 < var5; ++var3)
{
   int number = var4[var3];
   result += number;
}

return result;
```

Array Type

- Array
  - Arrays of primitive types so that boxing-unboxing can be avoided.
    - IntArray
    - ShortArray
    - ByteArray
  - Creating array of primate types
    - intArrayOf `intArrayOf(1, 2, 3, 4)`
    - shortArrayOf `shortArrayOf(1, 2, 3, 4)`
    - byteArrayOf `byteArrayOf(1, 2, 3, 4)`

Nullable Types

- What are Nullable Types?
  - Help Eliminate Null Pointer Exceptions
  - Not every variable in Kotlin can be assigned **null**

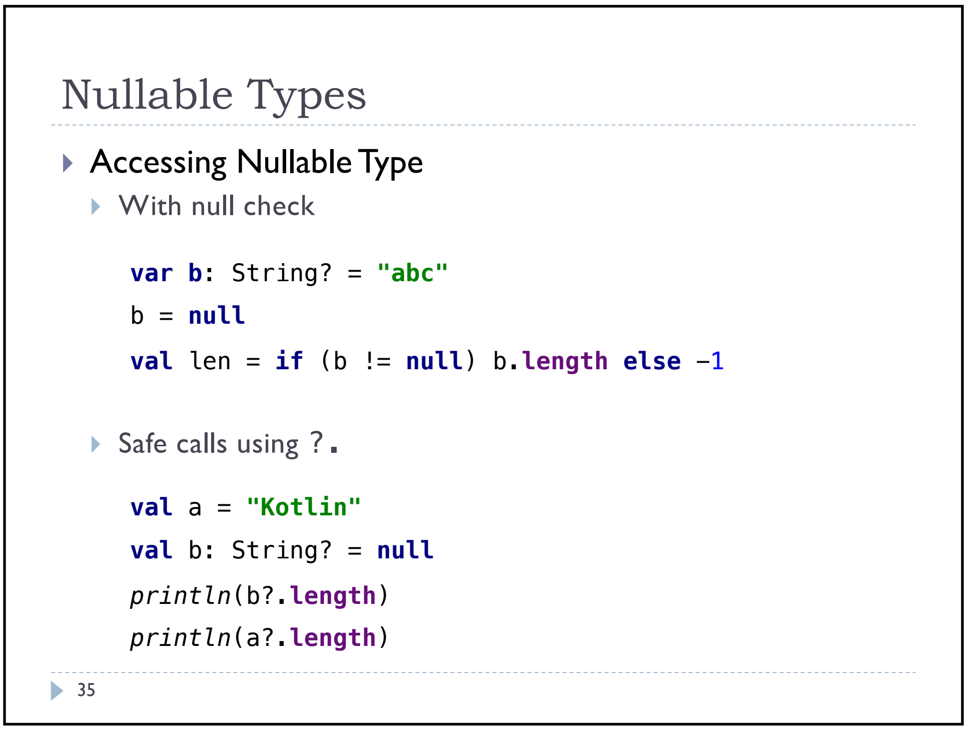
```
var a: String = "abc"
a = null
```

Null cannot be a value for a Non-Null type String

  - Nullable References are marked using ?

```
var b: String? = "abc"
b = null
```

  - Any type can be marked as a nullable



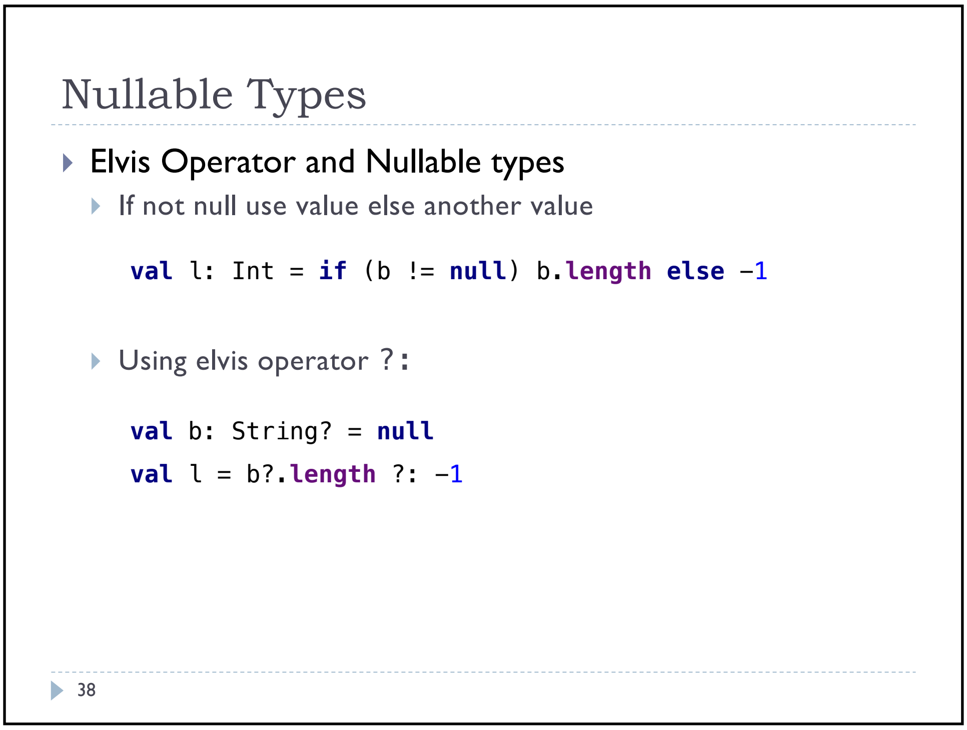
Nullable Types

- Accessing Nullable Type
  - With null check

```
var b: String? = "abc"
b = null
val len = if (b != null) b.length else -1
```

  - Safe calls using `?.`

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length)
```



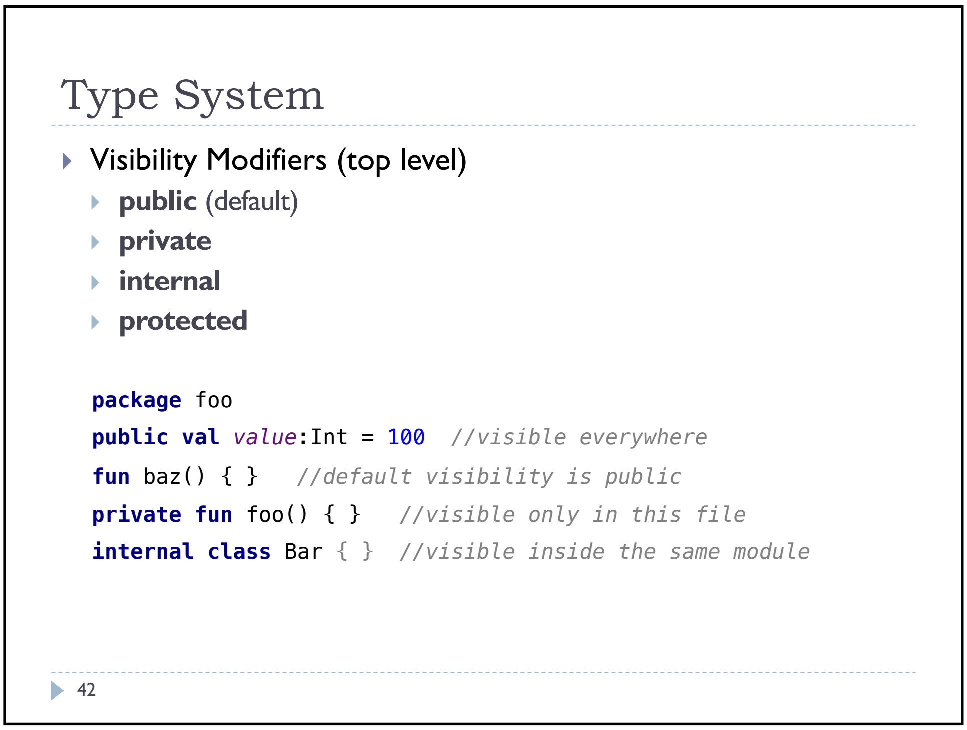
Nullable Types

- Elvis Operator and Nullable types
  - If not null use value else another value

```
val l: Int = if (b != null) b.length else -1
```

  - Using elvis operator `?:`

```
val b: String? = null
val l = b?.length ?: -1
```

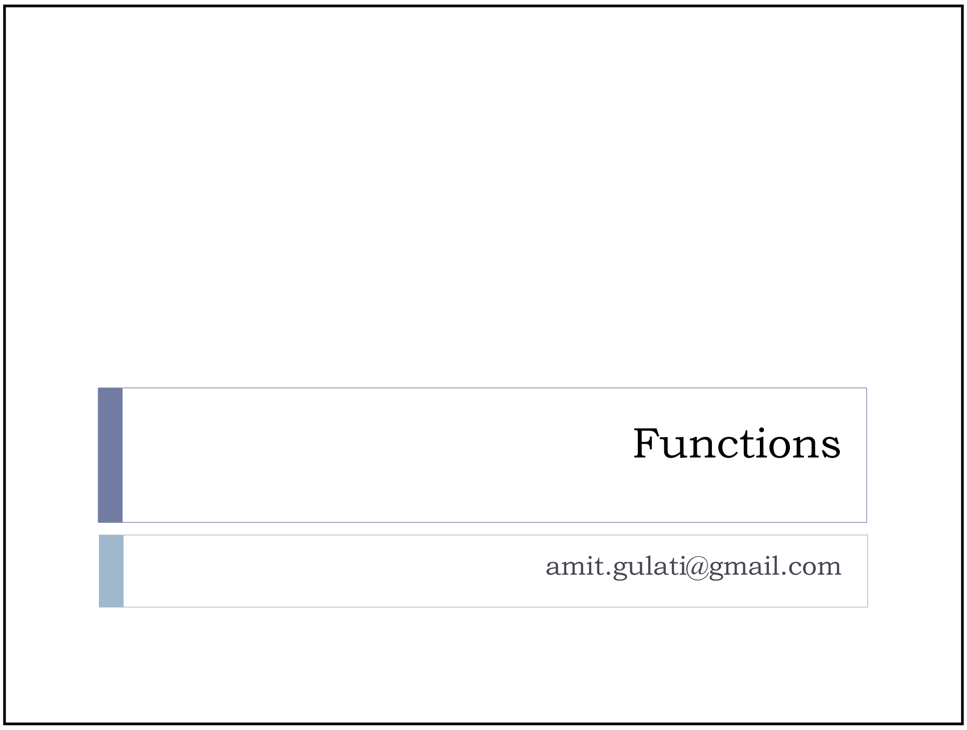


Type System

- Visibility Modifiers (top level)
  - **public** (default)
  - **private**
  - **internal**
  - **protected**

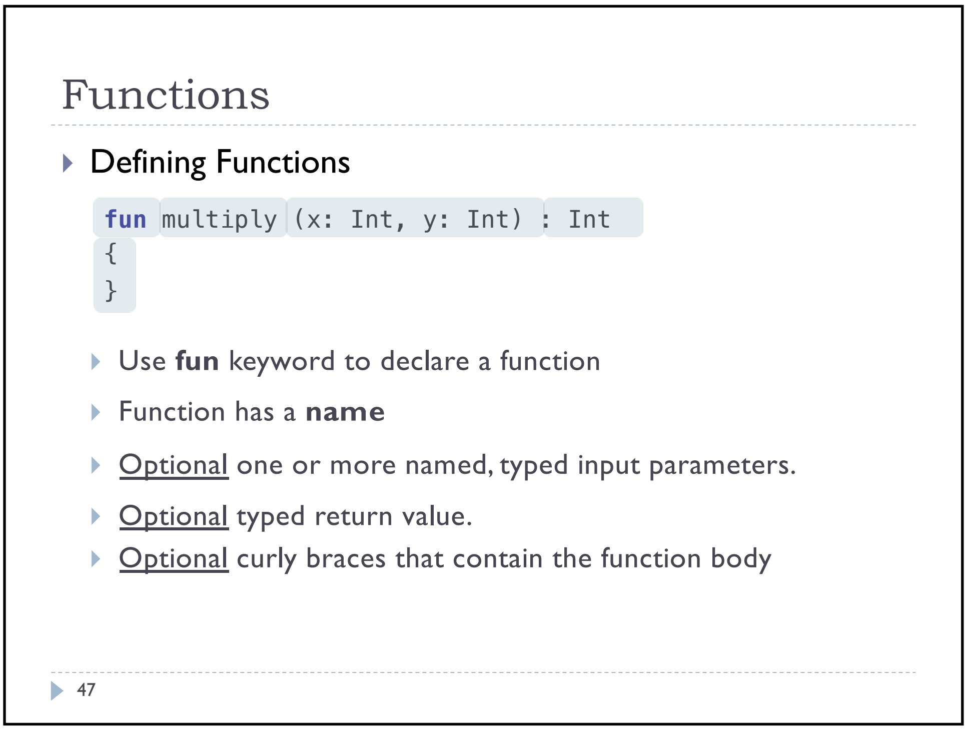
```
package foo
public val value:Int = 100  //visible everywhere
fun baz() { }   //default visibility is public
private fun foo() { }   //visible only in this file
internal class Bar { }  //visible inside the same module
```

42



Functions

amit.gulati@gmail.com

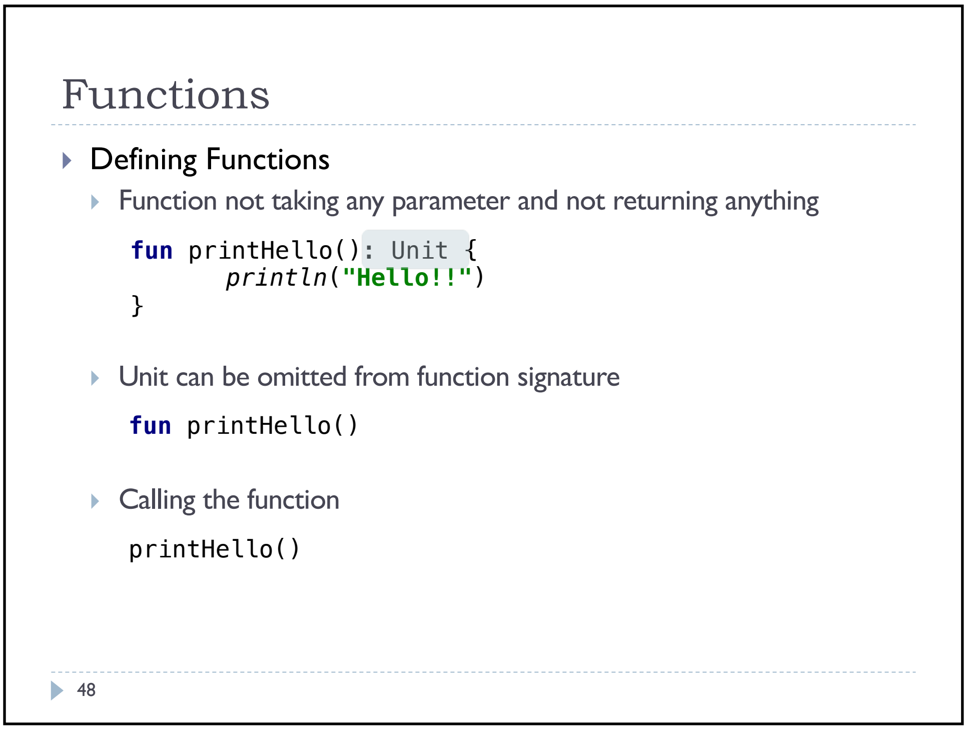


Functions

- Defining Functions

```
fun multiply (x: Int, y: Int) : Int
{
}
```

  - Use **fun** keyword to declare a function
  - Function has a **name**
  - Optional one or more named, typed input parameters.
  - Optional typed return value.
  - Optional curly braces that contain the function body



Functions

- Defining Functions
  - Function not taking any parameter and not returning anything

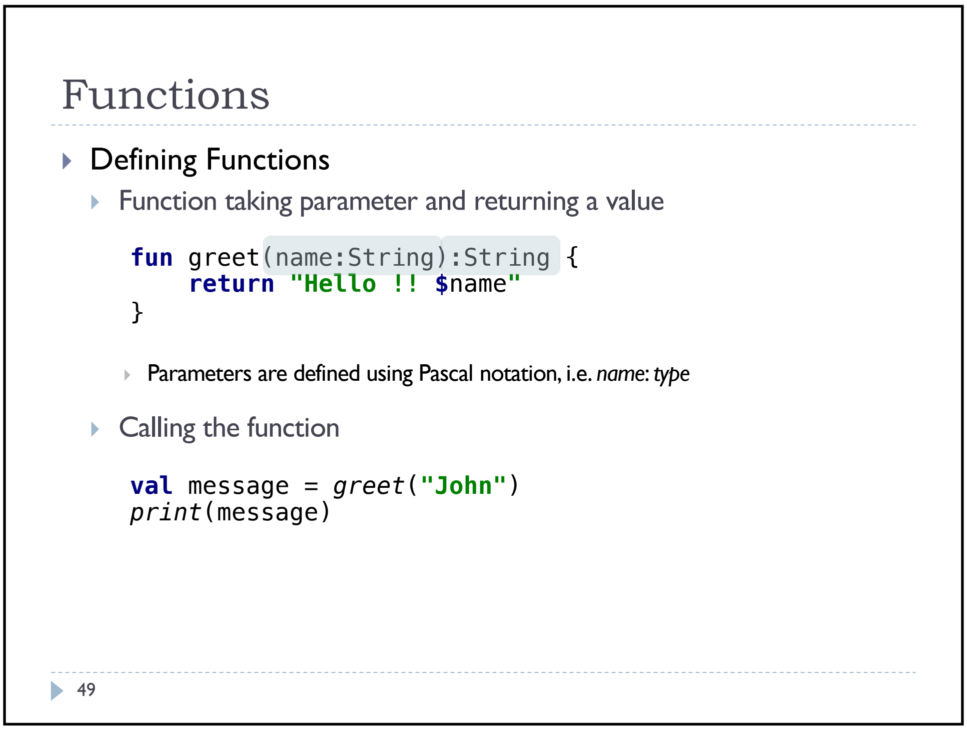
```
fun printHello(): Unit {
        println("Hello!!")
}
```

  - Unit can be omitted from function signature

```
fun printHello()
```

  - Calling the function

```
printHello()
```



Functions

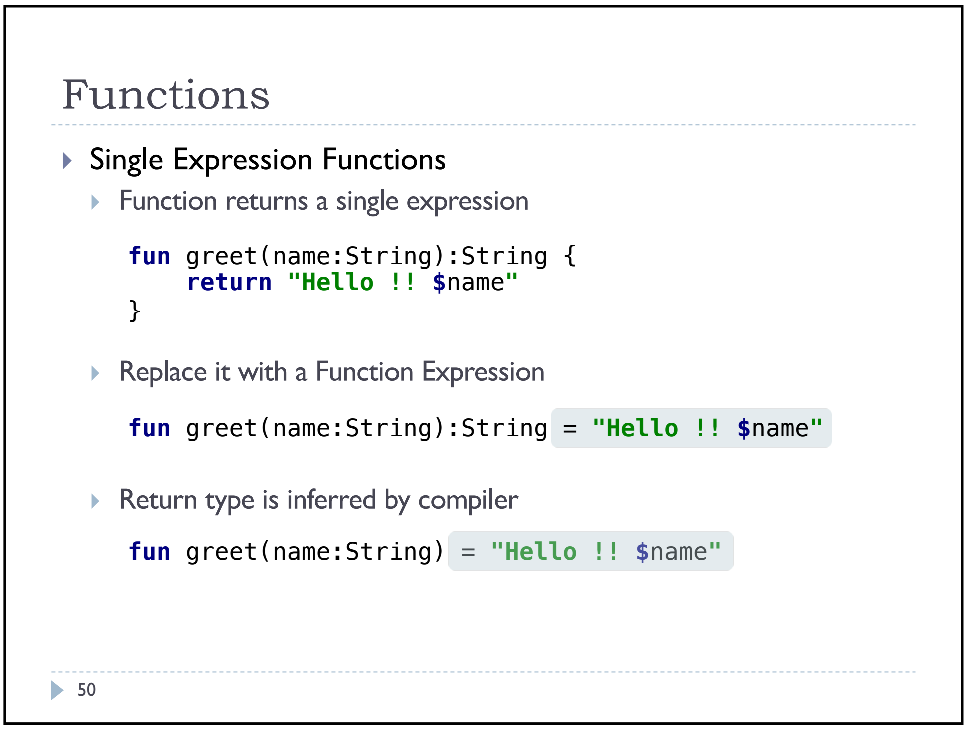
- Defining Functions
  - Function taking parameter and returning a value

```
fun greet(name:String):String {
    return "Hello !! $name"
}
```

- Parameters are defined using Pascal notation, i.e. *name: type*

- Calling the function

```
val message = greet("John")
print(message)
```



Functions

- Single Expression Functions
  - Function returns a single expression

```
fun greet(name:String):String {
    return "Hello !! $name"
}
```

- Replace it with a Function Expression

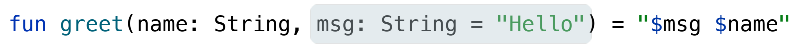
```
fun greet(name:String):String = "Hello !! $name"
```

- Return type is inferred by compiler

```
fun greet(name:String) = "Hello !! $name"
```

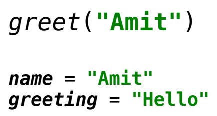
Functions

- Default Arguments
  - Function parameters can have default values.
  - Specified using the assignment operator



```
fun greet(name: String, msg: String = "Hello") = "$msg $name"
```

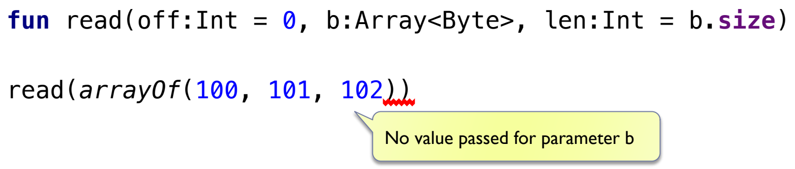
  - Used when a corresponding argument is omitted.



```
greet("Amit")

name = "Amit"
greeting = "Hello"
```

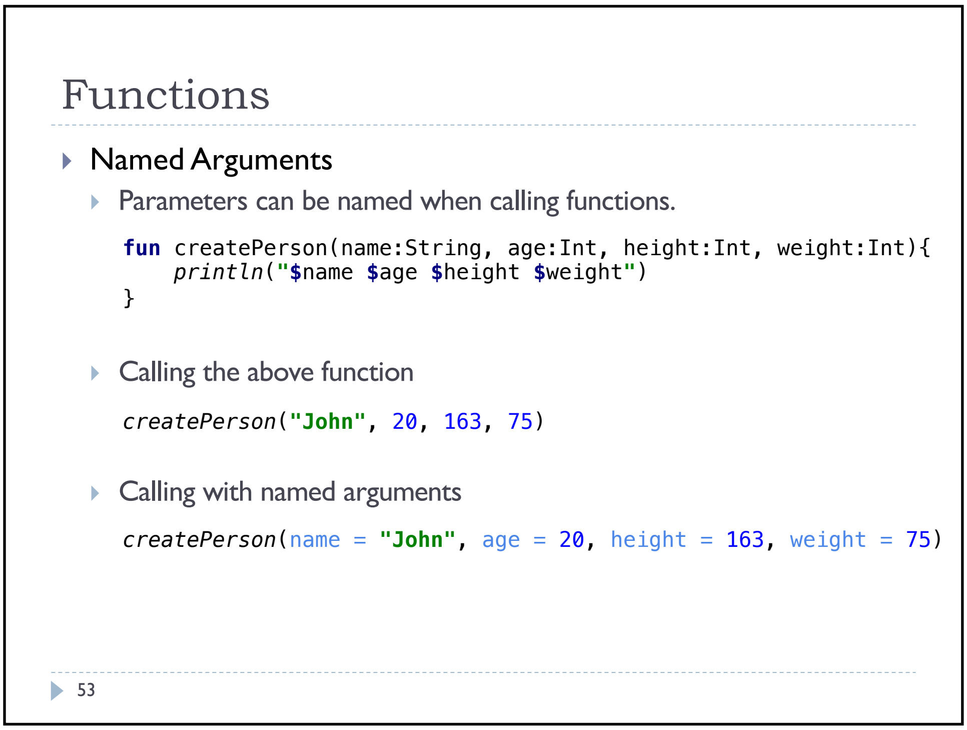
Functions

- Default Arguments
  - Parameters with default values are placed towards the end of the parameter list.
  - If placed in the beginning of parameter list



```
fun read(off:Int = 0, b:Array<Byte>, len:Int = b.size)

read(arrayOf(100, 101, 102))
```

No value passed for parameter b



Functions

- Named Arguments
  - Parameters can be named when calling functions.

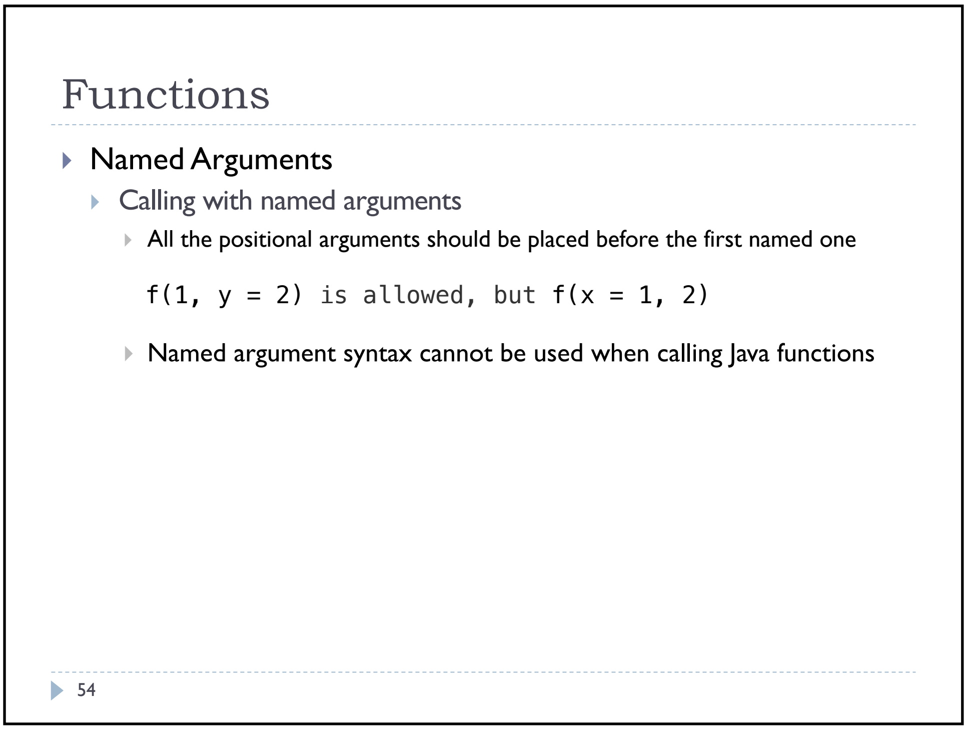
```
fun createPerson(name:String, age:Int, height:Int, weight:Int){
    println("$name $age $height $weight")
}
```

  - Calling the above function

```
createPerson("John", 20, 163, 75)
```

  - Calling with named arguments

```
createPerson(name = "John", age = 20, height = 163, weight = 75)
```



Functions

- Named Arguments
  - Calling with named arguments
    - All the positional arguments should be placed before the first named one

```
f(1, y = 2) is allowed, but f(x = 1, 2)
```

    - Named argument syntax cannot be used when calling Java functions

Functions

- Returning Multiple values
  - Pair and Triple
    - Pair<A, B>

```
Pair<Int, Int>(100, 99)
```

    - Triple<A, B, C>

```
Triple<Int, String, Char>(1, "One", 'O')
```

  - Function returning multiple values

```
fun minMax(numbers:IntArray):Pair<Int, Int>
```

Functions

- Destructuring

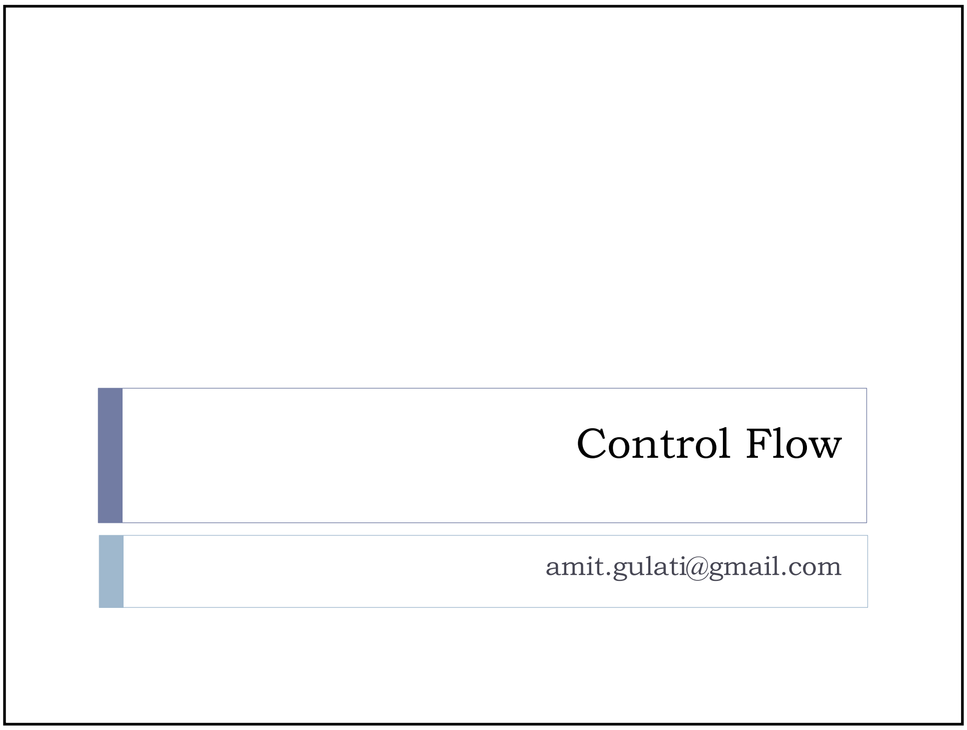
```
fun minMax(numbers:IntArray):Pair<Int, Int>
```

  - Without Destructuring

```
val result = minMax(intArrayOf(100, 34, 99, 20, 5))
println("${result.first}, ${result.second}")
```

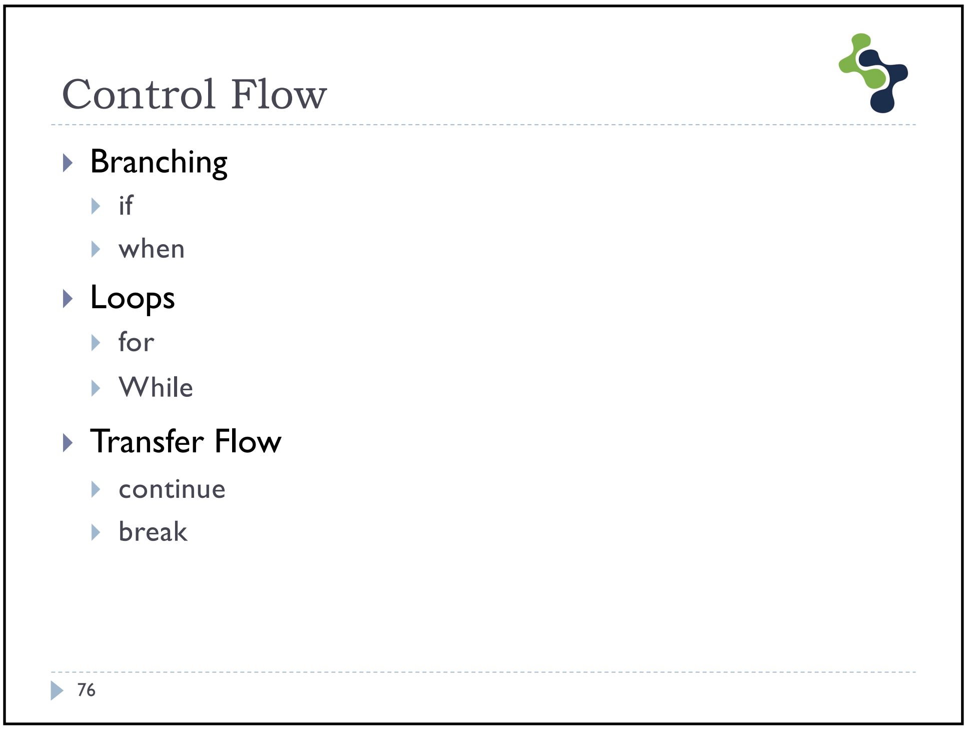
  - With Destructuring

```
val (min, max) = minMax(intArrayOf(100, 34, 99, 20, 5))
println("$min, $max")
```



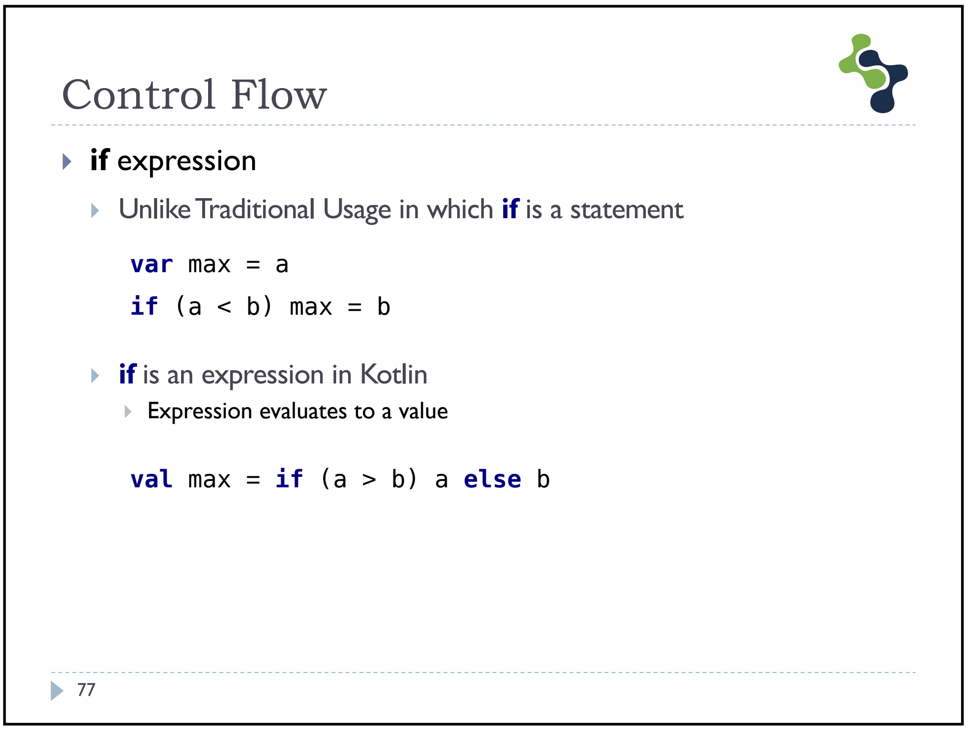
Control Flow

amit.gulati@gmail.com



Control Flow

- Branching
  - if
  - when
- Loops
  - for
  - While
- Transfer Flow
  - continue
  - break



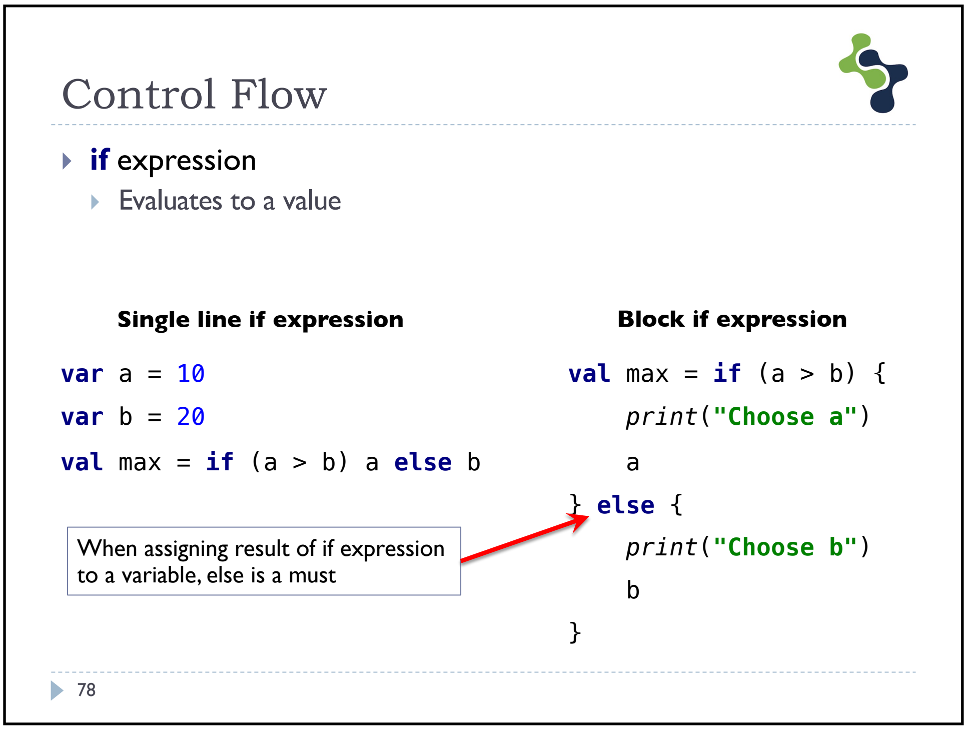
Control Flow

- **if** expression
  - Unlike Traditional Usage in which **if** is a statement

```
var max = a
if (a < b) max = b
```

  - **if** is an expression in Kotlin
    - Expression evaluates to a value

```
val max = if (a > b) a else b
```



Control Flow

- **if** expression
  - Evaluates to a value

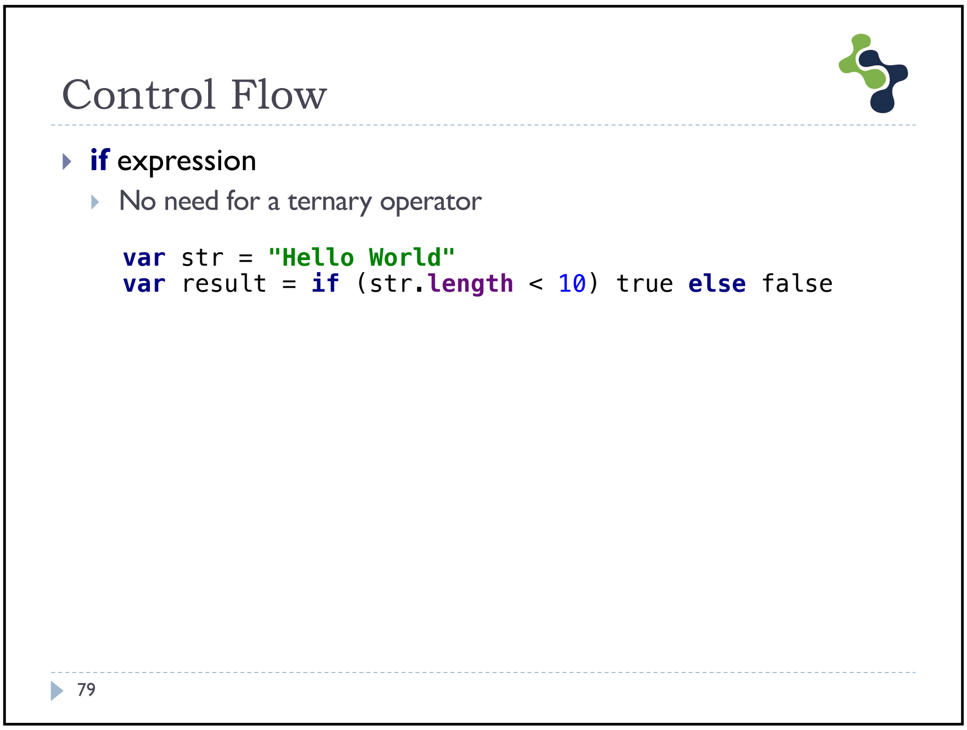
Single line if expression

```
var a = 10
var b = 20
val max = if (a > b) a else b
```

When assigning result of if expression to a variable, else is a must

Block if expression

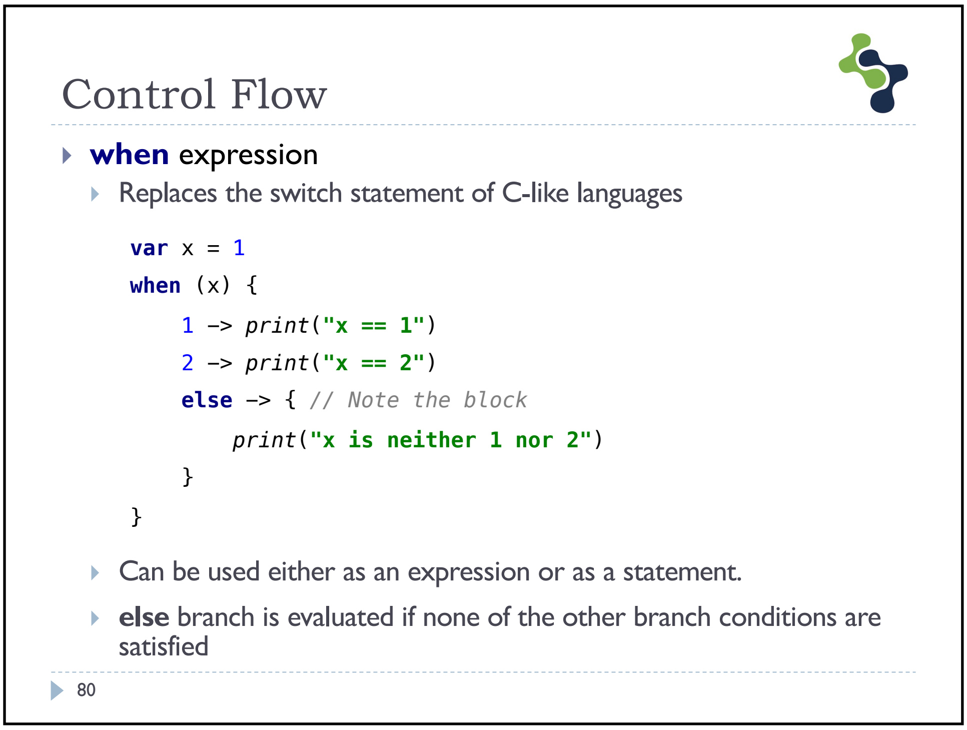
```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```



Control Flow

- **if** expression
  - No need for a ternary operator

```
var str = "Hello World"
var result = if (str.length < 10) true else false
```



Control Flow

- **when** expression
  - Replaces the switch statement of C-like languages

```
var x = 1
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

  - Can be used either as an expression or as a statement.
  - **else** branch is evaluated if none of the other branch conditions are satisfied

Control Flow

- **when** expression
  - Multiple matches

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

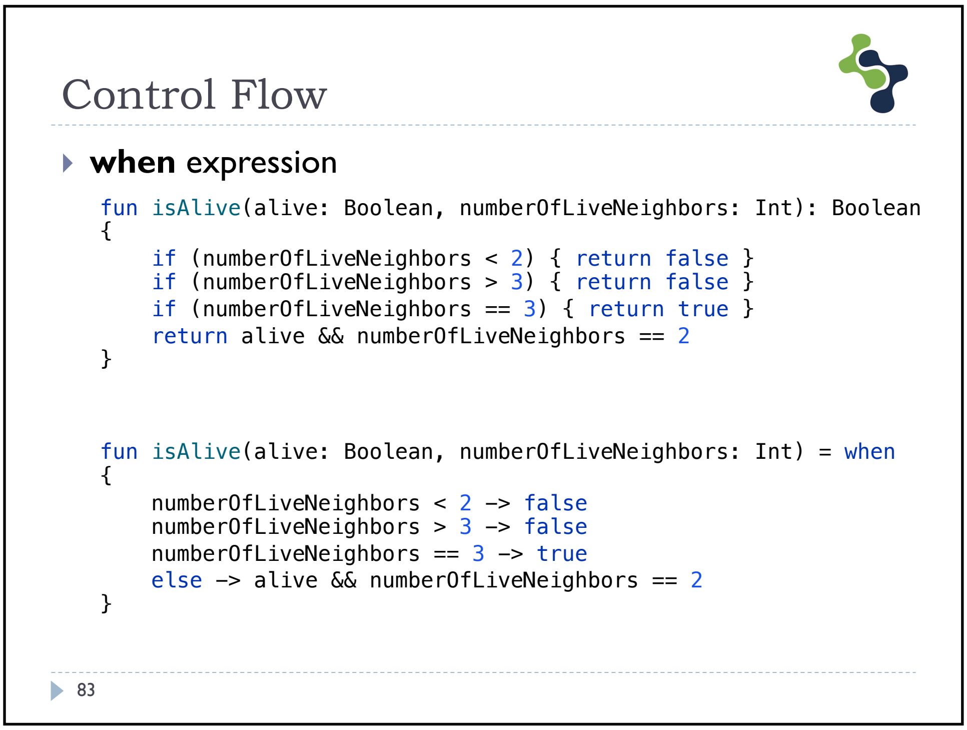
  - Expressions as matching values

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

Control Flow

- **when** expression
  - Range as matching values

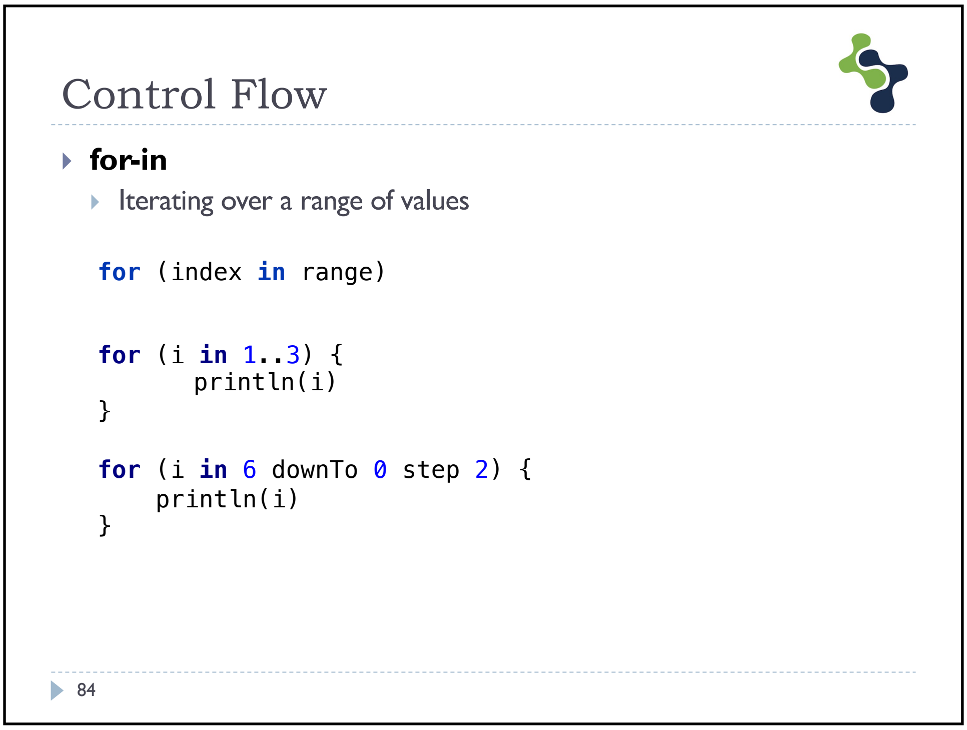
```
when(temperature) {
        in Float.MIN_VALUE..60.0f -> "Too Cold"
        in 70.0f..Float.MAX_VALUE -> "Too Hot"
        in 60.0f..70.0f -> "Just Right"
        else -> "Not Sure"
```



Control Flow

- **when** expression

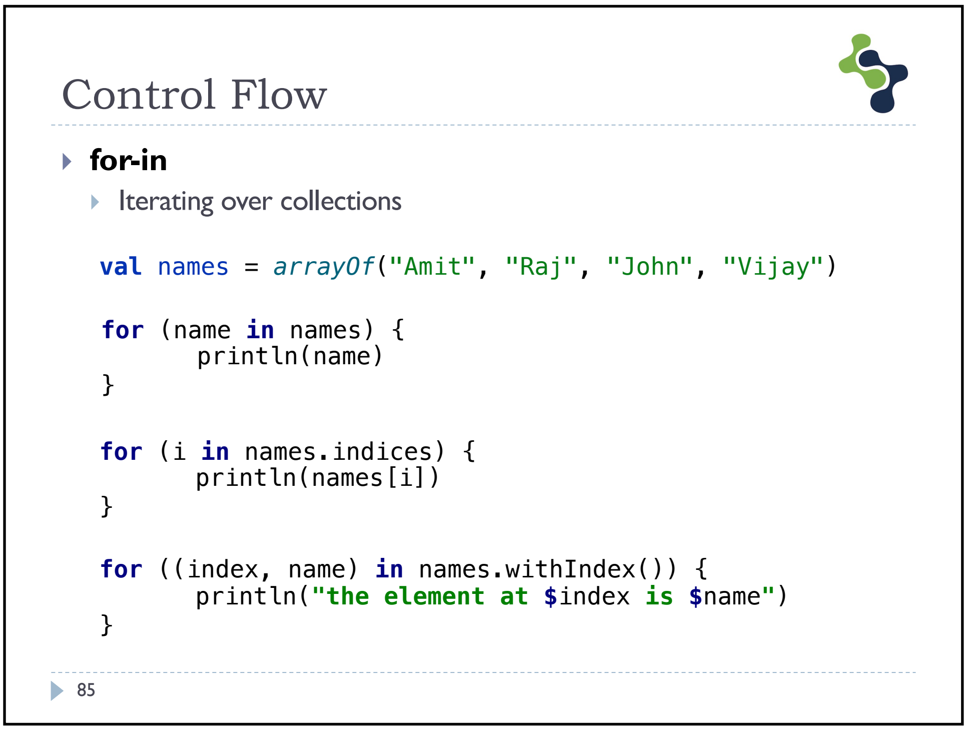
```
fun isAlive(alive: Boolean, numberOfLiveNeighbors: Int): Boolean
{
    if (numberOfLiveNeighbors < 2) { return false }
    if (numberOfLiveNeighbors > 3) { return false }
    if (numberOfLiveNeighbors == 3) { return true }
    return alive && numberOfLiveNeighbors == 2
}


fun isAlive(alive: Boolean, numberOfLiveNeighbors: Int) = when
{
    numberOfLiveNeighbors < 2 -> false
    numberOfLiveNeighbors > 3 -> false
    numberOfLiveNeighbors == 3 -> true
    else -> alive && numberOfLiveNeighbors == 2
}
```



Control Flow

- **for-in**
  - Iterating over a range of values

```
for (index in range)


for (i in 1..3) {
       println(i)
}

for (i in 6 downTo 0 step 2) {
    println(i)
}
```

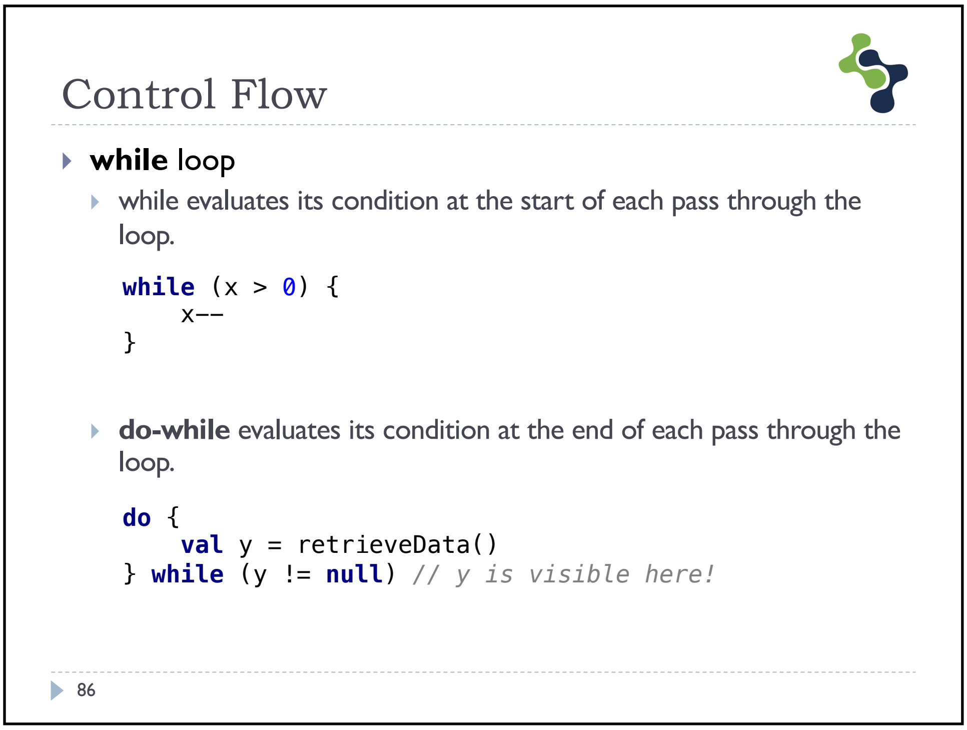


Control Flow

- **for-in**
  - Iterating over collections

```
val names = arrayOf("Amit", "Raj", "John", "Vijay")

for (name in names) {
        println(name)
}

for (i in names.indices) {
        println(names[i])
}

for ((index, name) in names.withIndex()) {
        println("the element at $index is $name")
}
```

85



Control Flow

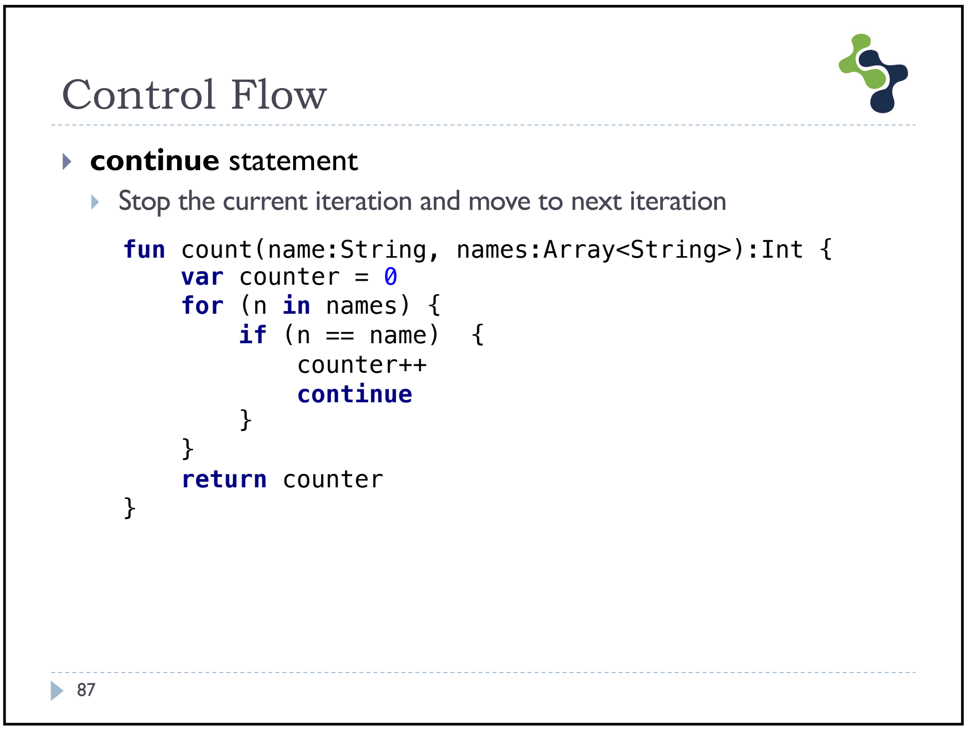
- **while** loop
  - while evaluates its condition at the start of each pass through the loop.

```
while (x > 0) {
    x--
}
```

  - **do-while** evaluates its condition at the end of each pass through the loop.

```
do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

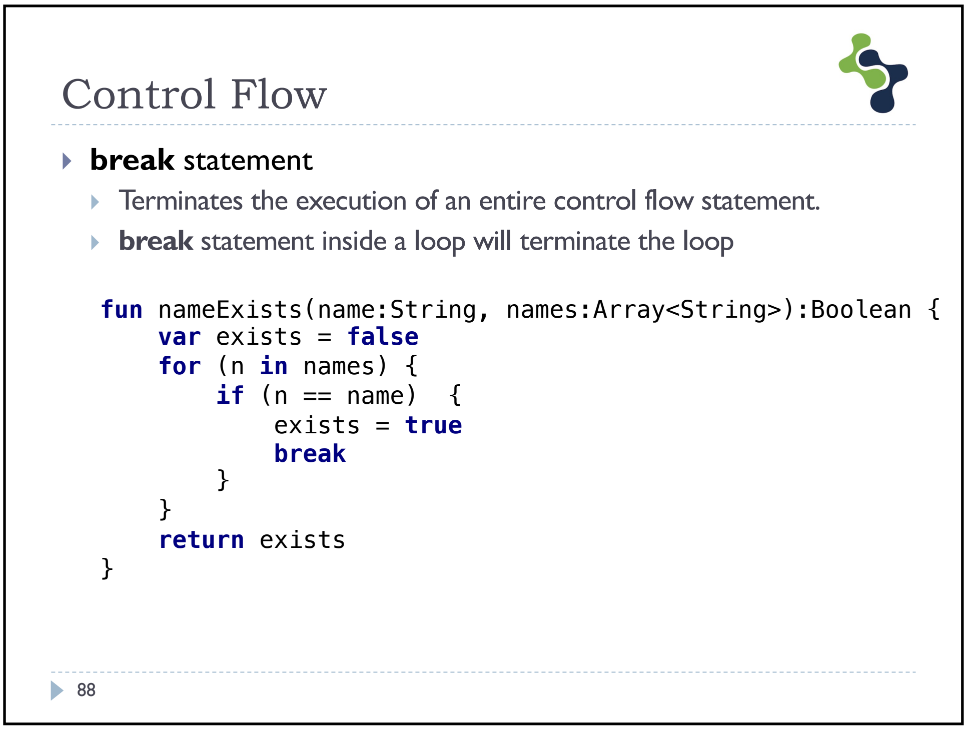
86



Control Flow

- **continue** statement
  - Stop the current iteration and move to next iteration

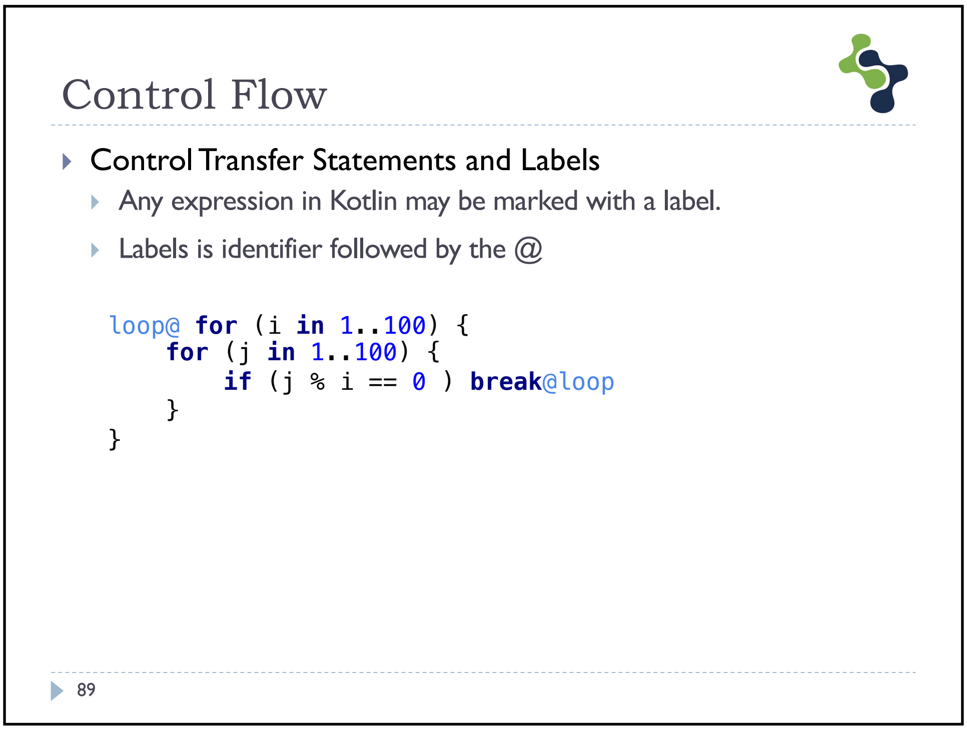
```
fun count(name:String, names:Array<String>):Int {
    var counter = 0
    for (n in names) {
        if (n == name)  {
            counter++
            continue
        }
    }
    return counter
}
```



Control Flow

- **break** statement
  - Terminates the execution of an entire control flow statement.
  - **break** statement inside a loop will terminate the loop

```
fun nameExists(name:String, names:Array<String>):Boolean {
    var exists = false
    for (n in names) {
        if (n == name)  {
            exists = true
            break
        }
    }
    return exists
}
```



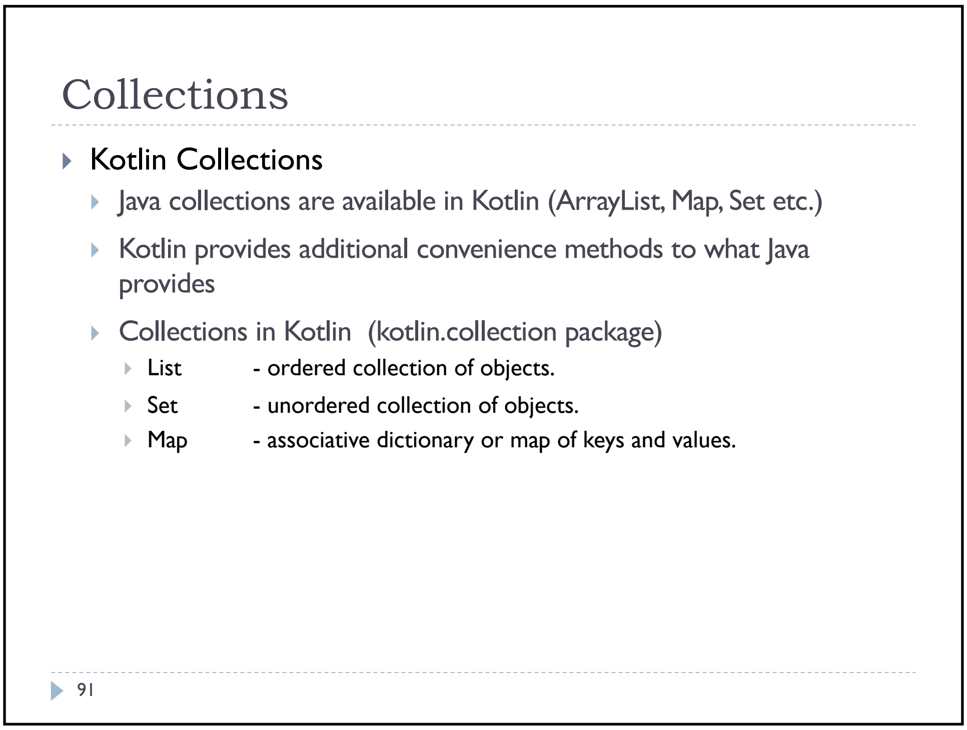
Control Flow

- Control Transfer Statements and Labels
  - Any expression in Kotlin may be marked with a label.
  - Labels is identifier followed by the @

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (j % i == 0 ) break@loop
    }
}
```

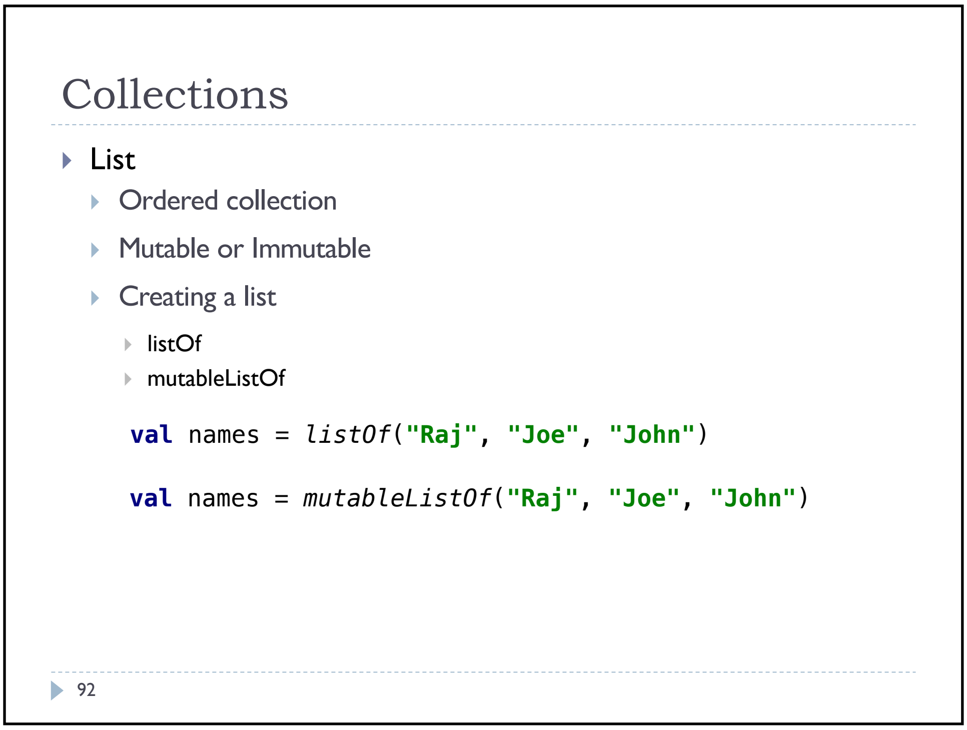
Collections

amit.gulati@gmail.com



Collections

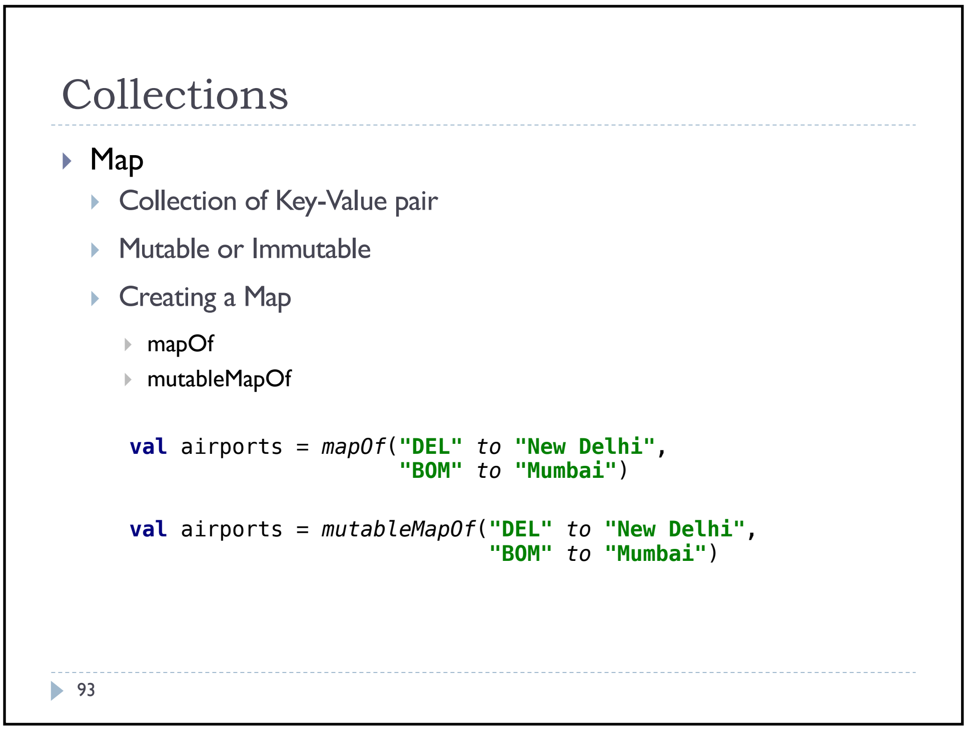
- Kotlin Collections
  - Java collections are available in Kotlin (ArrayList, Map, Set etc.)
  - Kotlin provides additional convenience methods to what Java provides
  - Collections in Kotlin (kotlin.collection package)
    - List - ordered collection of objects.
    - Set - unordered collection of objects.
    - Map - associative dictionary or map of keys and values.



Collections

- List
  - Ordered collection
  - Mutable or Immutable
  - Creating a list
    - listOf
    - mutableListOf

```kotlin
val names = listOf("Raj", "Joe", "John")

val names = mutableListOf("Raj", "Joe", "John")
```

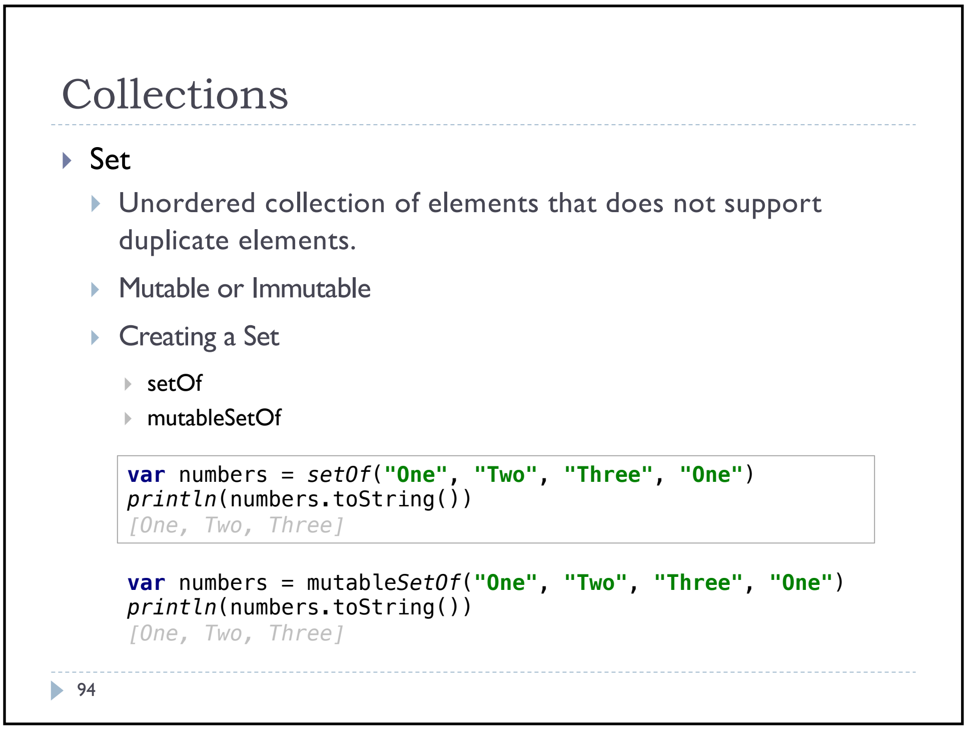


Collections

- Map
  - Collection of Key-Value pair
  - Mutable or Immutable
  - Creating a Map
    - mapOf
    - mutableMapOf

```
val airports = mapOf("DEL" to "New Delhi",
                     "BOM" to "Mumbai")

val airports = mutableMapOf("DEL" to "New Delhi",
                            "BOM" to "Mumbai")
```

93



Collections

- Set
  - Unordered collection of elements that does not support duplicate elements.
  - Mutable or Immutable
  - Creating a Set
    - setOf
    - mutableSetOf

```
var numbers = setOf("One", "Two", "Three", "One")
println(numbers.toString())
[One, Two, Three]
```

```
var numbers = mutableSetOf("One", "Two", "Three", "One")
println(numbers.toString())
[One, Two, Three]
```

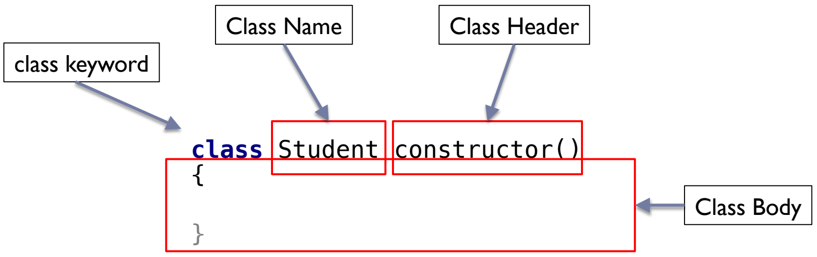
94

Classes and Objects

amit.gulati@gmail.com

Classes and Objects

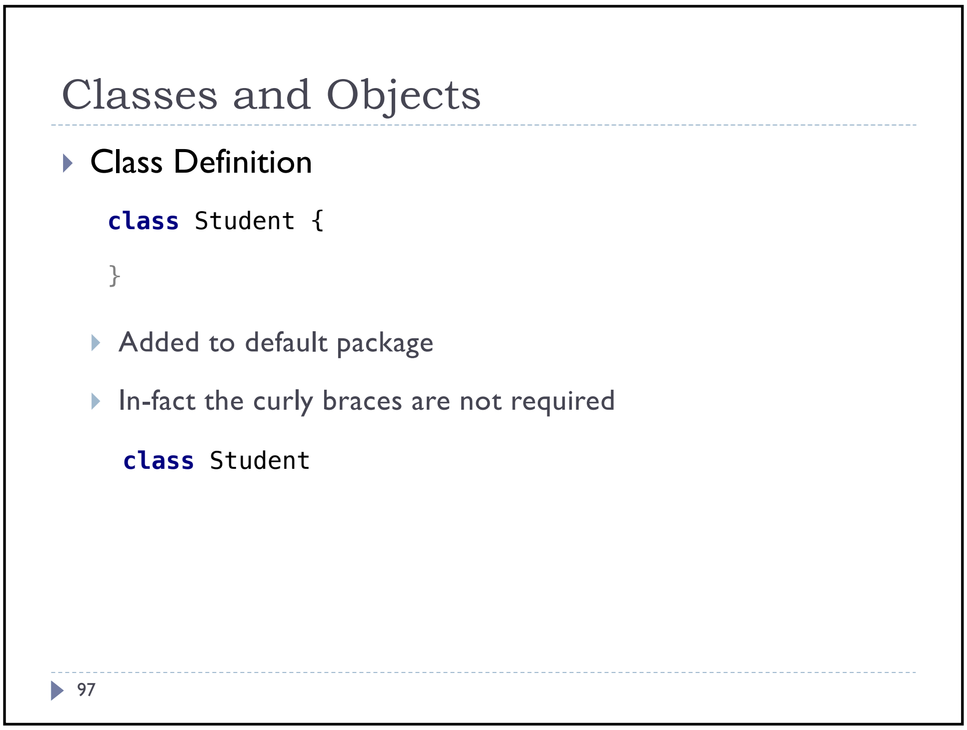
- Class Definition



```
class Student constructor()
{

}
```

- Only thing that is mandatory is the class keyword and class name

96



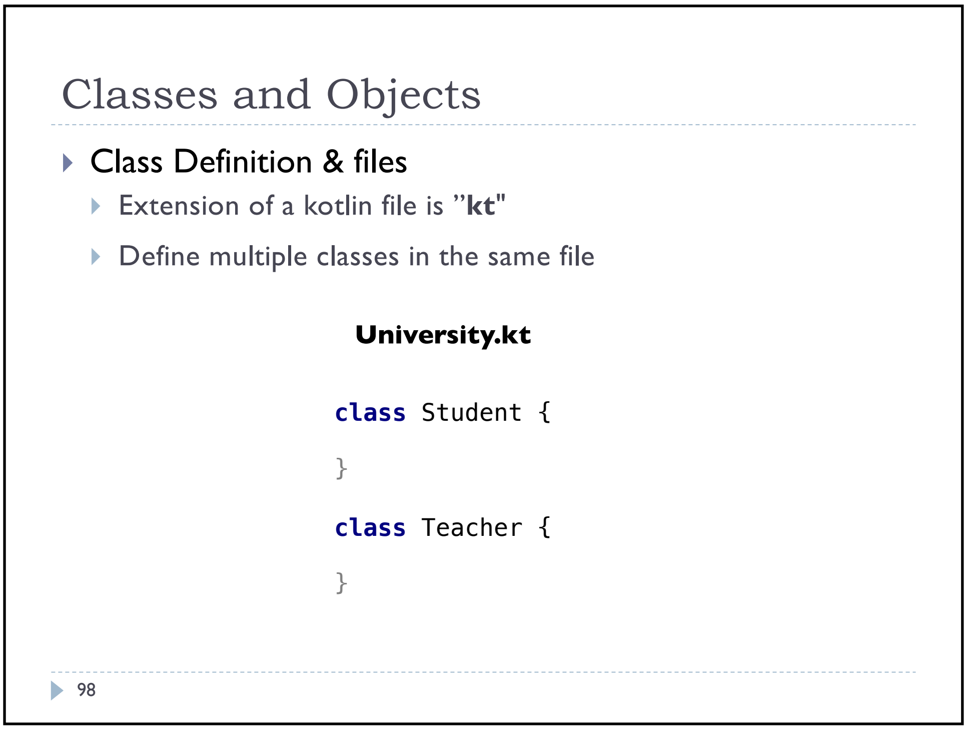
Classes and Objects

- Class Definition

```
class Student {

}
```

  - Added to default package
  - In-fact the curly braces are not required

```
class Student
```



Classes and Objects

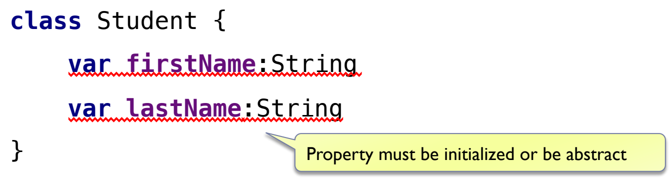
- Class Definition & files
  - Extension of a kotlin file is ”**kt**"
  - Define multiple classes in the same file

University.kt

```
class Student {

}

class Teacher {

}
```

Classes and Objects

- Class with Properties
  - Define the state / attributes of the class
  - Can be mutable (**var**) / immutable (**val**)

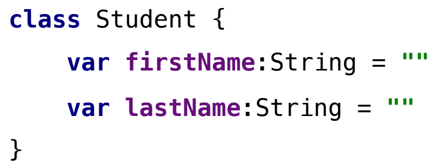


```
class Student {
    var firstName:String
    var lastName:String
}
```

Property must be initialized or be abstract

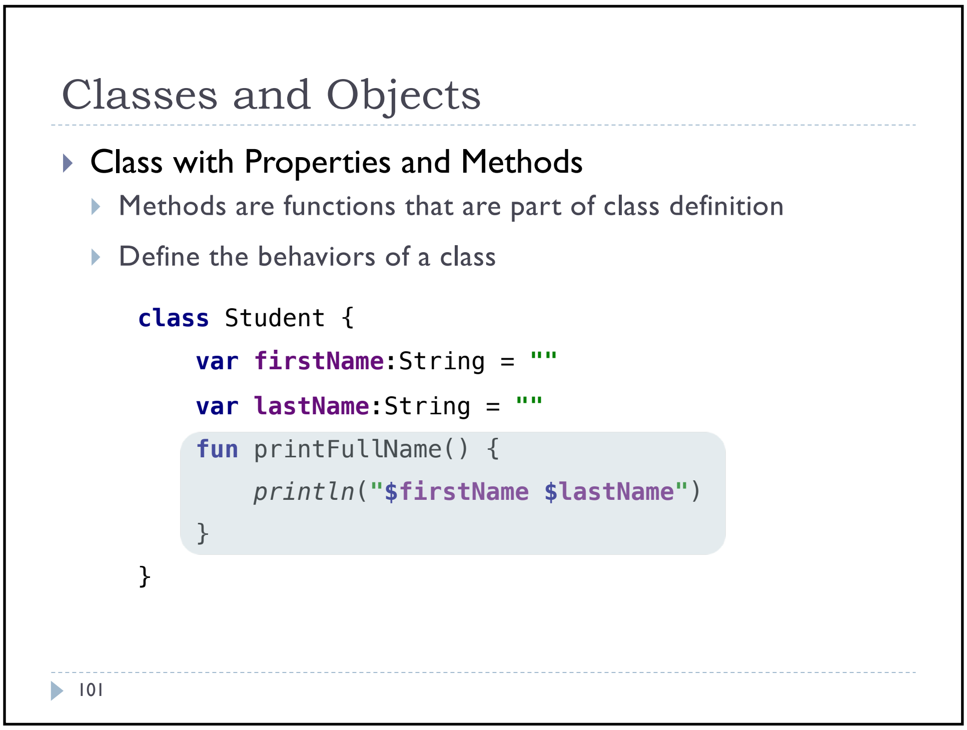
Classes and Objects

- Class with Properties
  - Properties must be initialized
    - As part of declaration



```
class Student {
    var firstName:String = ""
    var lastName:String = ""
}
```

    - Constructor / initializer (more on this later)

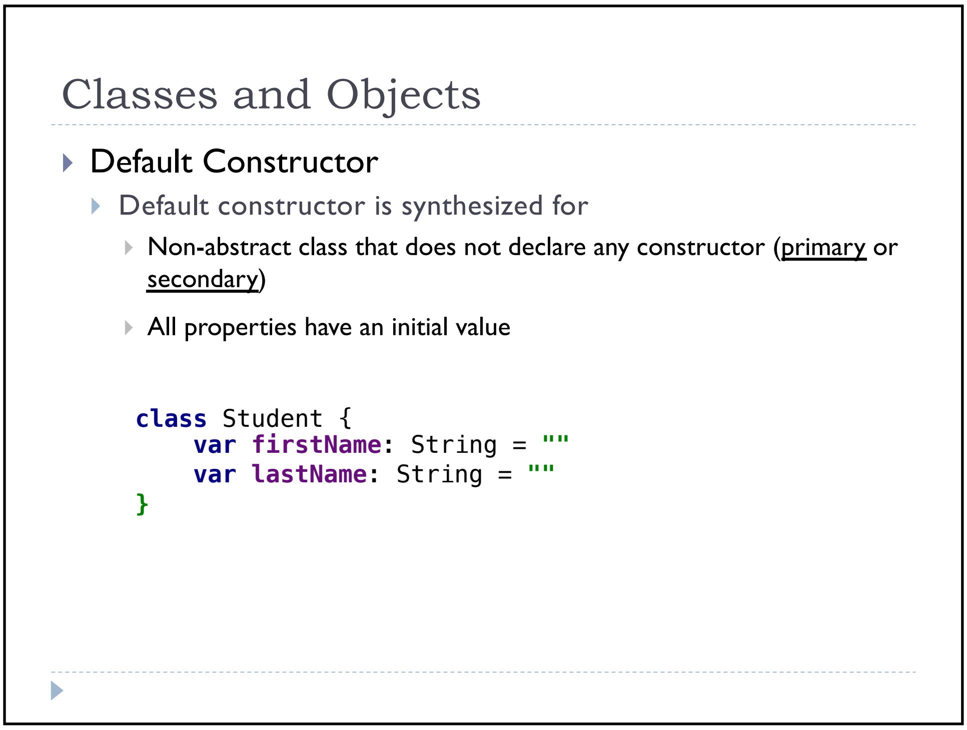


Classes and Objects

- Class with Properties and Methods
  - Methods are functions that are part of class definition
  - Define the behaviors of a class

```
class Student {
    var firstName:String = ""
    var lastName:String = ""
    fun printFullName() {
        println("$firstName $lastName")
    }
}
```

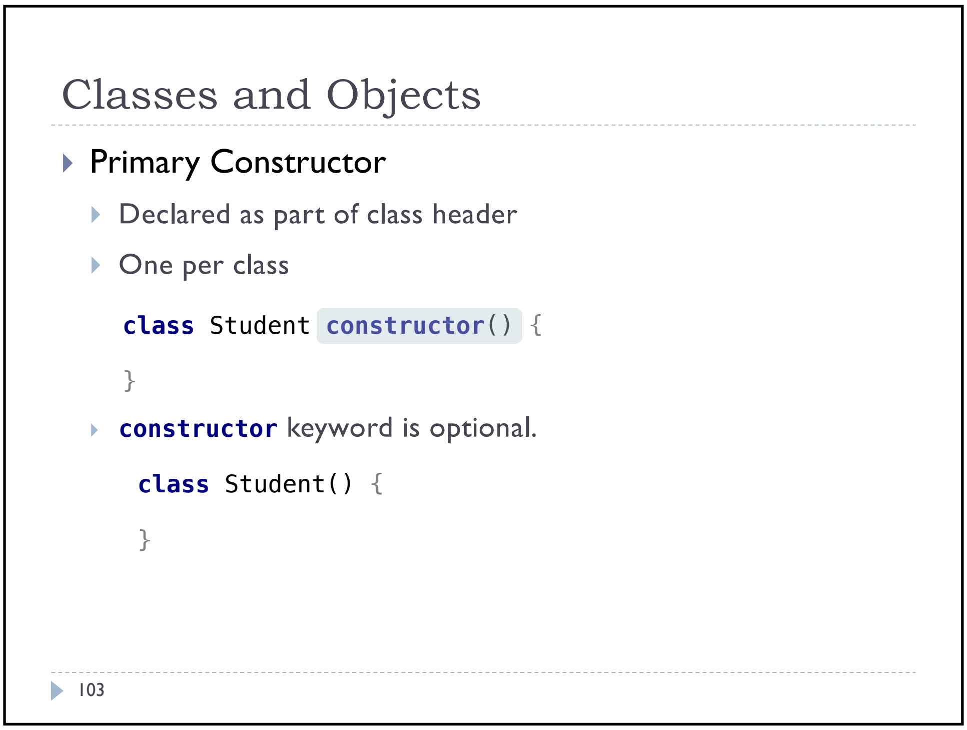
101



Classes and Objects

- Default Constructor
  - Default constructor is synthesized for
    - Non-abstract class that does not declare any constructor (primary or secondary)
    - All properties have an initial value

```
class Student {
    var firstName: String = ""
    var lastName: String = ""
}
```



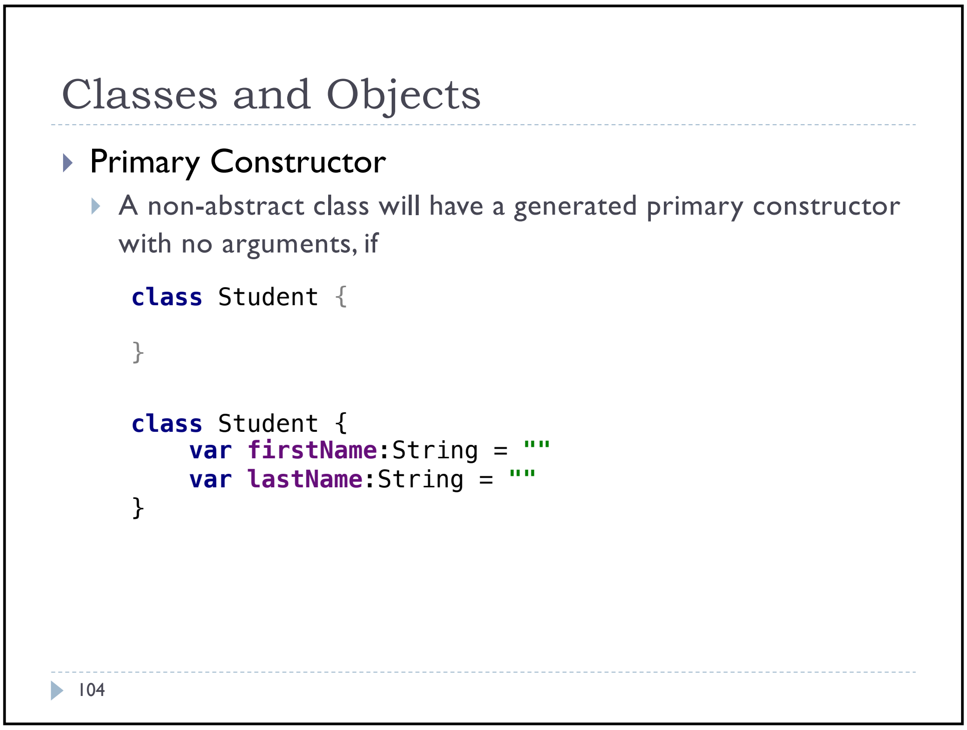
Classes and Objects

- Primary Constructor
  - Declared as part of class header
  - One per class

```
class Student constructor() {

}
```

  - `constructor` keyword is optional.

```
class Student() {

}
```



Classes and Objects

- Primary Constructor
  - A non-abstract class will have a generated primary constructor with no arguments, if

```
class Student {

}

class Student {
    var firstName:String = ""
    var lastName:String = ""
}
```

Classes and Objects

- Primary Constructor with parameters

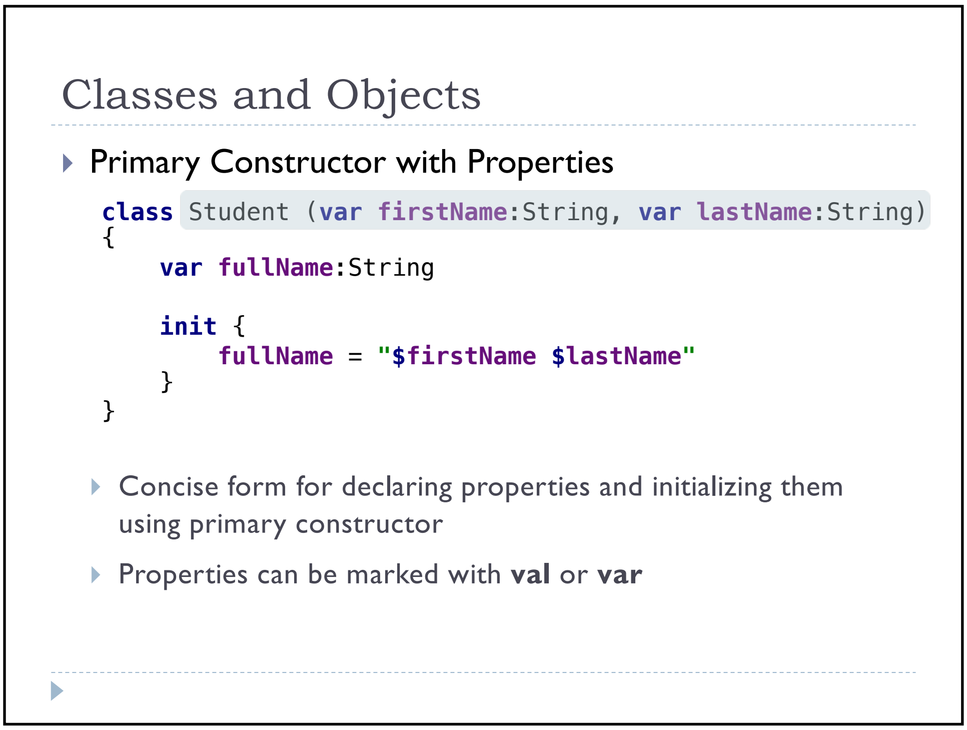
```
class Student (firstName:String, lastName:String) {
    var firstName:String = firstName
    var lastName:String = lastName

    fun printFullName() {
        println("$firstName $lastName")
    }
}
```

Classes and Objects

- Initializer block

```
class Student (firstName:String, lastName:String) {
    var firstName:String = firstName
    var lastName:String = lastName
    var fullName:String

    init {
        fullName = "$firstName $lastName"
    }
}
```

  - Code in initializer blocks becomes part of the primary constructor
  - Has access to parameters of primary constructors

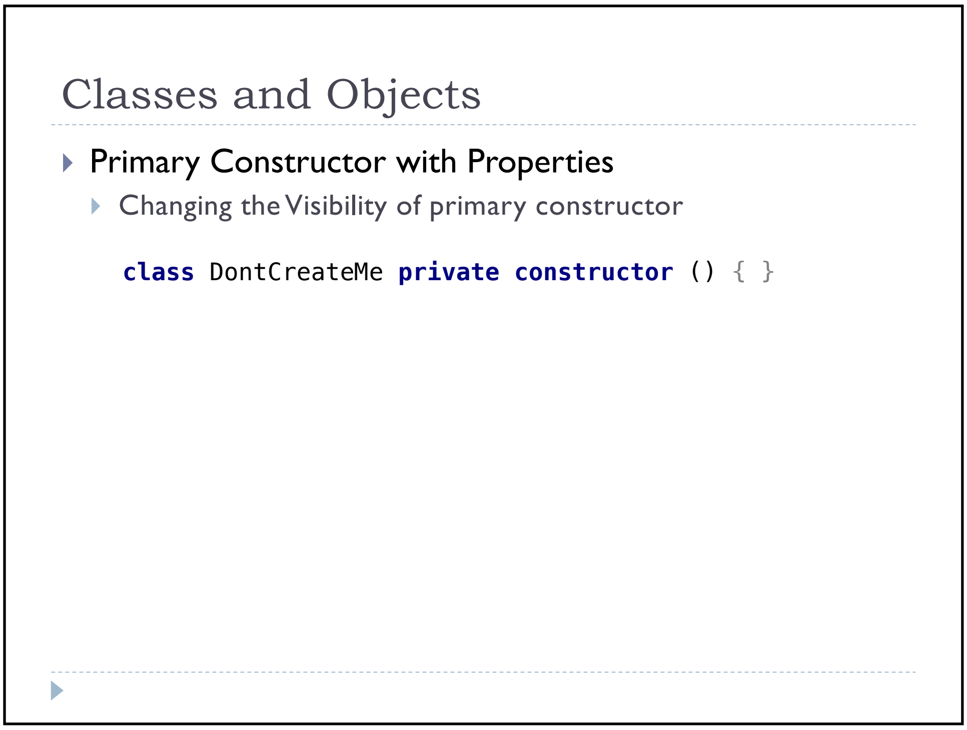


Classes and Objects

- Primary Constructor with Properties

```
class Student (var firstName:String, var lastName:String)
{
    var fullName:String

    init {
        fullName = "$firstName $lastName"
    }
}
```

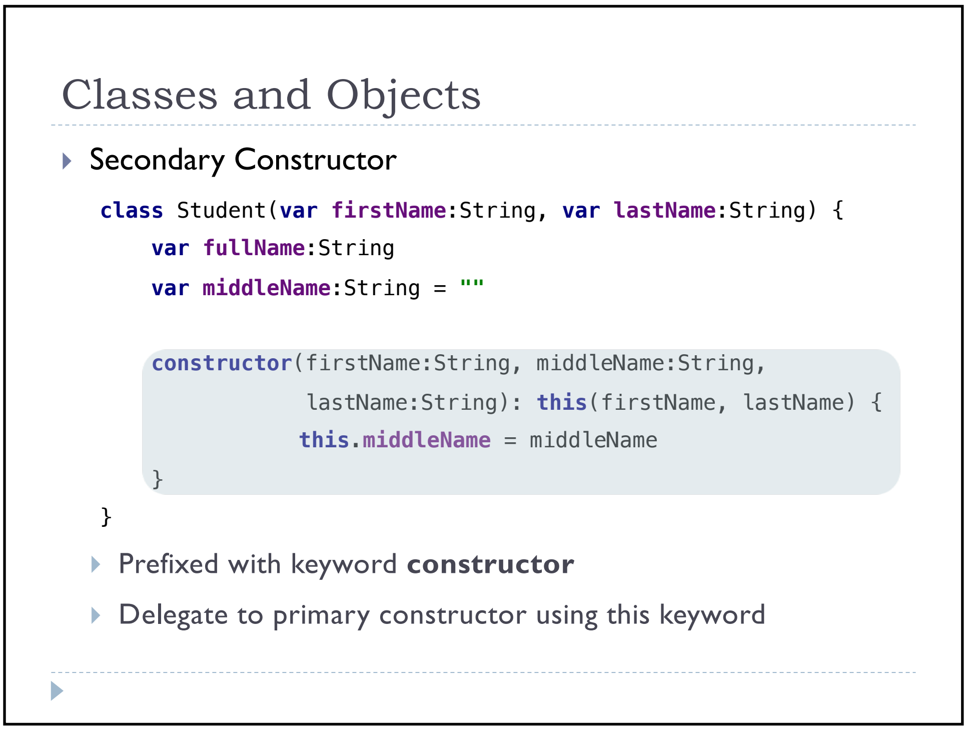
  - Concise form for declaring properties and initializing them using primary constructor
  - Properties can be marked with **val** or **var**



Classes and Objects

- Primary Constructor with Properties
  - Changing the Visibility of primary constructor

```
class DontCreateMe private constructor () { }
```

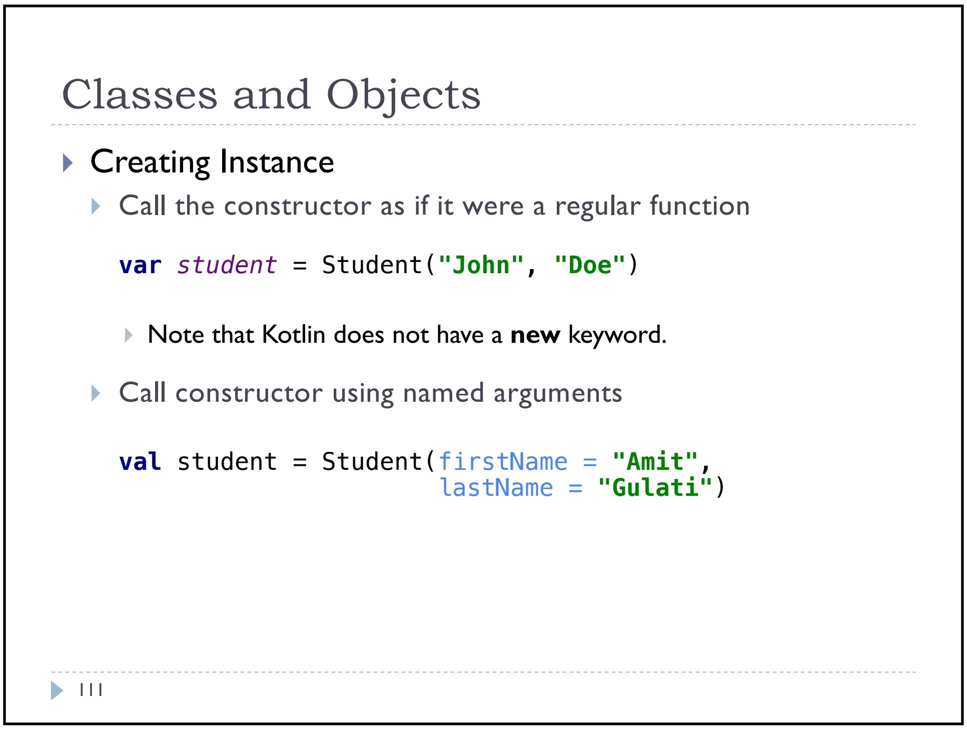


Classes and Objects

- Secondary Constructor

```
class Student(var firstName:String, var lastName:String) {
    var fullName:String
    var middleName:String = ""

    constructor(firstName:String, middleName:String,
                lastName:String): this(firstName, lastName) {
               this.middleName = middleName
    }
}
```

  - Prefixed with keyword **constructor**
  - Delegate to primary constructor using this keyword



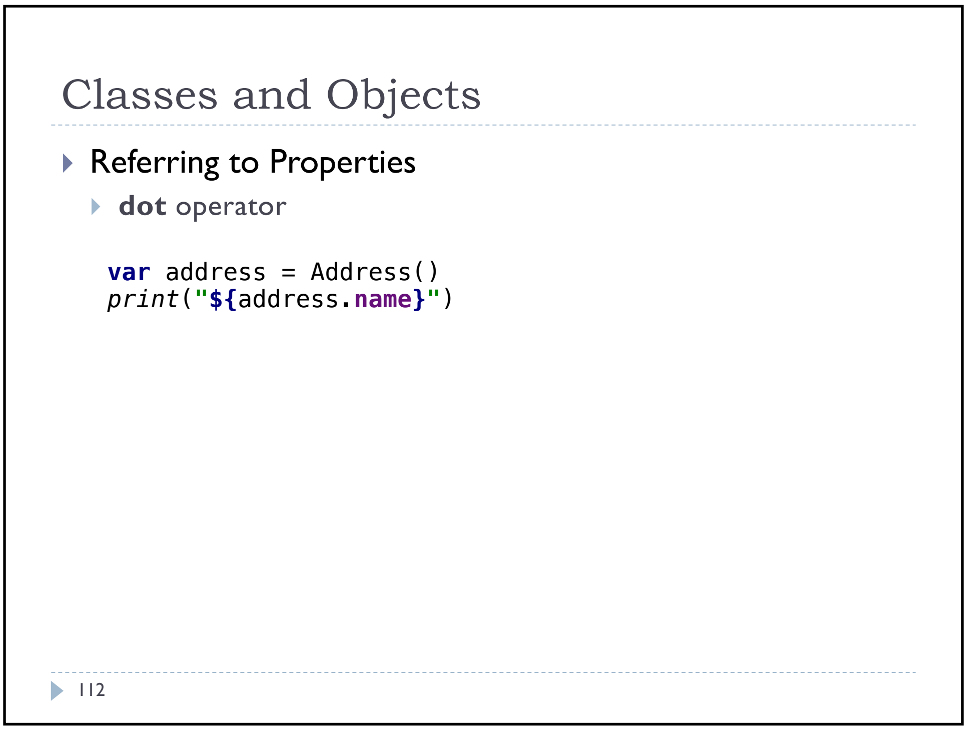
Classes and Objects

- Creating Instance
  - Call the constructor as if it were a regular function

```
var student = Student("John", "Doe")
```

    - Note that Kotlin does not have a **new** keyword.
  - Call constructor using named arguments

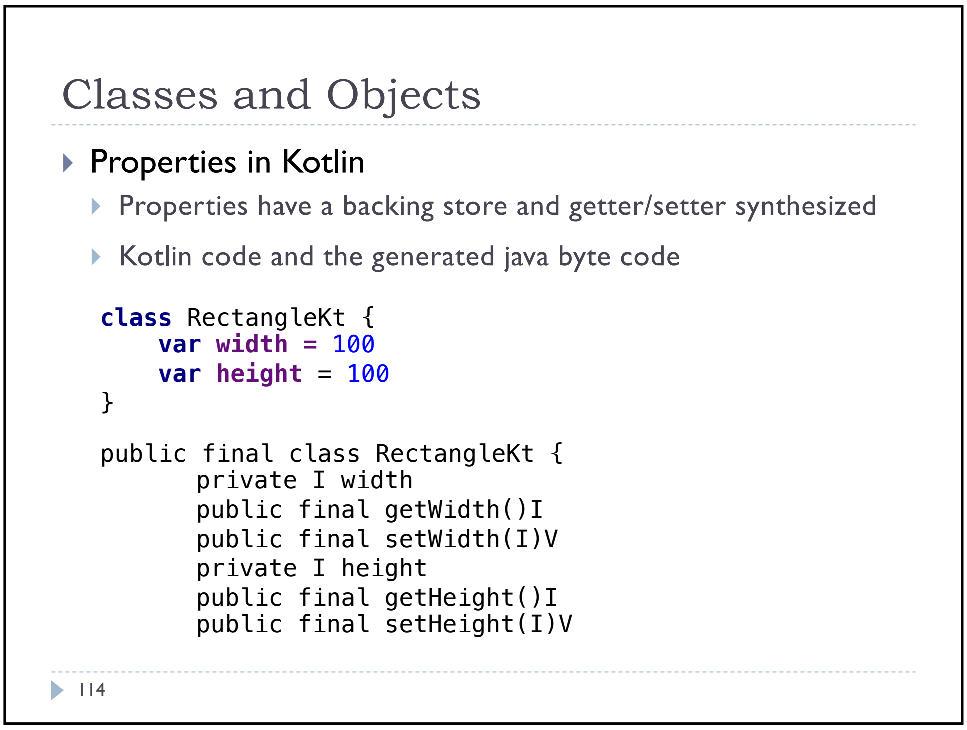
```
val student = Student(firstName = "Amit",
                      lastName = "Gulati")
```



Classes and Objects

- Referring to Properties
  - **dot** operator

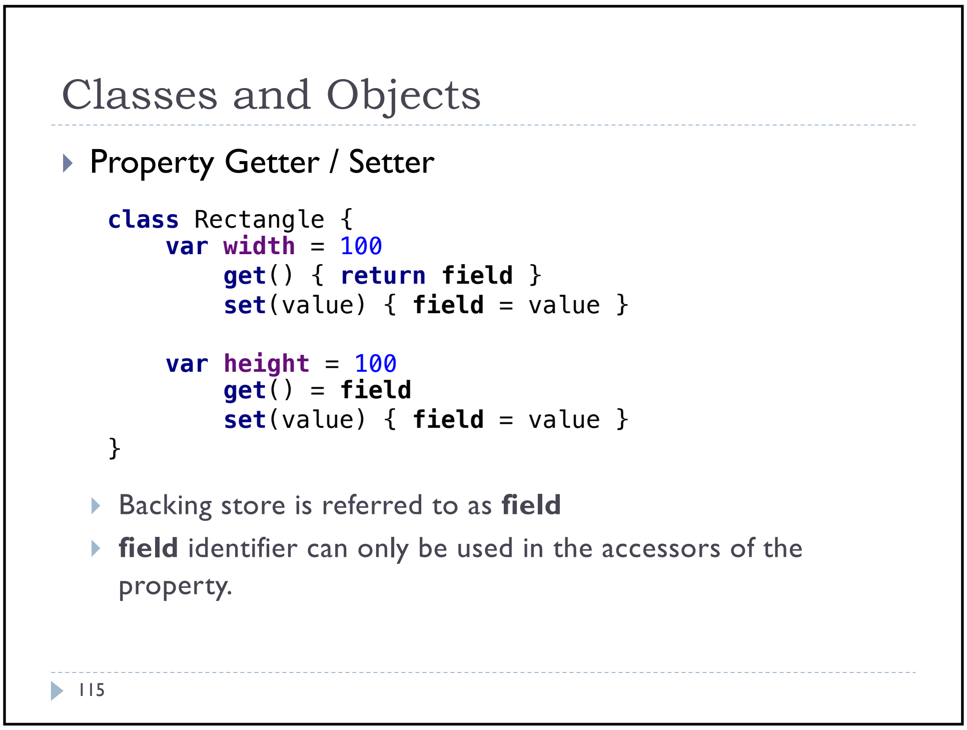
```
var address = Address()
print("${address.name}")
```



Classes and Objects

- Properties in Kotlin
  - Properties have a backing store and getter/setter synthesized
  - Kotlin code and the generated java byte code

```
class RectangleKt {
    var width = 100
    var height = 100
}

public final class RectangleKt {
        private I width
        public final getWidth()I
        public final setWidth(I)V
        private I height
        public final getHeight()I
        public final setHeight(I)V
```

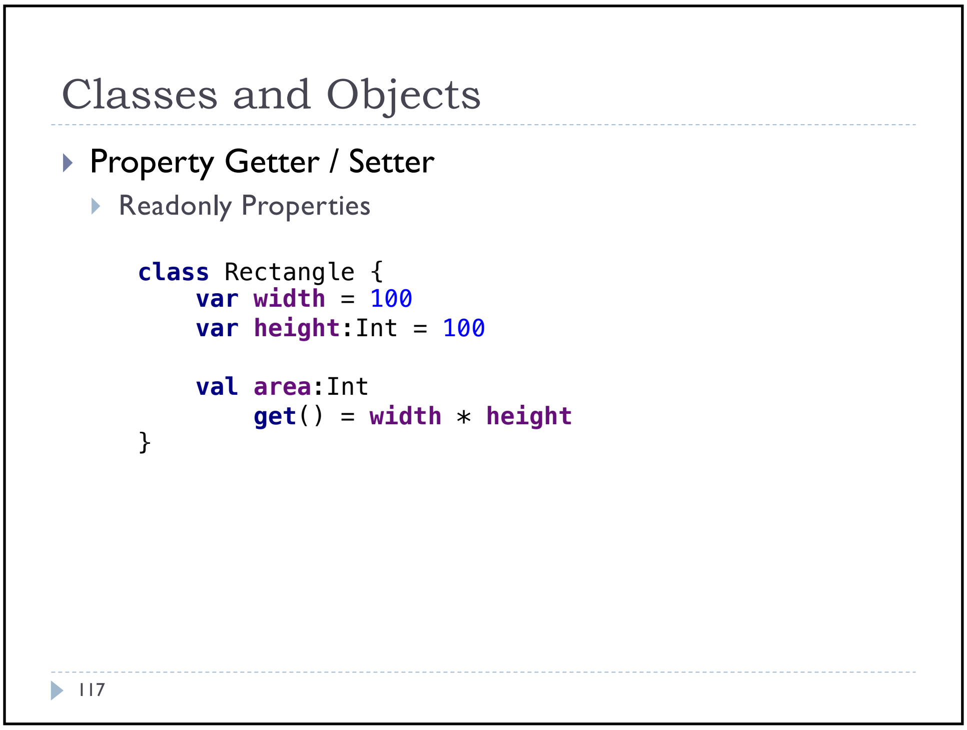


Classes and Objects

- Property Getter / Setter

```
class Rectangle {
    var width = 100
        get() { return field }
        set(value) { field = value }

    var height = 100
        get() = field
        set(value) { field = value }
}
```

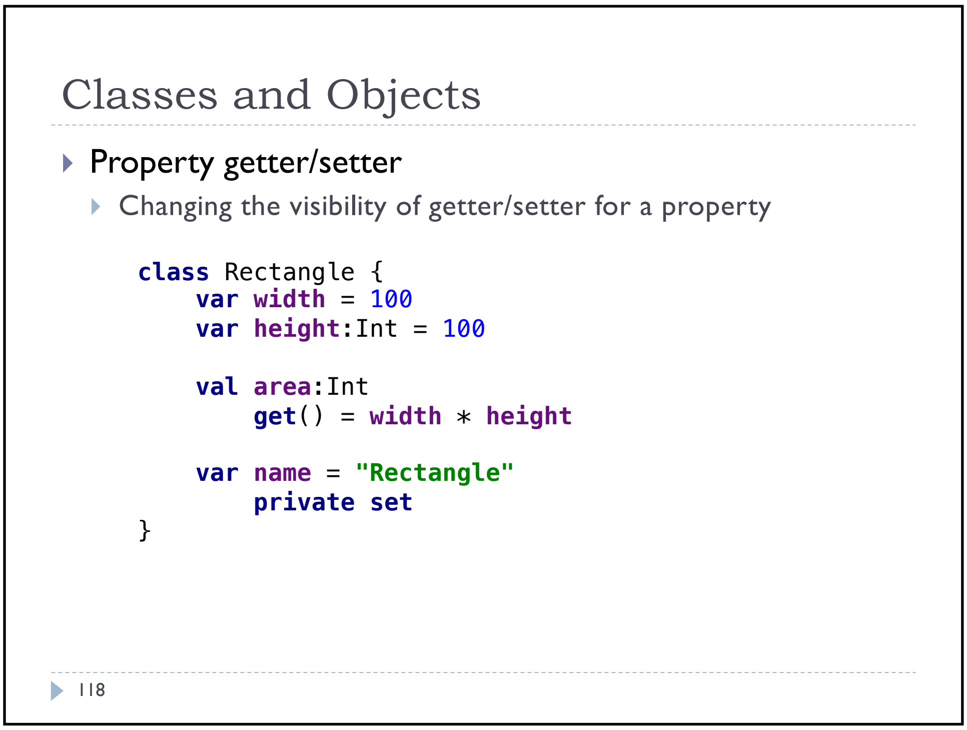
  - Backing store is referred to as **field**
  - **field** identifier can only be used in the accessors of the property.



Classes and Objects

- Property Getter / Setter
  - Readonly Properties

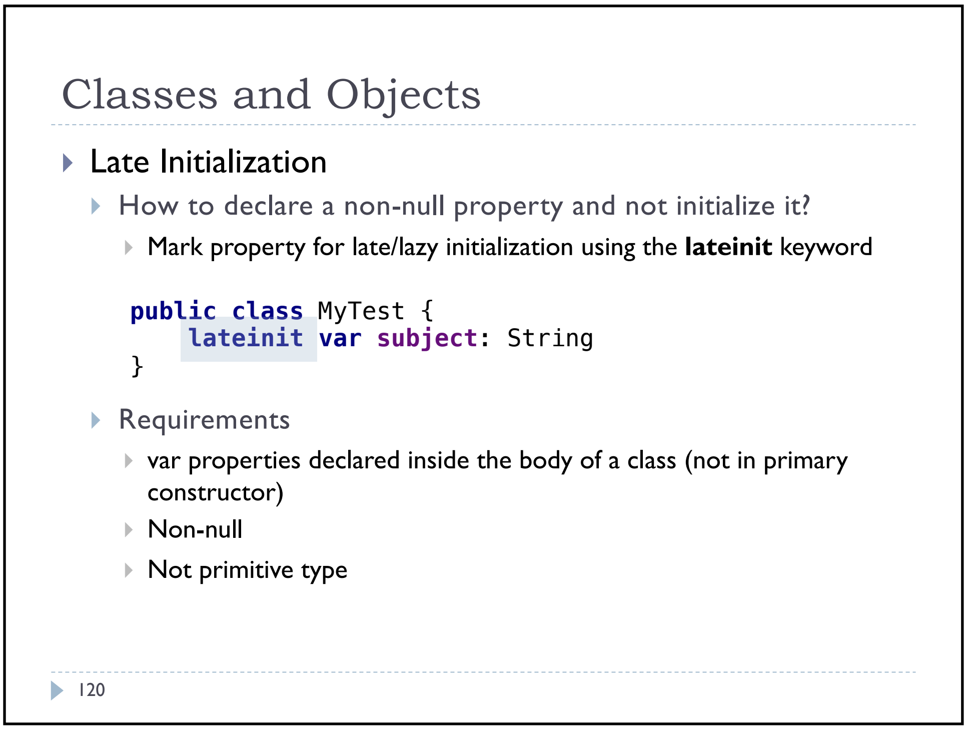
```
class Rectangle {
    var width = 100
    var height:Int = 100

    val area:Int
        get() = width * height
}
```



Classes and Objects

- Property getter/setter
  - Changing the visibility of getter/setter for a property

```
class Rectangle {
    var width = 100
    var height:Int = 100

    val area:Int
        get() = width * height

    var name = "Rectangle"
        private set
}
```

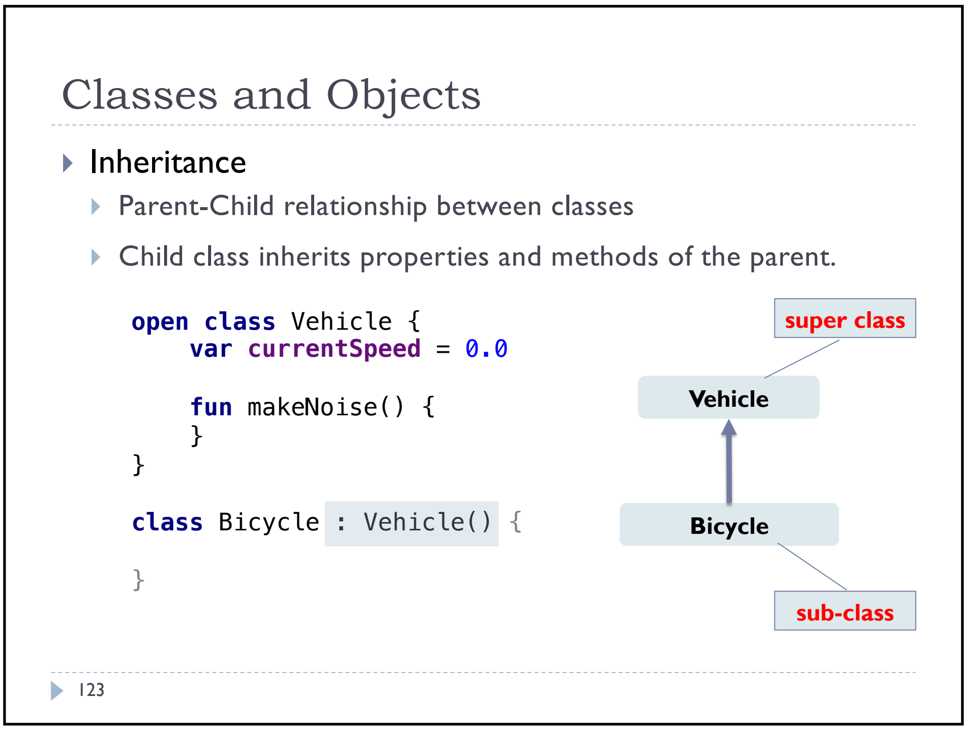


Classes and Objects

- Late Initialization
  - How to declare a non-null property and not initialize it?
    - Mark property for late/lazy initialization using the **lateinit** keyword

```
public class MyTest {
    lateinit var subject: String
}
```

  - Requirements
    - var properties declared inside the body of a class (not in primary constructor)
    - Non-null
    - Not primitive type



Classes and Objects

- Inheritance
  - Parent-Child relationship between classes
  - Child class inherits properties and methods of the parent.

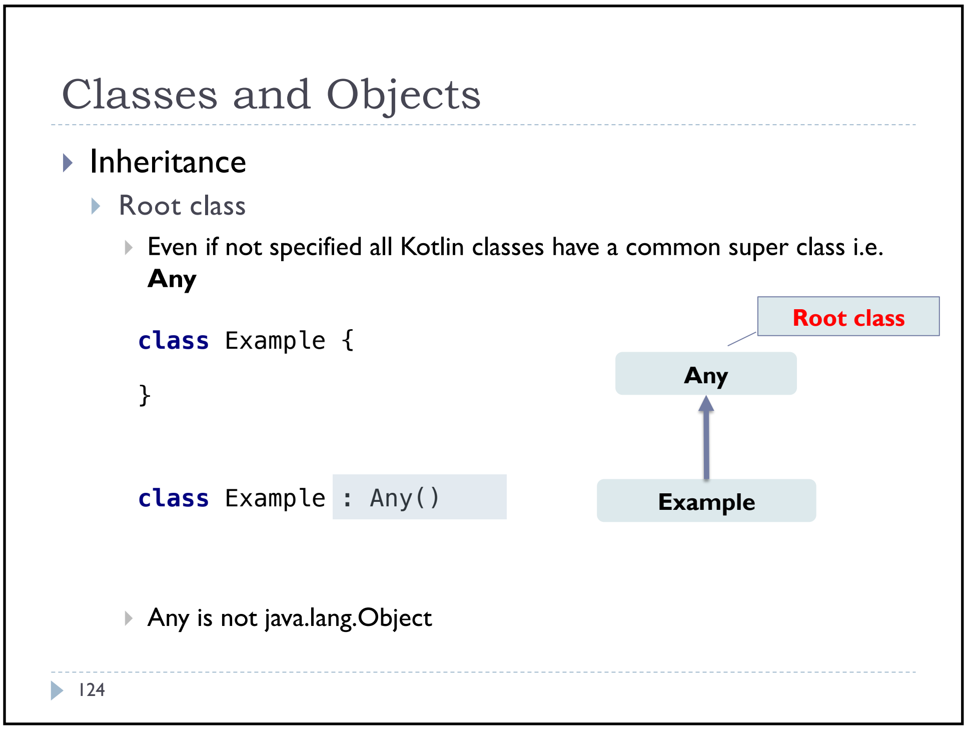
```
open class Vehicle {
    var currentSpeed = 0.0

    fun makeNoise() {
    }
}

class Bicycle : Vehicle() {

}
```

super class

Vehicle

Bicycle

sub-class



Classes and Objects

- Inheritance
  - Root class
    - Even if not specified all Kotlin classes have a common super class i.e. **Any**

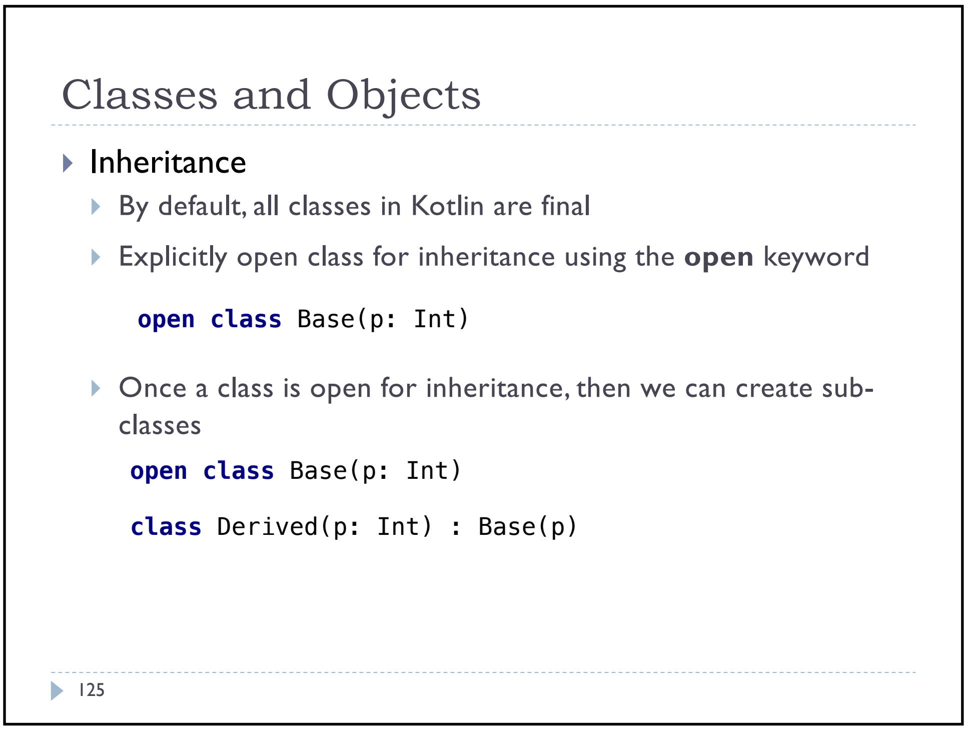
```
class Example {

}


class Example : Any()
```

Root class

Any

Example

- Any is not java.lang.Object



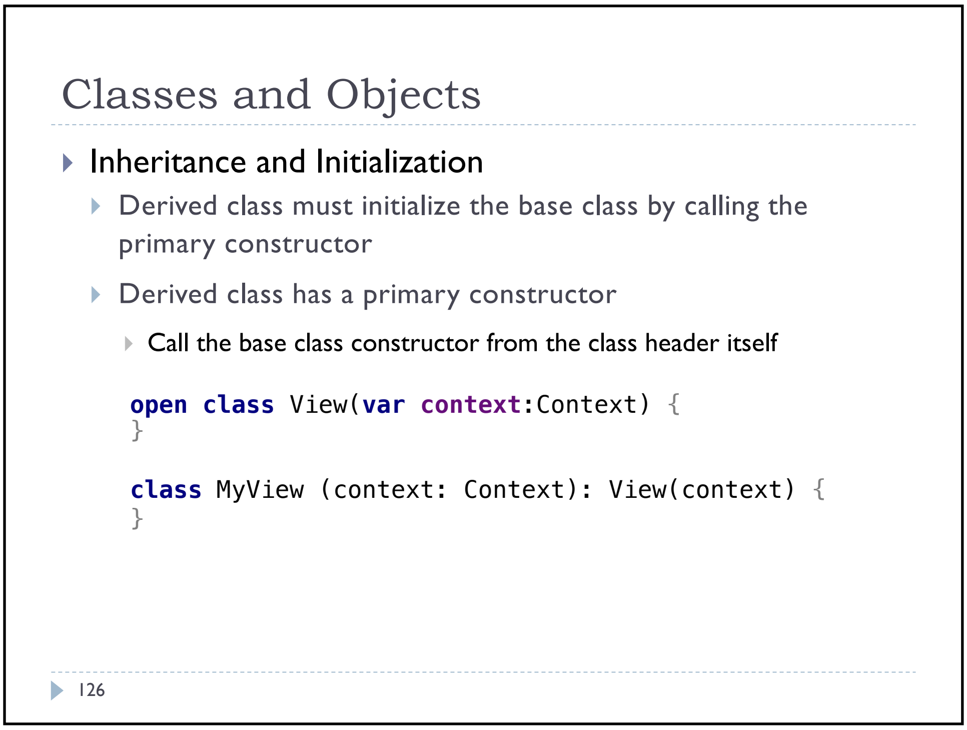
Classes and Objects

- Inheritance
  - By default, all classes in Kotlin are final
  - Explicitly open class for inheritance using the **open** keyword

```
open class Base(p: Int)
```

  - Once a class is open for inheritance, then we can create sub-classes

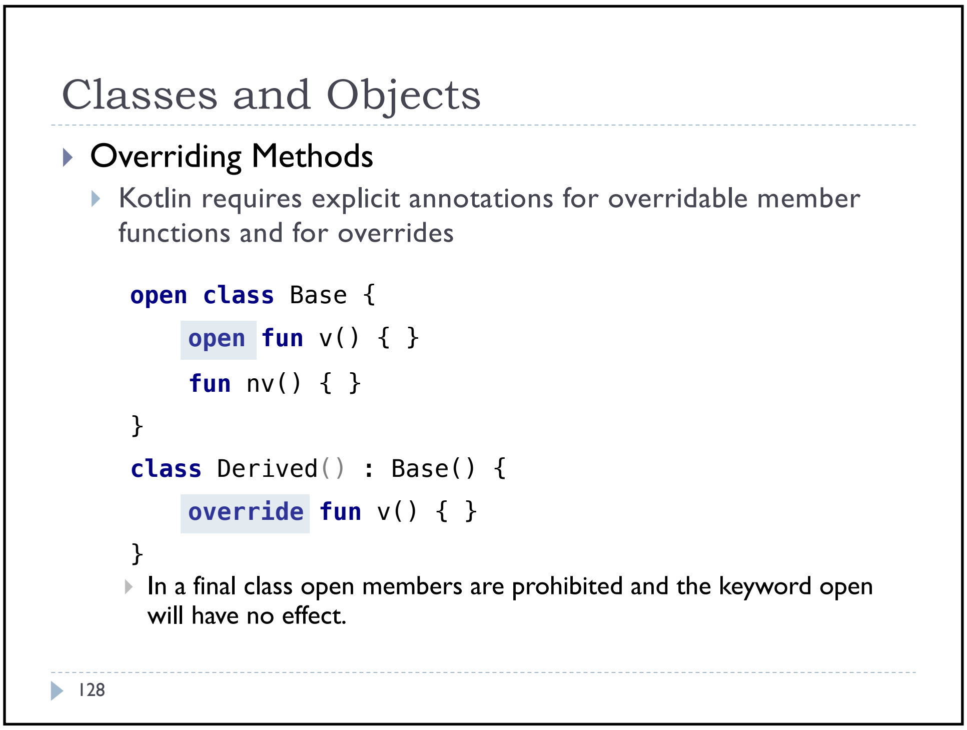
```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```



Classes and Objects

- Inheritance and Initialization
  - Derived class must initialize the base class by calling the primary constructor
  - Derived class has a primary constructor
    - Call the base class constructor from the class header itself

```
open class View(var context:Context) {
}

class MyView (context: Context): View(context) {
}
```

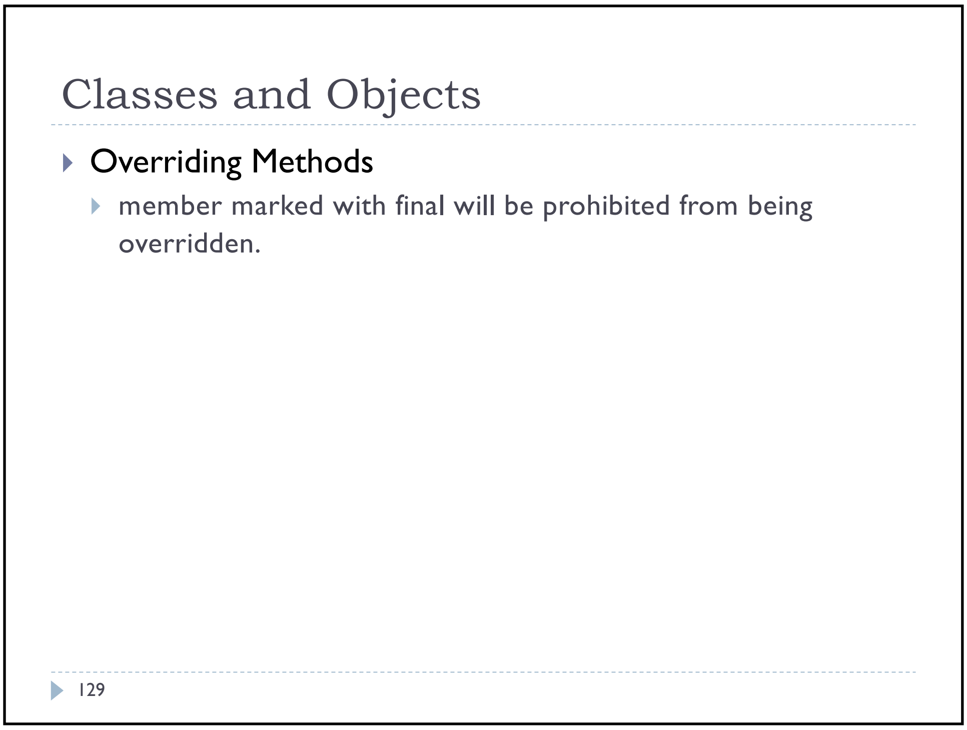


Classes and Objects

- Overriding Methods
  - Kotlin requires explicit annotations for overridable member functions and for overrides

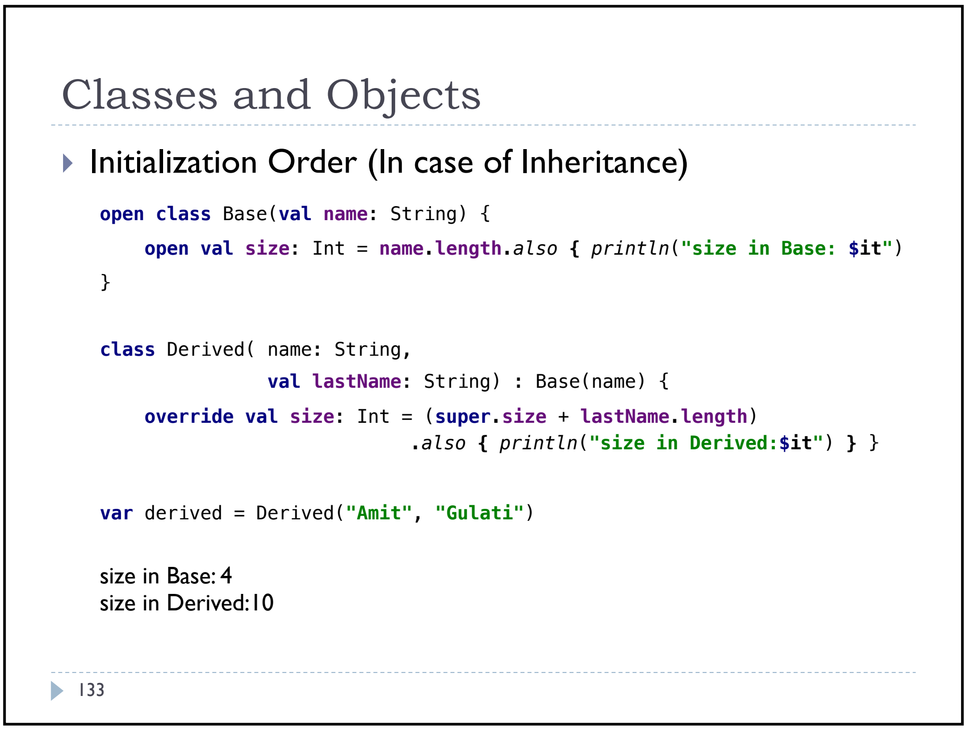
```
open class Base {
    open fun v() { }
    fun nv() { }
}
class Derived() : Base() {
    override fun v() { }
}
```

- In a final class open members are prohibited and the keyword open will have no effect.



Classes and Objects

- Overriding Methods
  - member marked with final will be prohibited from being overridden.

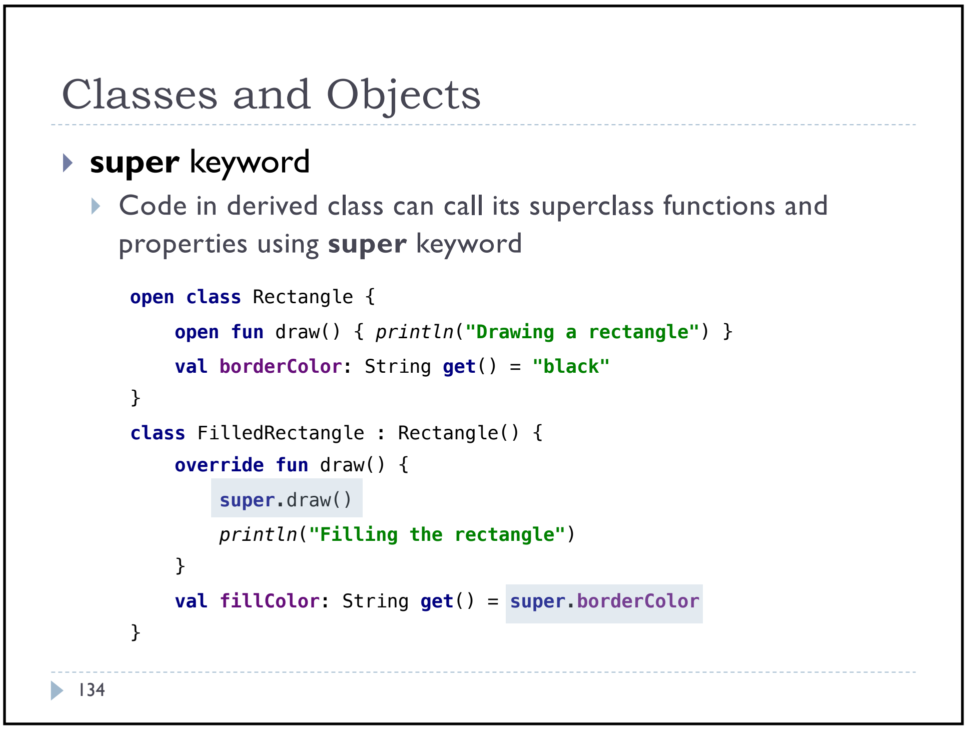


Classes and Objects

- Initialization Order (In case of Inheritance)

```
open class Base(val name: String) {
    open val size: Int = name.length.also { println("size in Base: $it") }
}

class Derived( name: String,
               val lastName: String) : Base(name) {
    override val size: Int = (super.size + lastName.length)
                               .also { println("size in Derived:$it") } }

var derived = Derived("Amit", "Gulati")
```

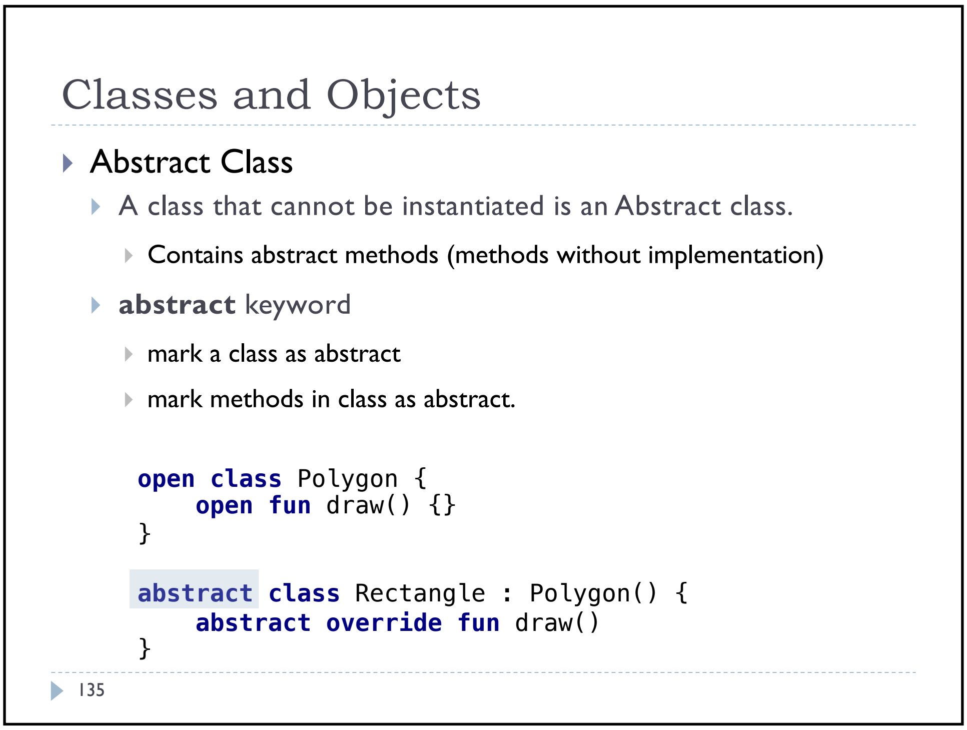
size in Base: 4
size in Derived:10



Classes and Objects

- **super** keyword
  - Code in derived class can call its superclass functions and properties using **super** keyword

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}
class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }
    val fillColor: String get() = super.borderColor
}
```

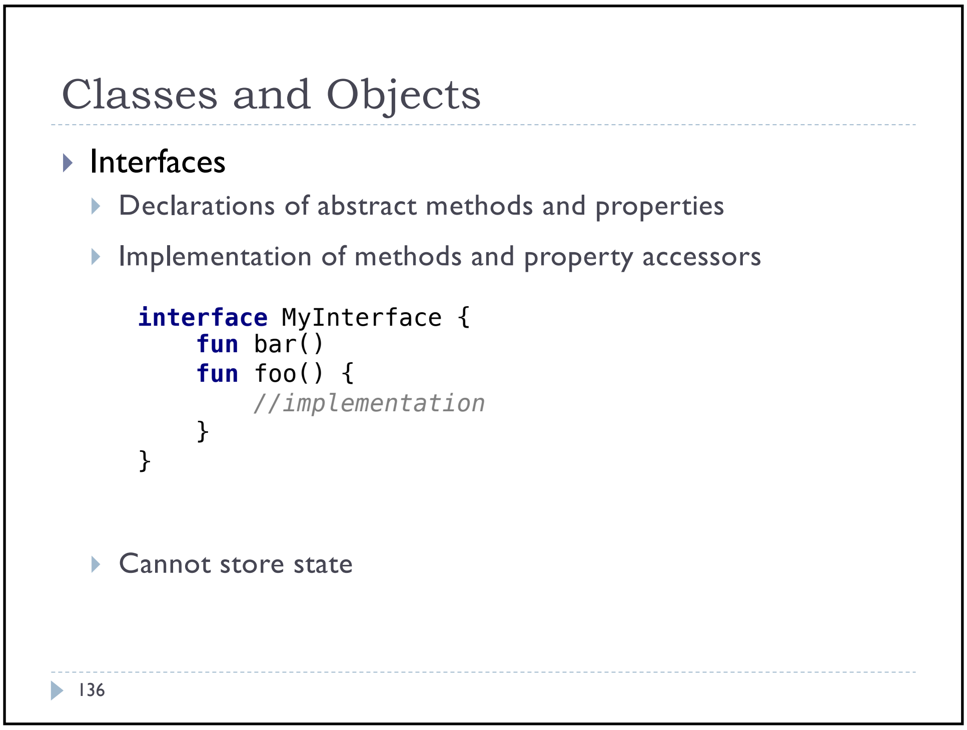


Classes and Objects

- Abstract Class
  - A class that cannot be instantiated is an Abstract class.
    - Contains abstract methods (methods without implementation)
  - **abstract** keyword
    - mark a class as abstract
    - mark methods in class as abstract.

```
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle : Polygon() {
    abstract override fun draw()
}
```

135



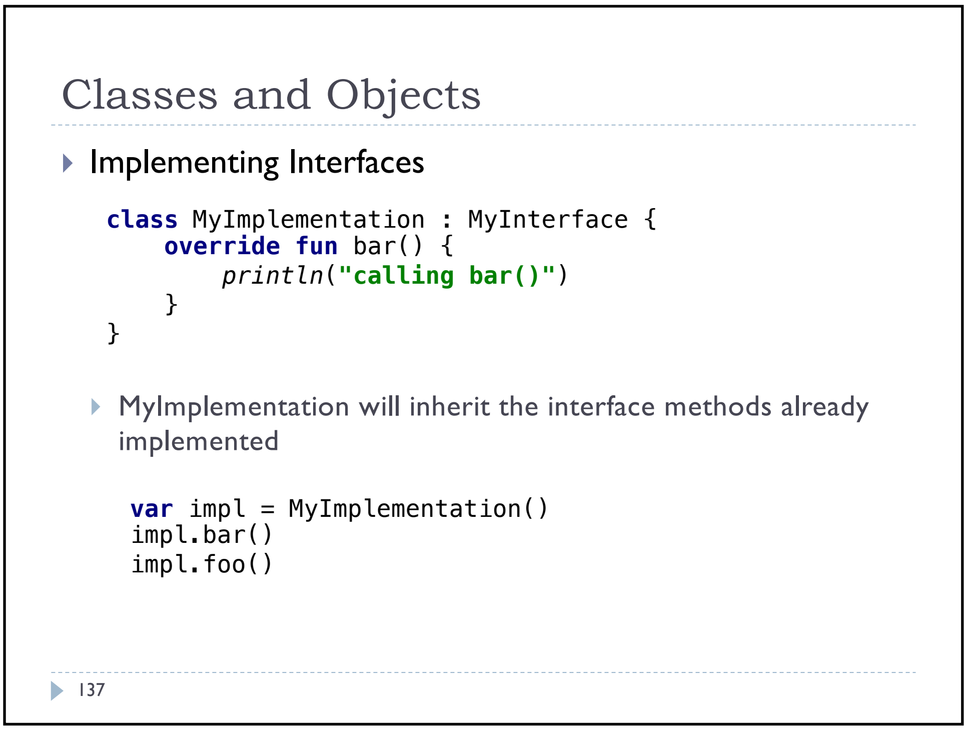
Classes and Objects

- Interfaces
  - Declarations of abstract methods and properties
  - Implementation of methods and property accessors

```
interface MyInterface {
    fun bar()
    fun foo() {
        //implementation
    }
}
```

  - Cannot store state

136



Classes and Objects

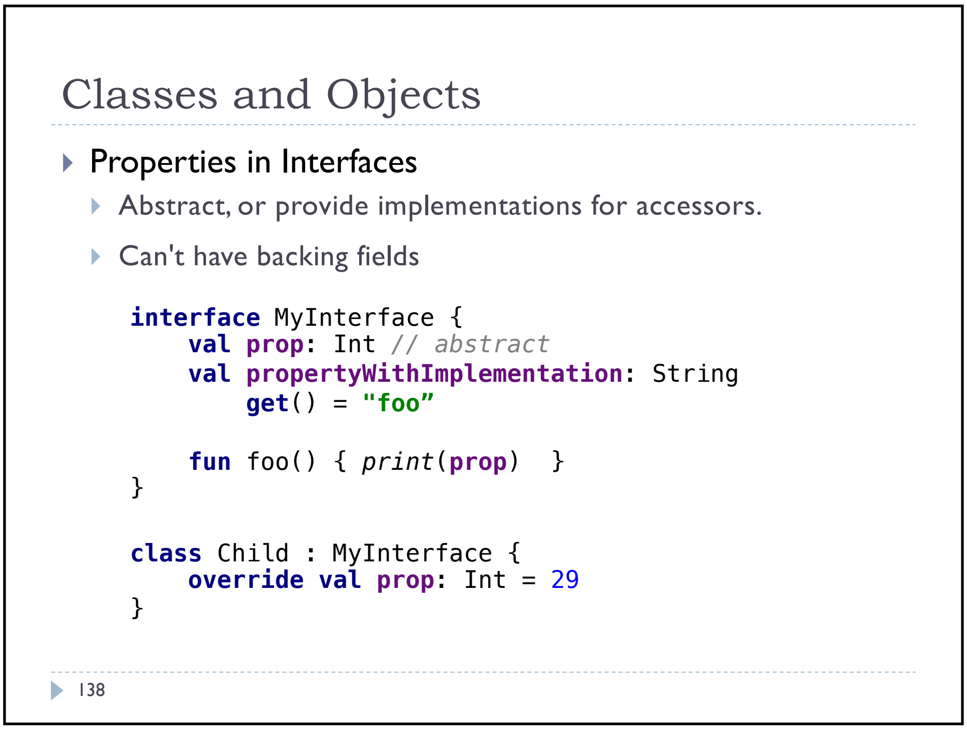
- Implementing Interfaces

```
class MyImplementation : MyInterface {
    override fun bar() {
        println("calling bar()")
    }
}
```

  - MyImplementation will inherit the interface methods already implemented

```
var impl = MyImplementation()
impl.bar()
impl.foo()
```

137

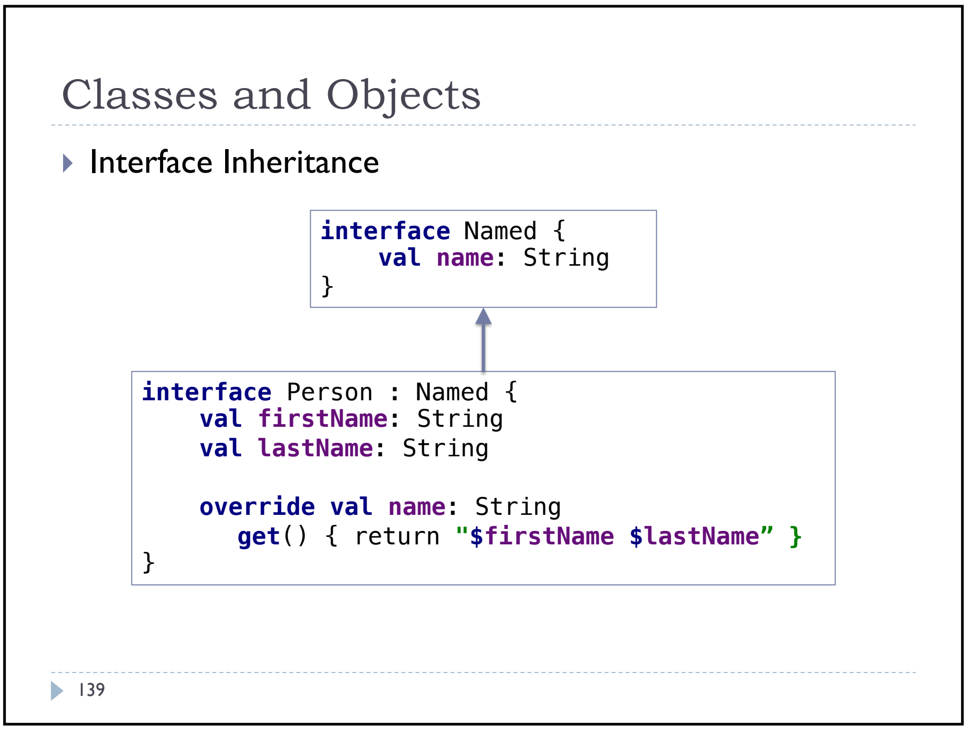


Classes and Objects

- Properties in Interfaces
  - Abstract, or provide implementations for accessors.
  - Can't have backing fields

```
interface MyInterface {
    val prop: Int // abstract
    val propertyWithImplementation: String
        get() = "foo”

    fun foo() { print(prop)  }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

138

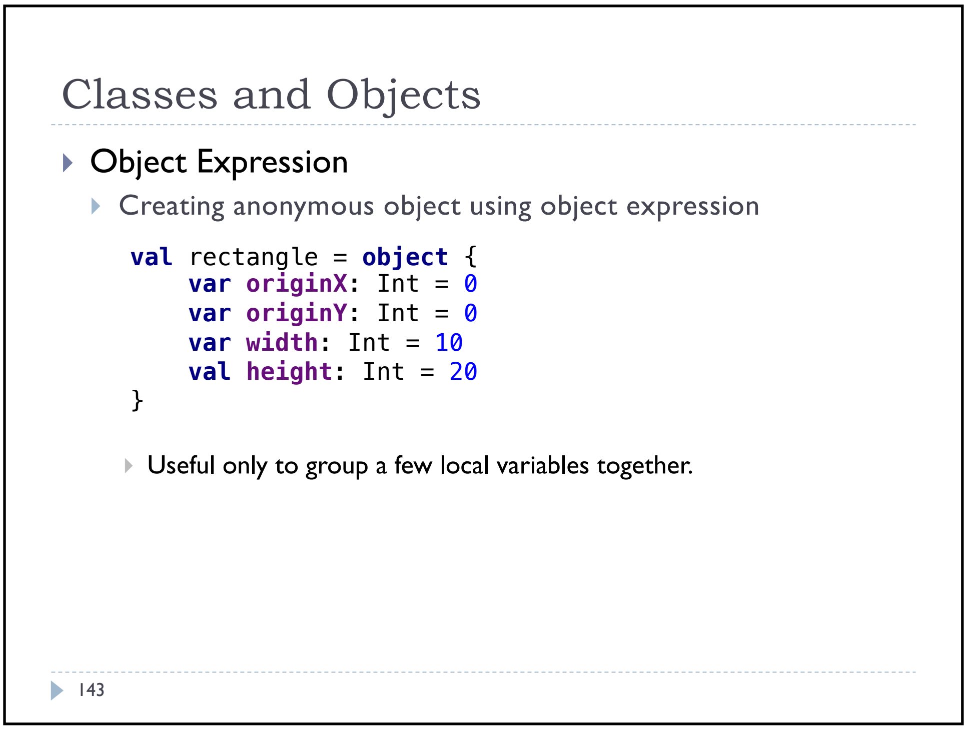


Classes and Objects

- Interface Inheritance

```
interface Named {
    val name: String
}
```

```
interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String
       get() { return "$firstName $lastName” }
}
```

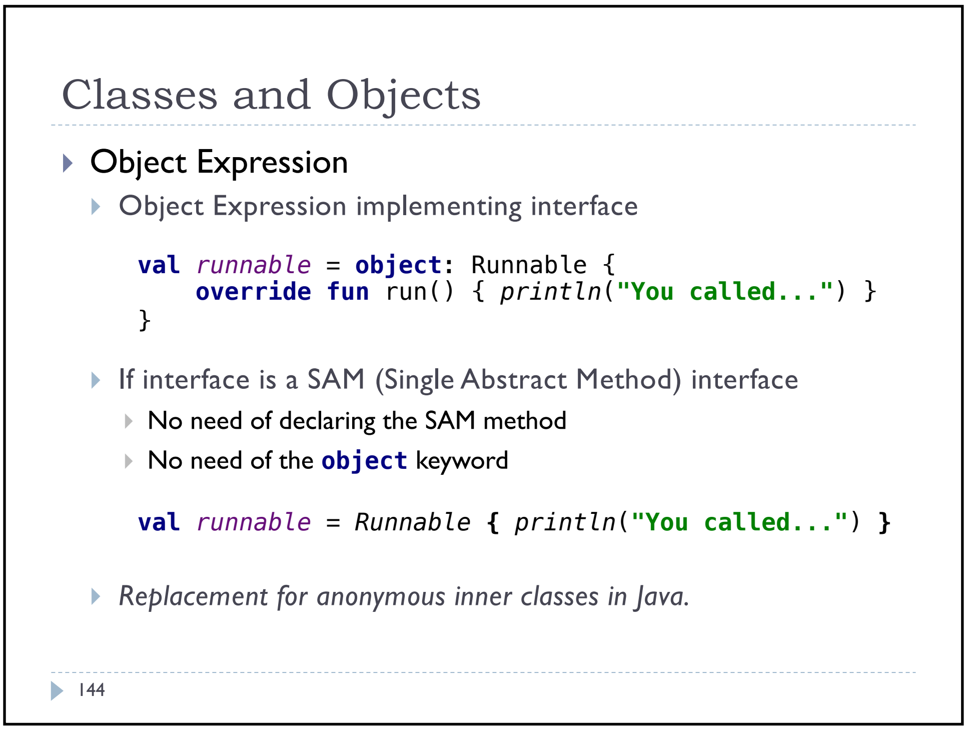


Classes and Objects

- Object Expression
  - Creating anonymous object using object expression

```
val rectangle = object {
    var originX: Int = 0
    var originY: Int = 0
    var width: Int = 10
    val height: Int = 20
}
```

  - Useful only to group a few local variables together.



Classes and Objects

- Object Expression
  - Object Expression implementing interface

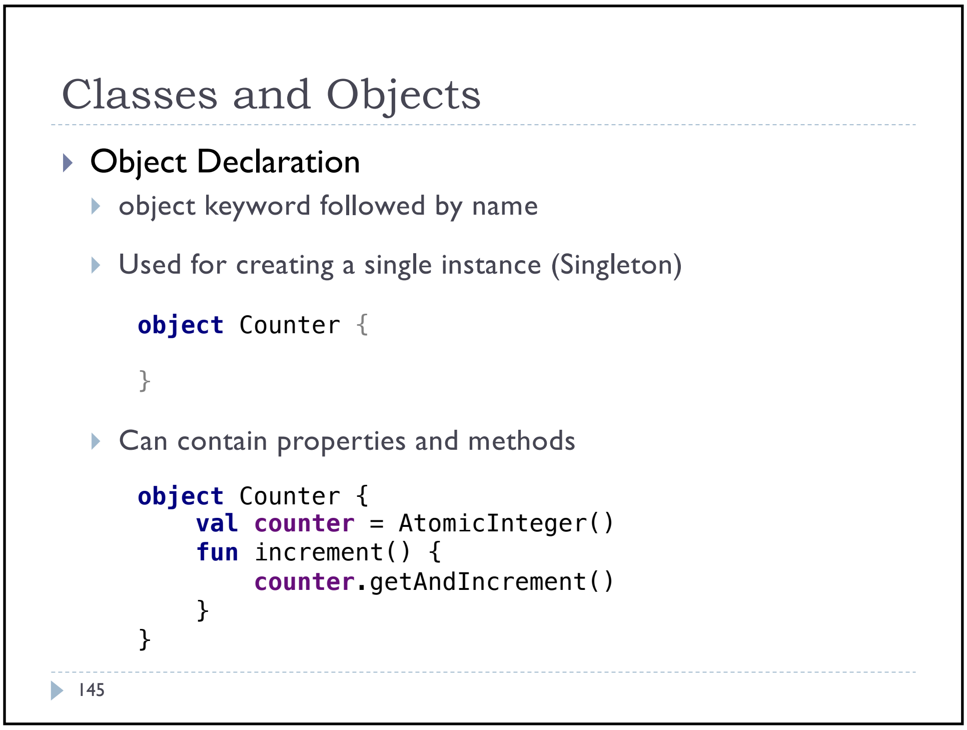
```
val runnable = object: Runnable {
    override fun run() { println("You called...") }
}
```

  - If interface is a SAM (Single Abstract Method) interface
    - No need of declaring the SAM method
    - No need of the `object` keyword

```
val runnable = Runnable { println("You called...") }
```

  - *Replacement for anonymous inner classes in Java.*

144



Classes and Objects

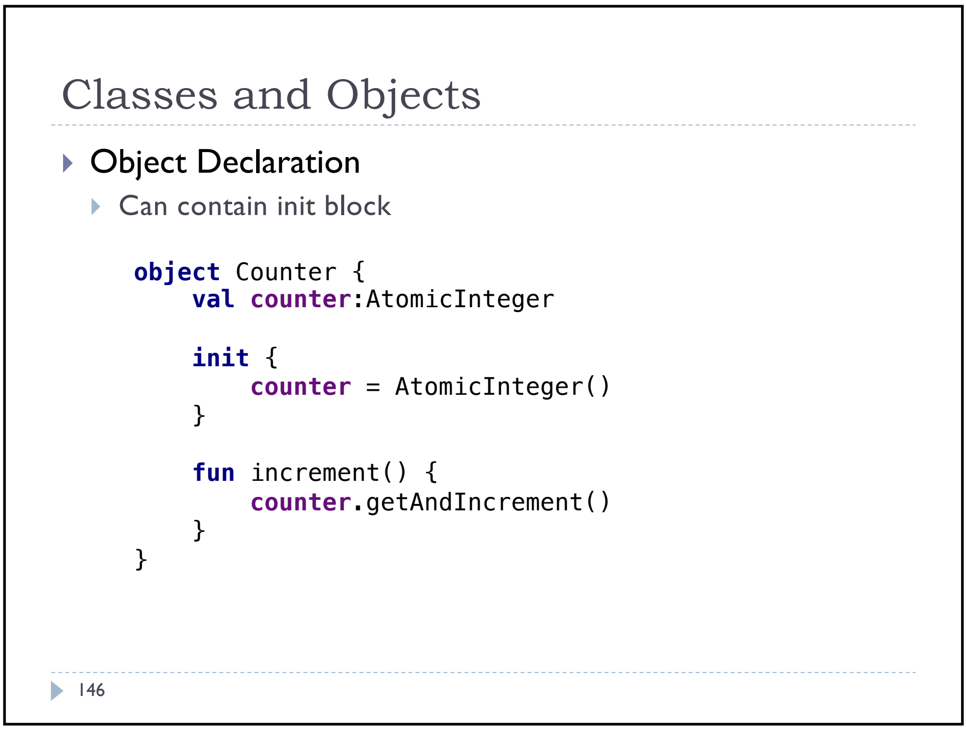
- Object Declaration
  - object keyword followed by name
  - Used for creating a single instance (Singleton)

```
object Counter {

}
```

  - Can contain properties and methods

```
object Counter {
    val counter = AtomicInteger()
    fun increment() {
        counter.getAndIncrement()
    }
}
```

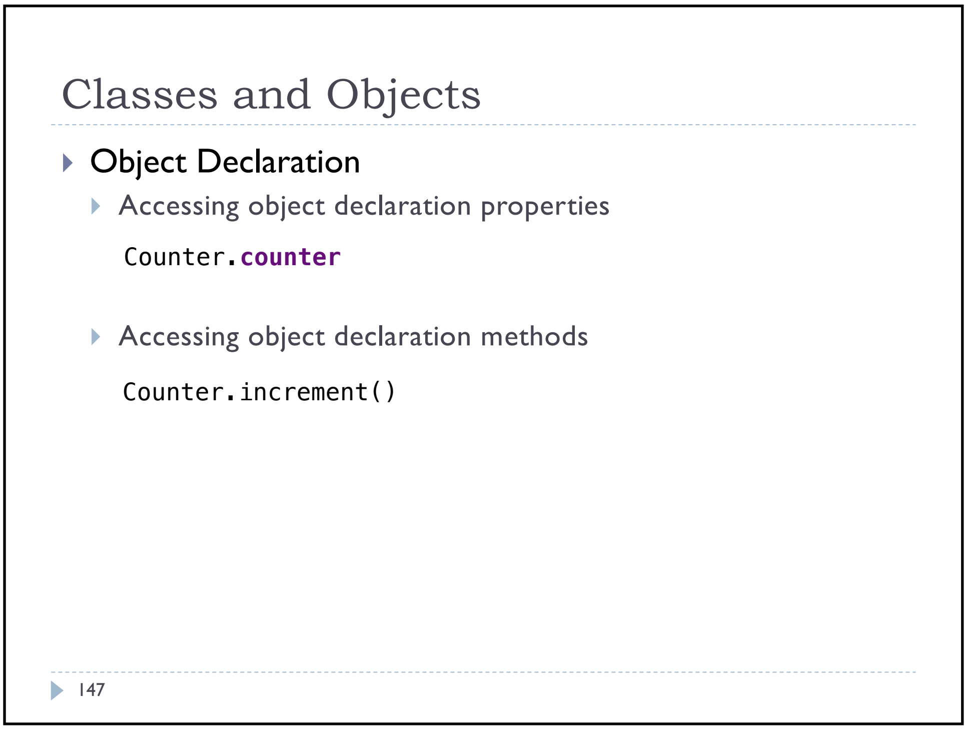
145



Classes and Objects

- Object Declaration
  - Can contain init block

```
object Counter {
    val counter:AtomicInteger

    init {
        counter = AtomicInteger()
    }

    fun increment() {
        counter.getAndIncrement()
    }
}
```



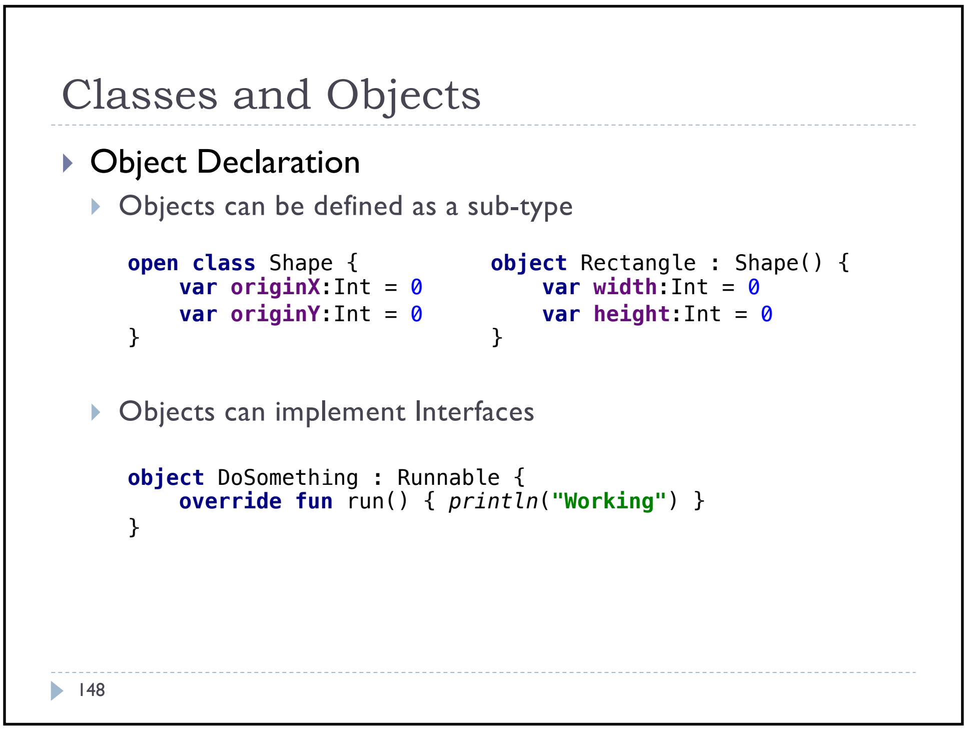
Classes and Objects

- Object Declaration
  - Accessing object declaration properties

```
Counter.counter
```

  - Accessing object declaration methods

```
Counter.increment()
```



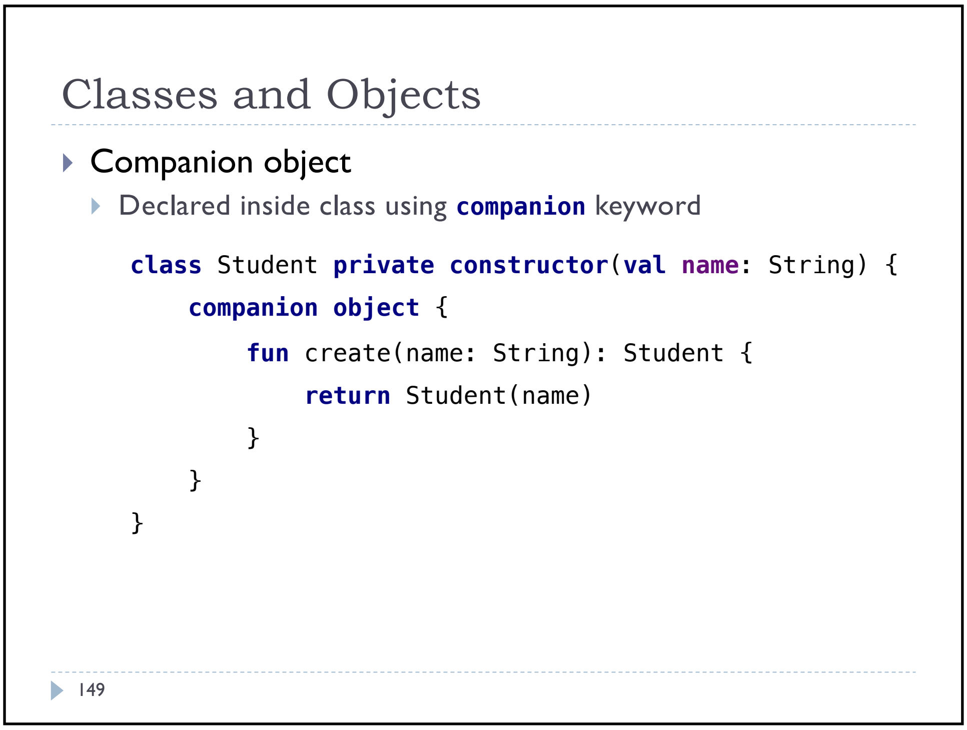
Classes and Objects

- Object Declaration
  - Objects can be defined as a sub-type

```
open class Shape {              object Rectangle : Shape() {
    var originX:Int = 0             var width:Int = 0
    var originY:Int = 0             var height:Int = 0
}                               }
```

  - Objects can implement Interfaces

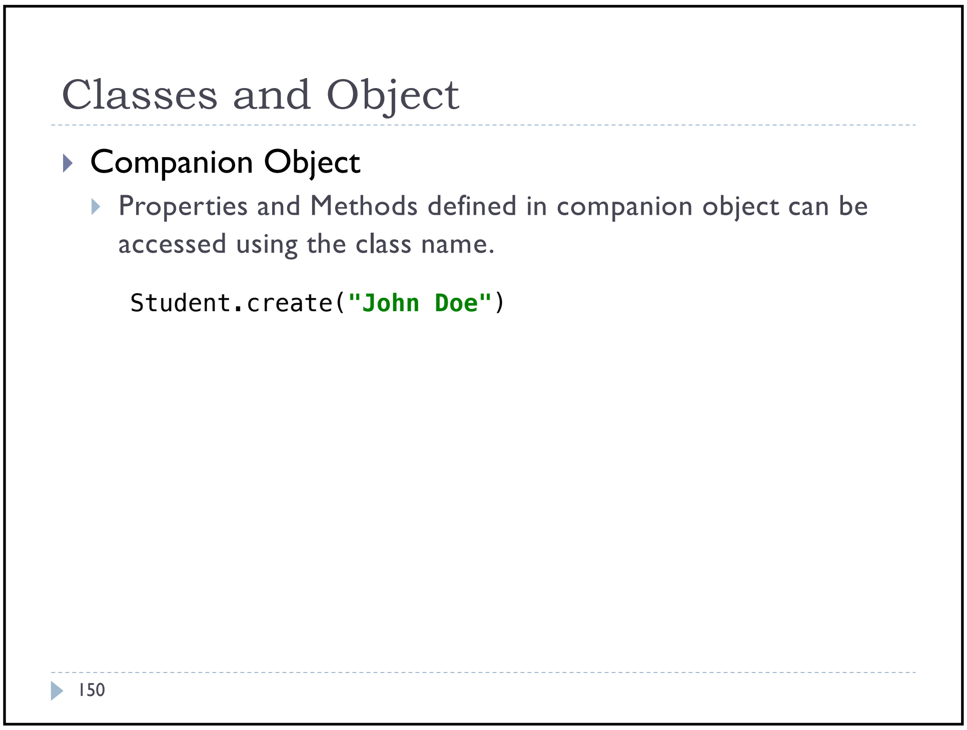
```
object DoSomething : Runnable {
    override fun run() { println("Working") }
}
```



Classes and Objects

- Companion object
  - Declared inside class using **companion** keyword

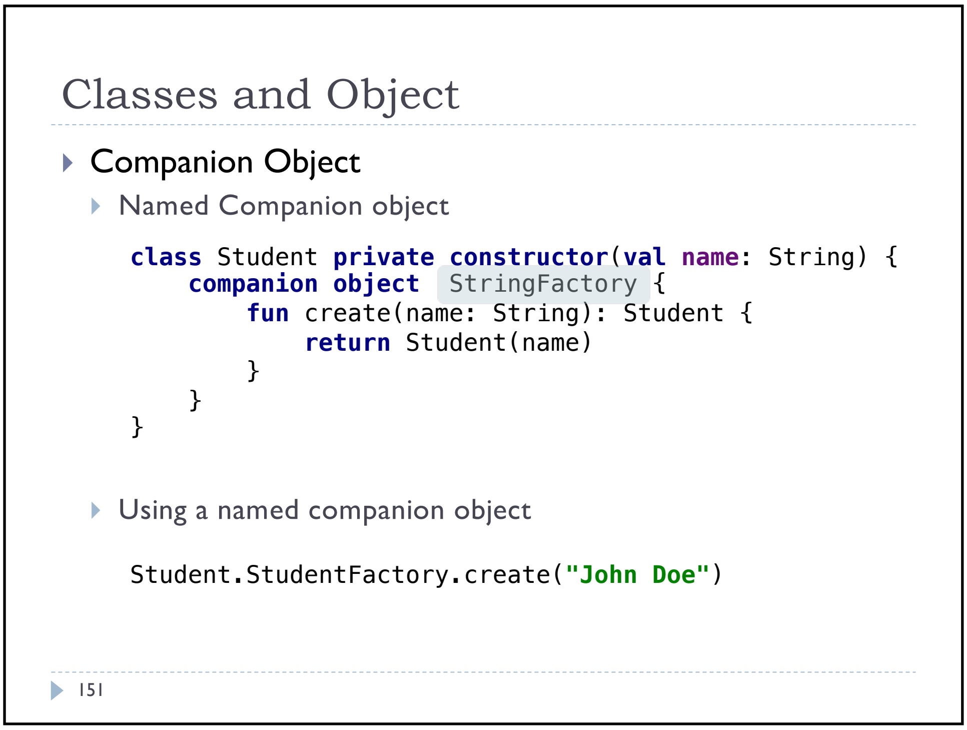
```
class Student private constructor(val name: String) {
    companion object {
        fun create(name: String): Student {
            return Student(name)
        }
    }
}
```



Classes and Object

- Companion Object
  - Properties and Methods defined in companion object can be accessed using the class name.

```
Student.create("John Doe")
```



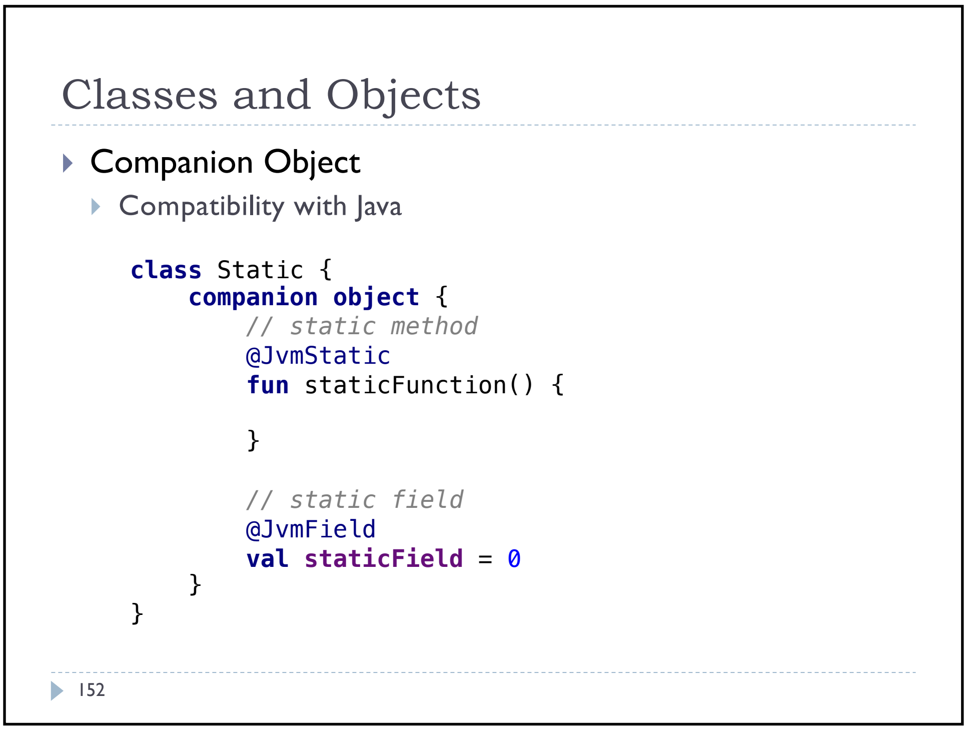
Classes and Object

- Companion Object
  - Named Companion object

```
class Student private constructor(val name: String) {
    companion object  StringFactory {
        fun create(name: String): Student {
            return Student(name)
        }
    }
}
```

  - Using a named companion object

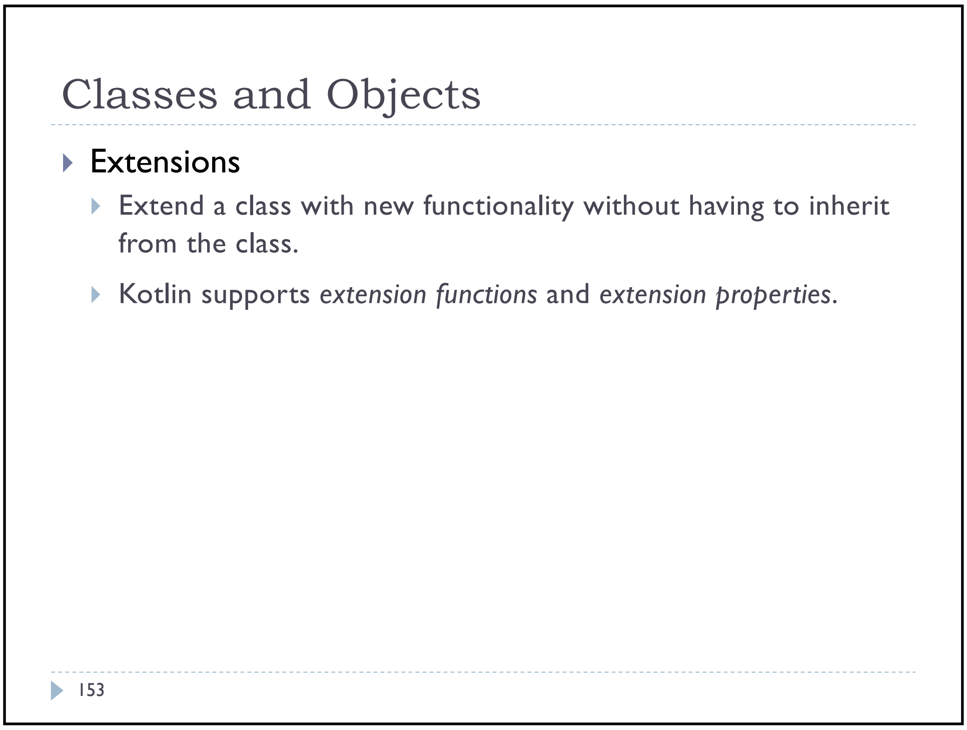
```
Student.StudentFactory.create("John Doe")
```



Classes and Objects

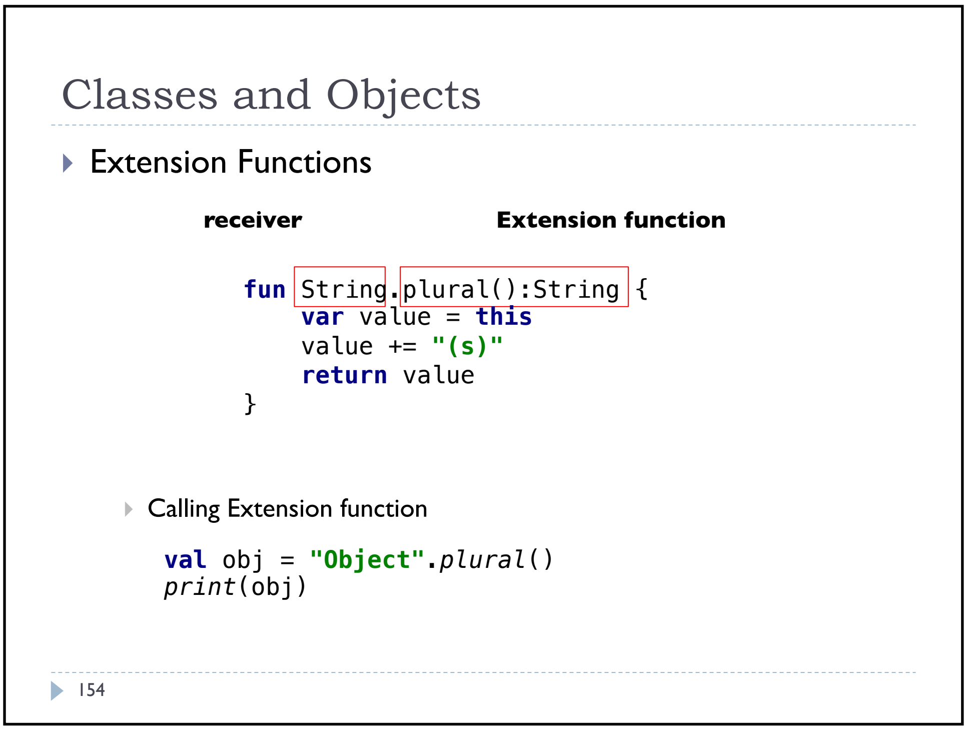
- Companion Object
  - Compatibility with Java

```
class Static {
    companion object {
        // static method
        @JvmStatic
        fun staticFunction() {

        }

        // static field
        @JvmField
        val staticField = 0
    }
}
```



Classes and Objects

- Extensions
  - Extend a class with new functionality without having to inherit from the class.
  - Kotlin supports *extension functions* and *extension properties*.



Classes and Objects

- Extension Functions

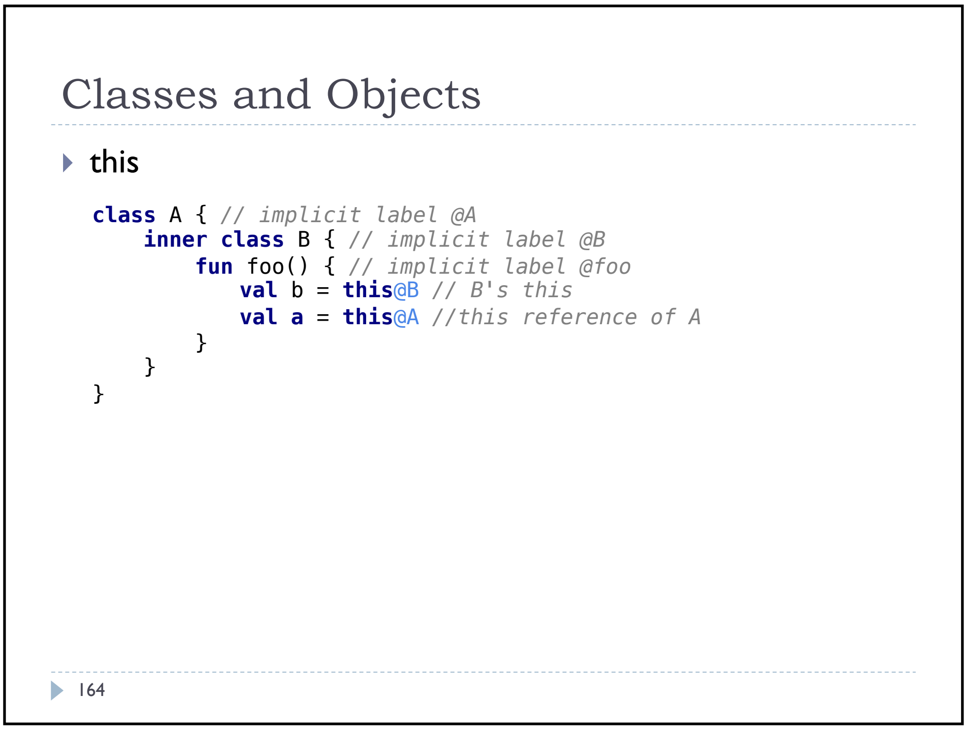
receiver **Extension function**

```
fun String.plural():String {
    var value = this
    value += "(s)"
    return value
}
```

- Calling Extension function

```
val obj = "Object".plural()
print(obj)
```

154

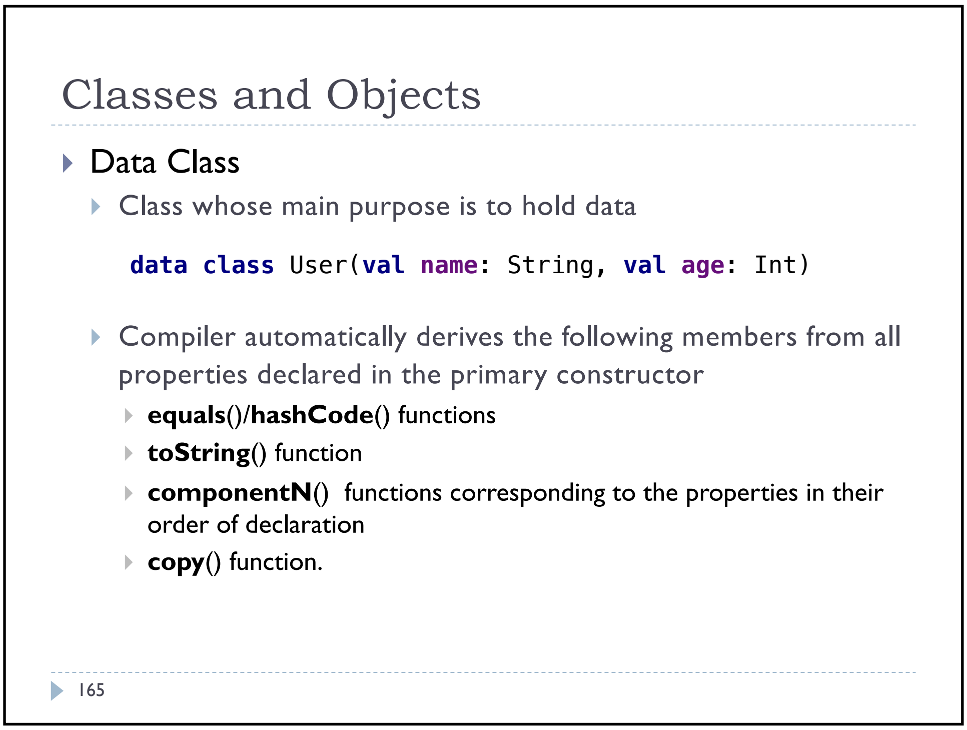


Classes and Objects

- this

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun foo() { // implicit label @foo
            val b = this@B // B's this
            val a = this@A //this reference of A
        }
    }
}
```

164

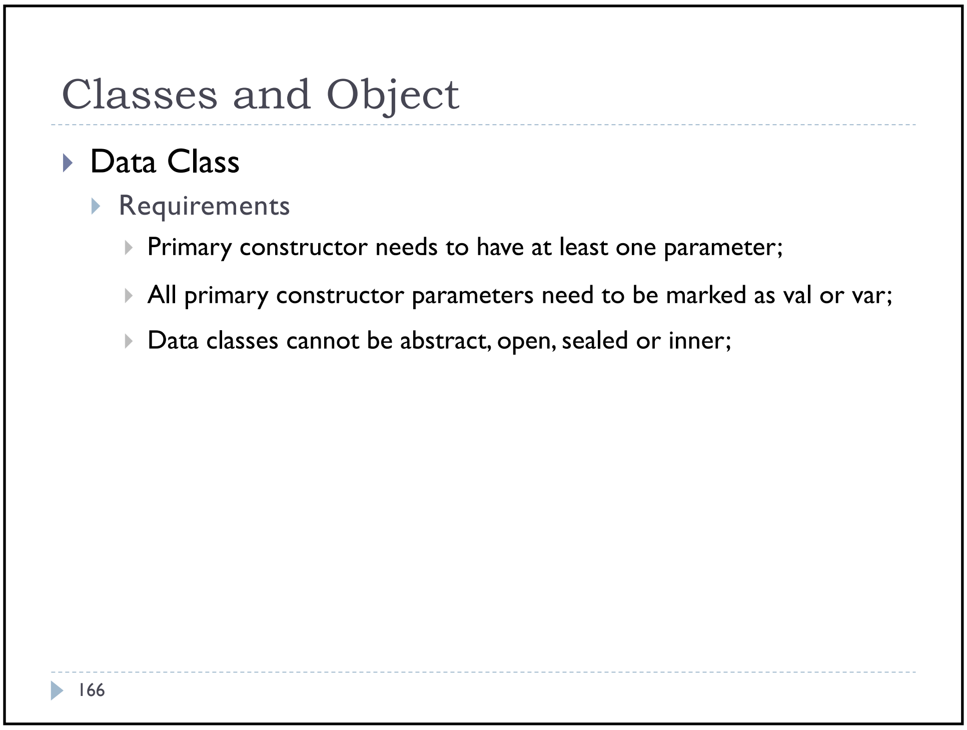


Classes and Objects

- Data Class
  - Class whose main purpose is to hold data

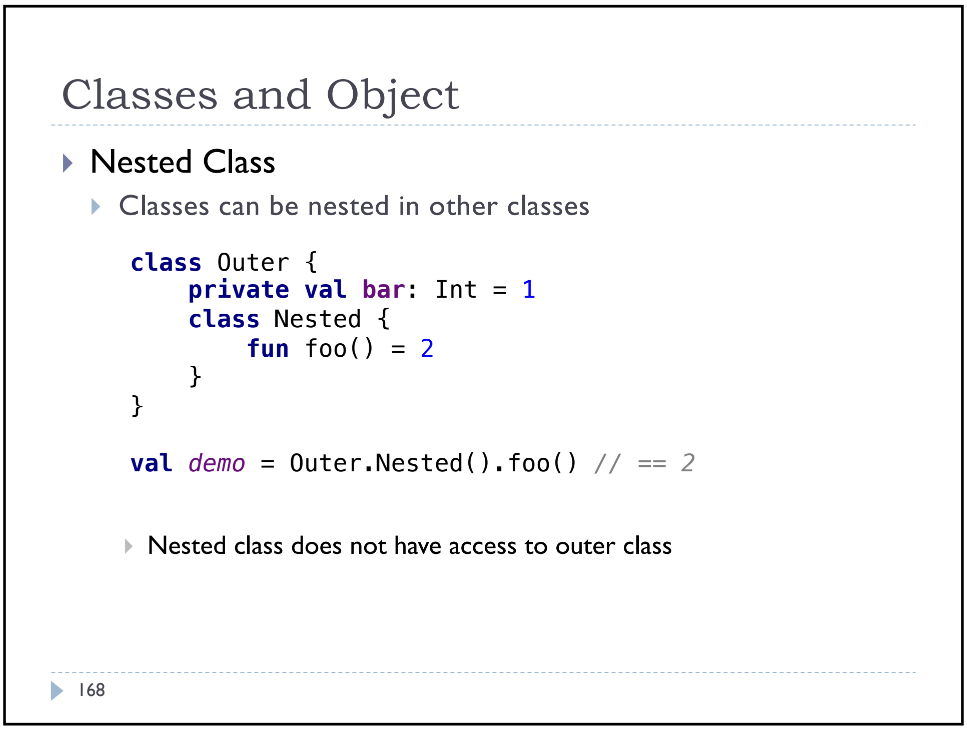
```
data class User(val name: String, val age: Int)
```

  - Compiler automatically derives the following members from all properties declared in the primary constructor
    - **equals**()/**hashCode**() functions
    - **toString**() function
    - **componentN**() functions corresponding to the properties in their order of declaration
    - **copy**() function.



Classes and Object

- Data Class
  - Requirements
    - Primary constructor needs to have at least one parameter;
    - All primary constructor parameters need to be marked as val or var;
    - Data classes cannot be abstract, open, sealed or inner;

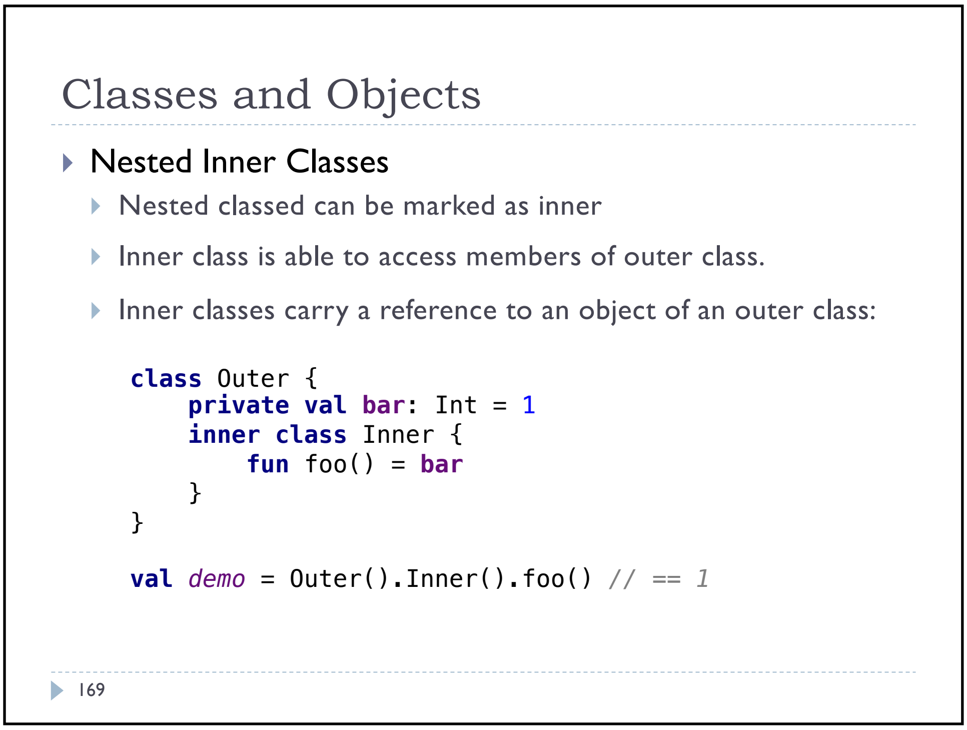


Classes and Object

- Nested Class
  - Classes can be nested in other classes

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

- Nested class does not have access to outer class



Classes and Objects

- Nested Inner Classes
  - Nested classed can be marked as inner
  - Inner class is able to access members of outer class.
  - Inner classes carry a reference to an object of an outer class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

Classes and Objects

- Nested Inner Classes
  - Accessing the superclass of the outer class
    - super keyword qualified with the outer class name: super@Outer

```
class Bar : Foo() {
    override fun f() { /* ... */ }
    override val x: Int get() = 0

    inner class Baz {
        fun g() {
            super@Bar.f()}
    }
}
```

Classes and Objects

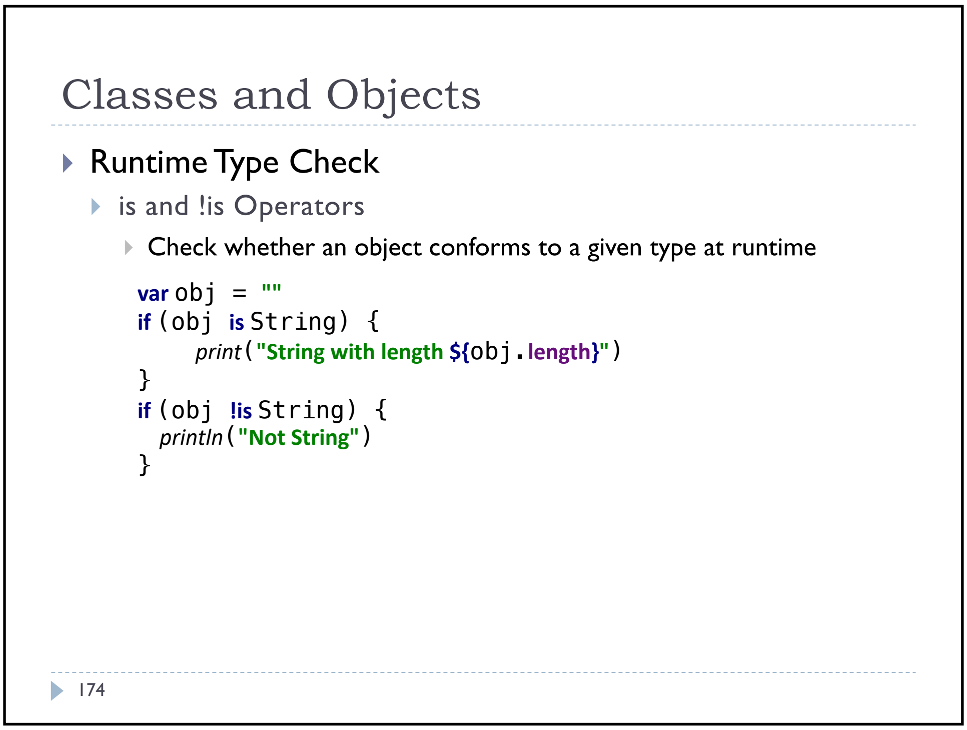
- Enum Classes
  - Basic usage of enum classes is implementing type-safe enum

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

    - Each enum constant is an object.
    - Enum constants are separated with commas.

  - Associating a value for each object

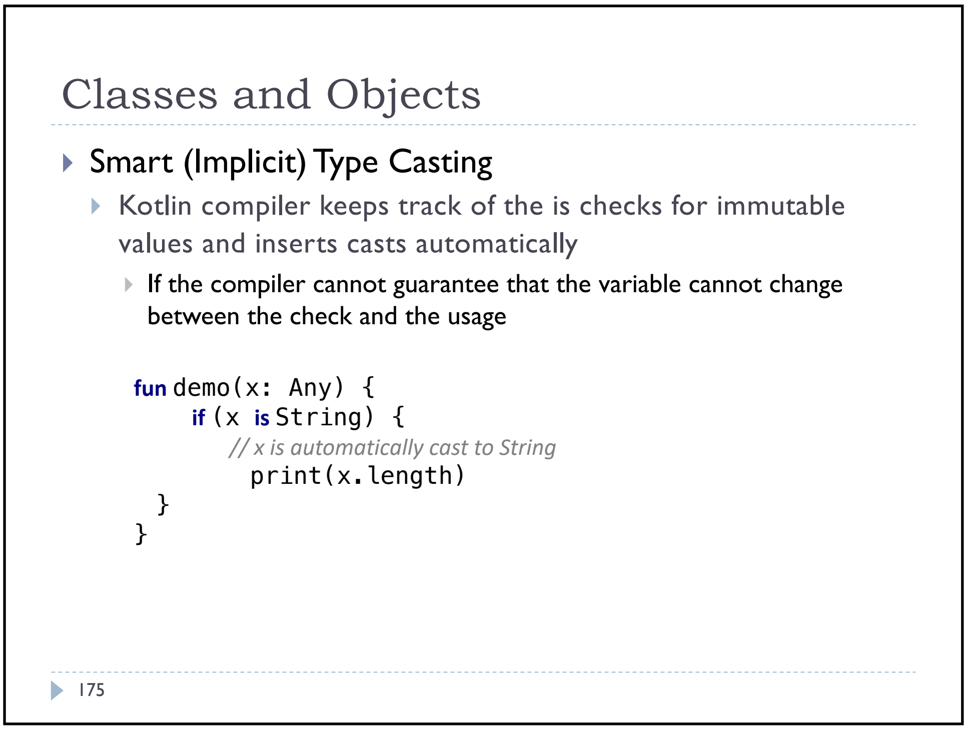
```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```



Classes and Objects

- Runtime Type Check
  - is and !is Operators
    - Check whether an object conforms to a given type at runtime

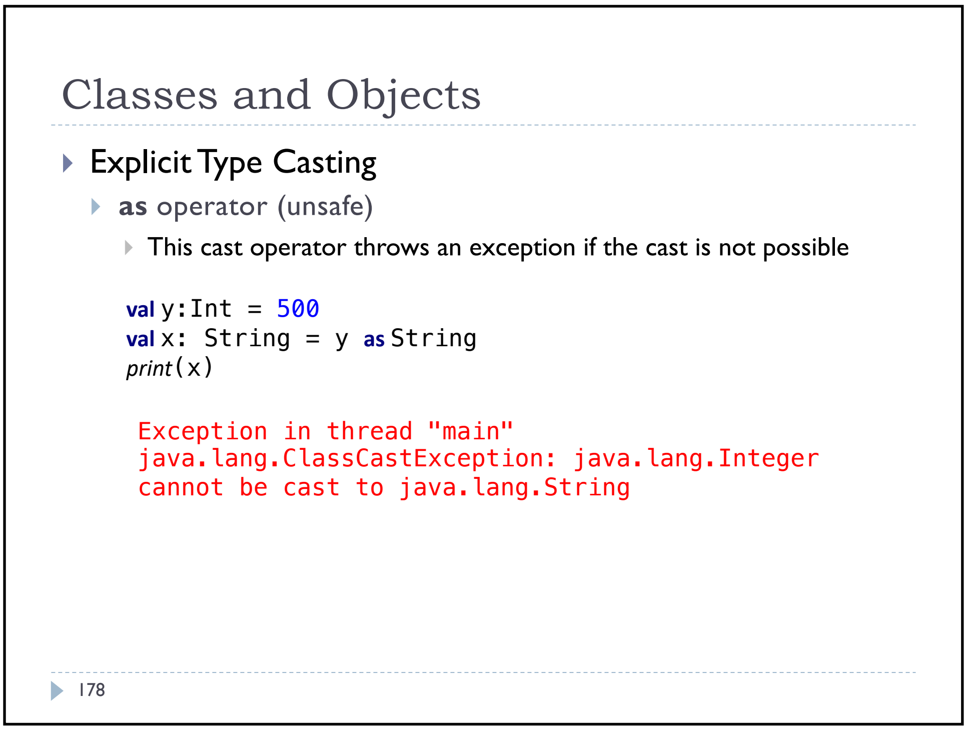
```
var obj = ""
if (obj is String) {
      print("String with length ${obj.length}")
}
if (obj !is String) {
   println("Not String")
}
```



Classes and Objects

- Smart (Implicit) Type Casting
  - Kotlin compiler keeps track of the is checks for immutable values and inserts casts automatically
    - If the compiler cannot guarantee that the variable cannot change between the check and the usage

```
fun demo(x: Any) {
      if (x is String) {
          // x is automatically cast to String
              print(x.length)
   }
}
```

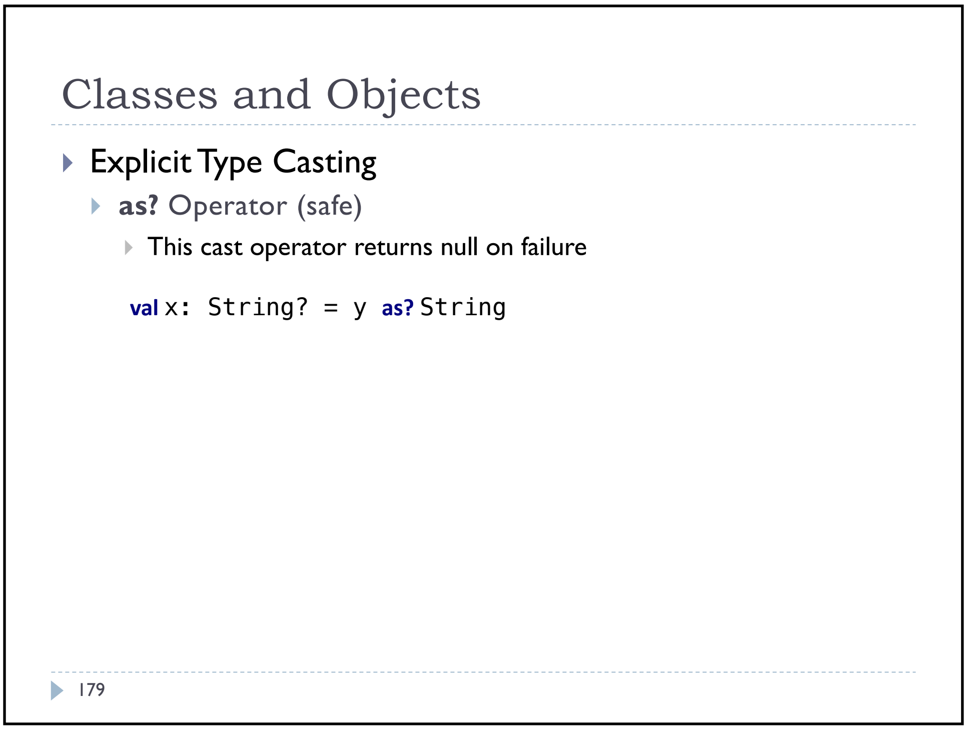


Classes and Objects

- Explicit Type Casting
  - **as** operator (unsafe)
    - This cast operator throws an exception if the cast is not possible

```
val y:Int = 500
val x: String = y as String
print(x)
```

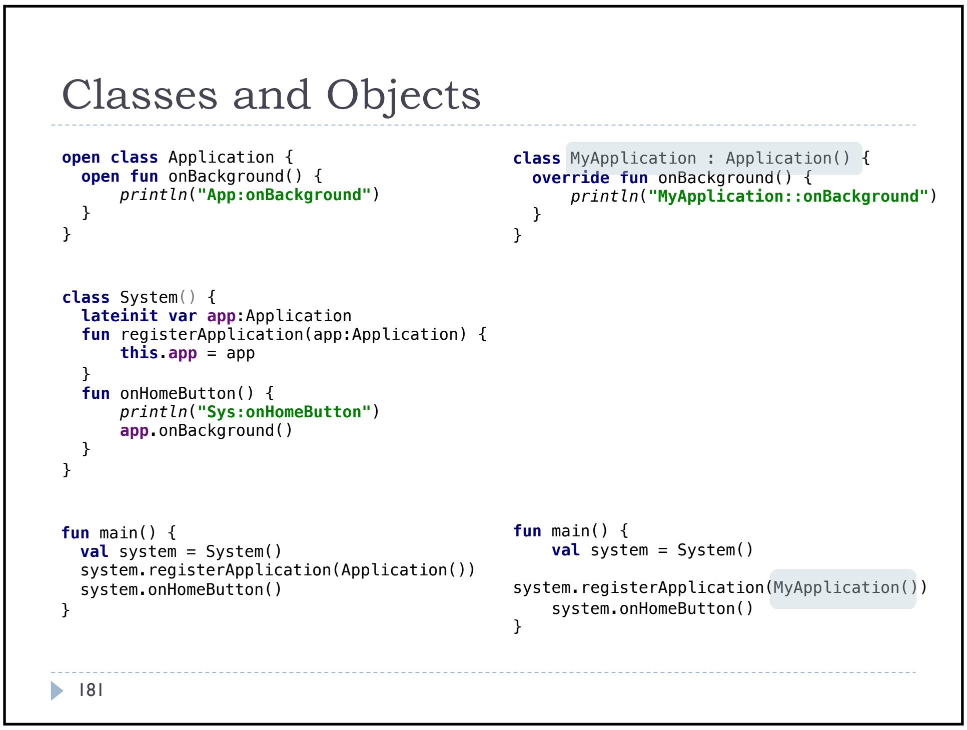
```
Exception in thread "main"
java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.String
```



Classes and Objects

- Explicit Type Casting
  - **as?** Operator (safe)
    - This cast operator returns null on failure

```
val x: String? = y as? String
```



Classes and Objects

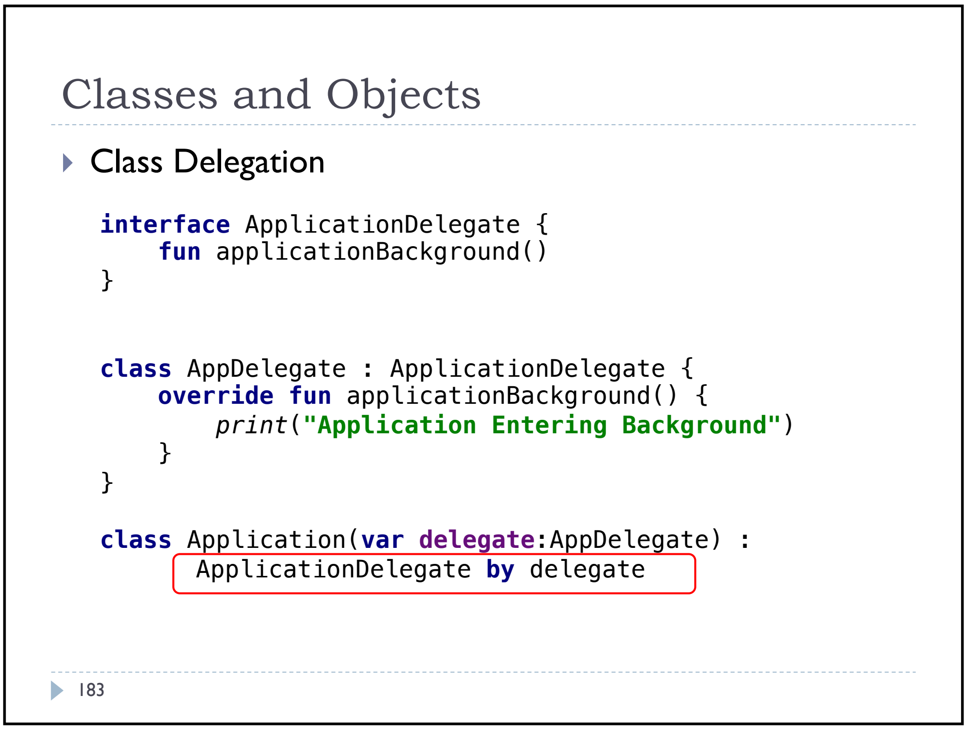
```
open class Application {
  open fun onBackground() {
      println("App:onBackground")
  }
}
```

```
class MyApplication : Application() {
  override fun onBackground() {
      println("MyApplication::onBackground")
  }
}
```

```
class System() {
  lateinit var app:Application
  fun registerApplication(app:Application) {
      this.app = app
  }
  fun onHomeButton() {
      println("Sys:onHomeButton")
      app.onBackground()
  }
}
```

```
fun main() {
  val system = System()
  system.registerApplication(Application())
  system.onHomeButton()
}
```

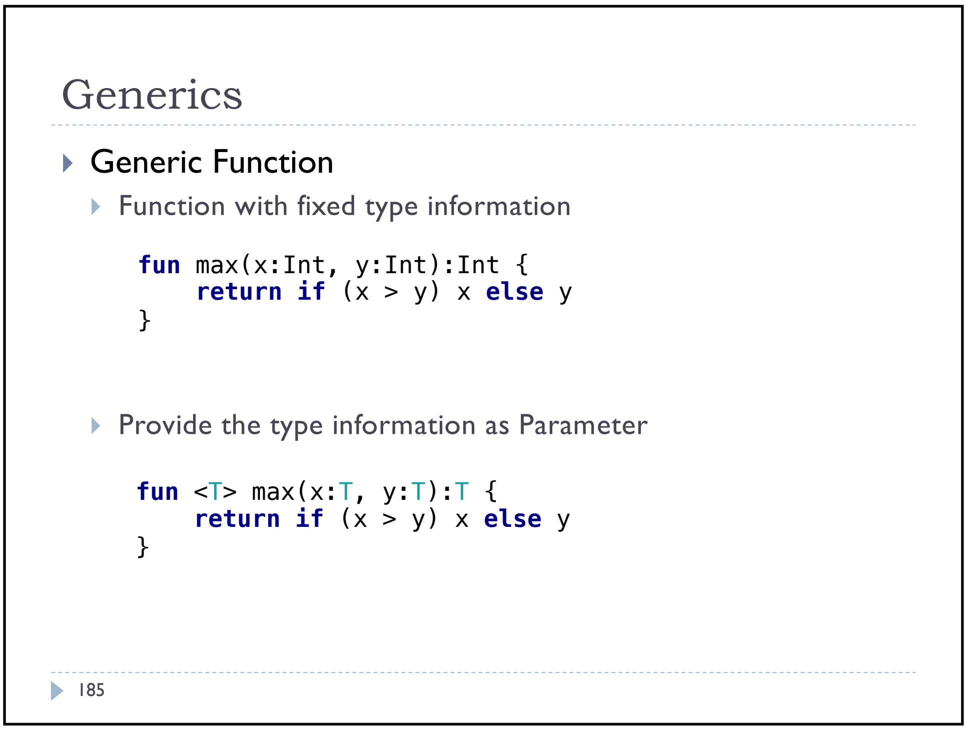
```
fun main() {
    val system = System()
system.registerApplication(MyApplication())
    system.onHomeButton()
}
```



Classes and Objects

- Class Delegation

```
interface ApplicationDelegate {
    fun applicationBackground()
}


class AppDelegate : ApplicationDelegate {
    override fun applicationBackground() {
        print("Application Entering Background")
    }
}

class Application(var delegate:AppDelegate) :
     ApplicationDelegate by delegate
```



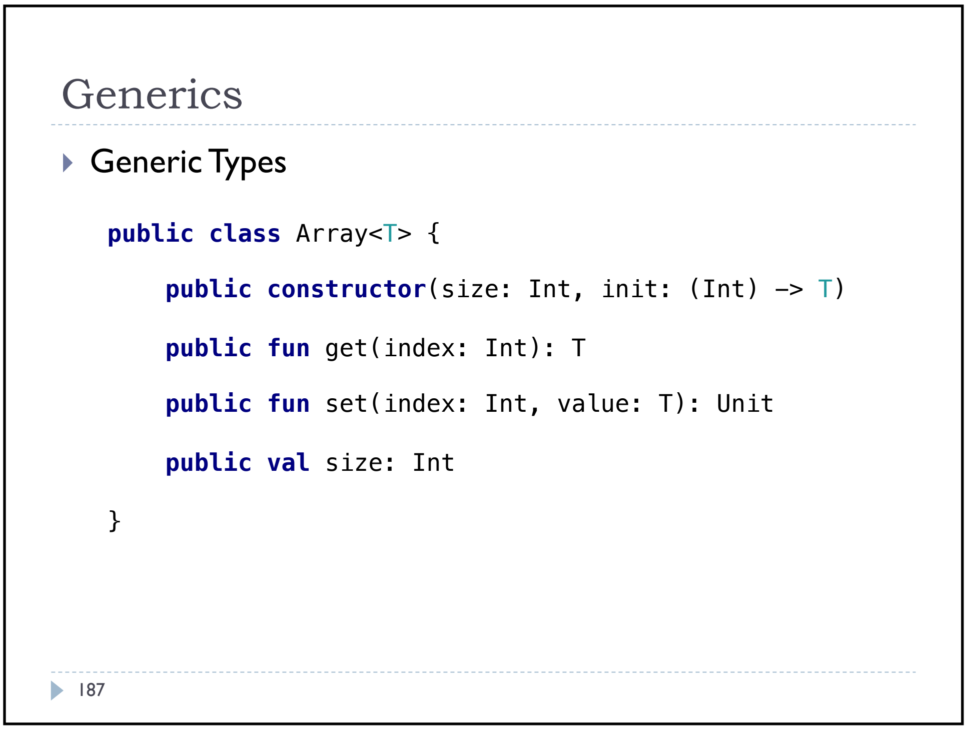
Generics

- Generic Function
  - Function with fixed type information

```
fun max(x:Int, y:Int):Int {
    return if (x > y) x else y
}
```

  - Provide the type information as Parameter

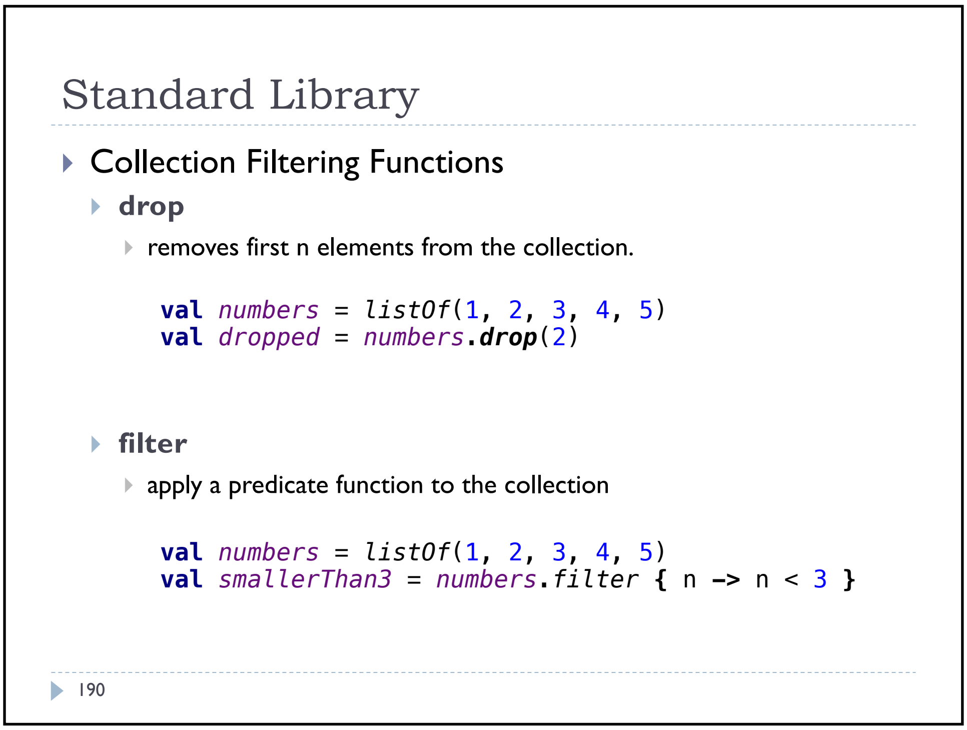
```
fun <T> max(x:T, y:T):T {
    return if (x > y) x else y
}
```



Generics

- Generic Types

```
public class Array<T> {

    public constructor(size: Int, init: (Int) -> T)

    public fun get(index: Int): T

    public fun set(index: Int, value: T): Unit

    public val size: Int

}
```



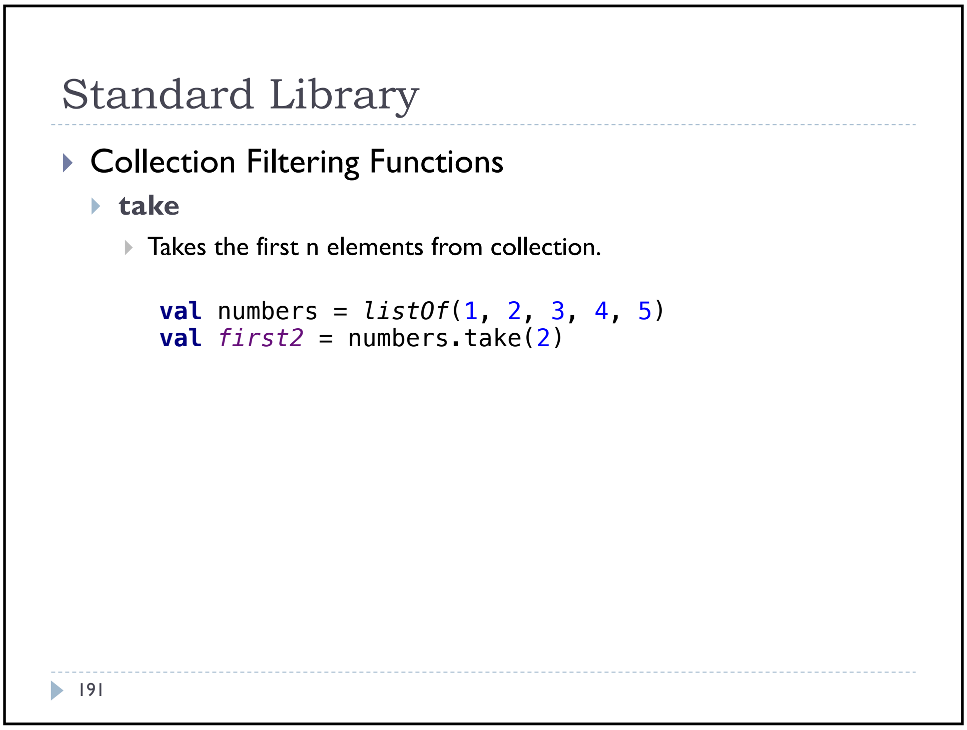
Standard Library

- Collection Filtering Functions
  - **drop**
    - removes first n elements from the collection.

```
val numbers = listOf(1, 2, 3, 4, 5)
val dropped = numbers.drop(2)
```

  - **filter**
    - apply a predicate function to the collection

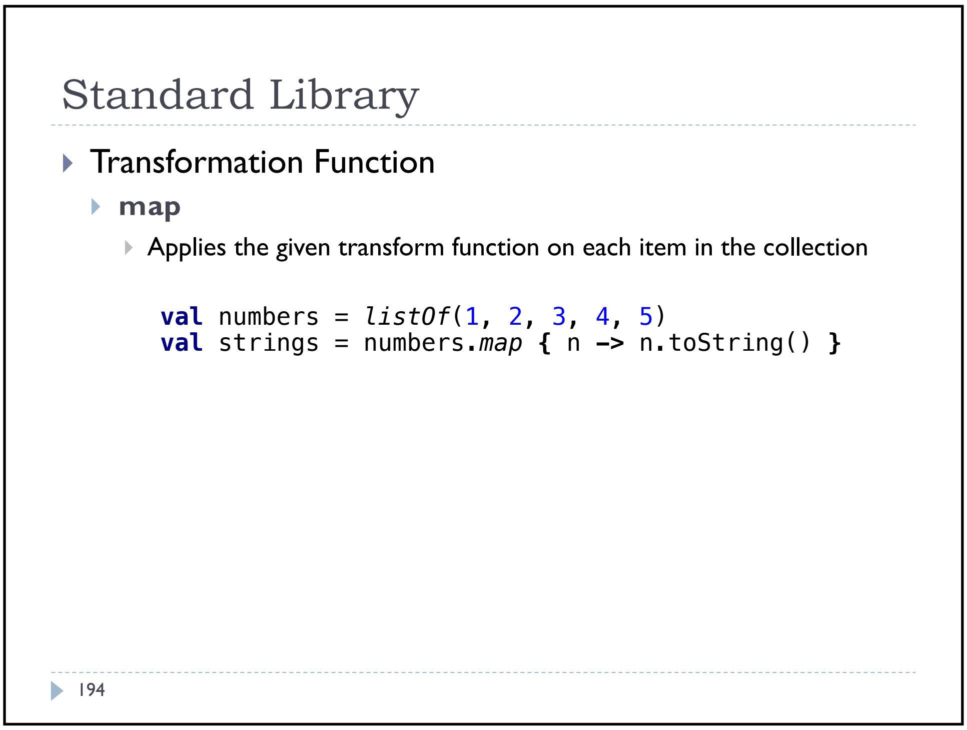
```
val numbers = listOf(1, 2, 3, 4, 5)
val smallerThan3 = numbers.filter { n -> n < 3 }
```



Standard Library

- Collection Filtering Functions
  - **take**
    - Takes the first n elements from collection.

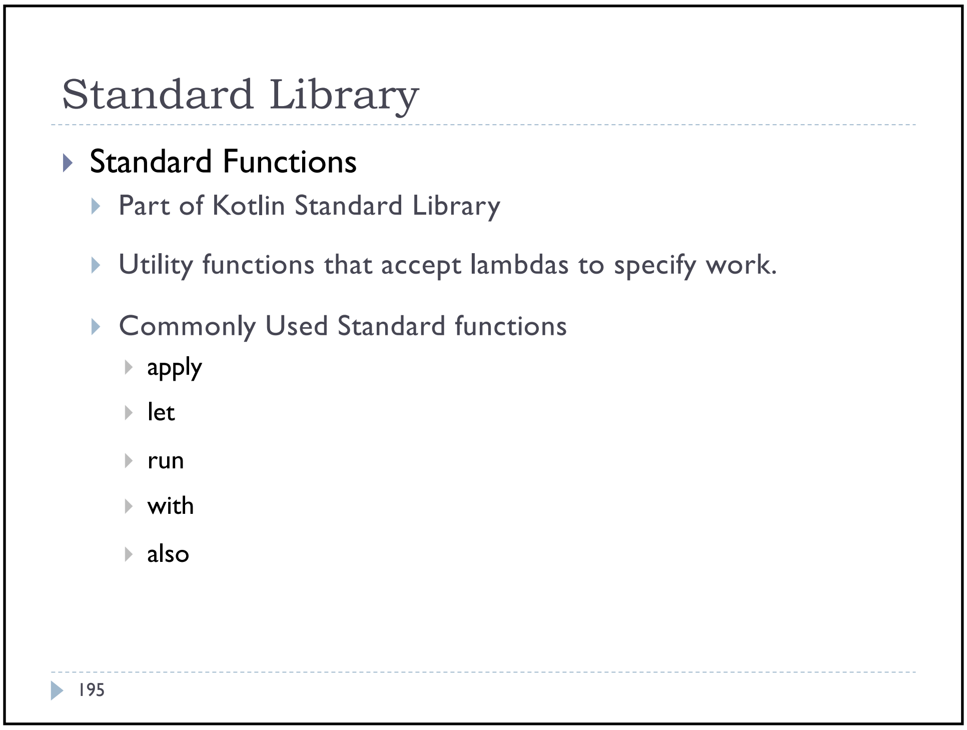
```
val numbers = listOf(1, 2, 3, 4, 5)
val first2 = numbers.take(2)
```



Standard Library

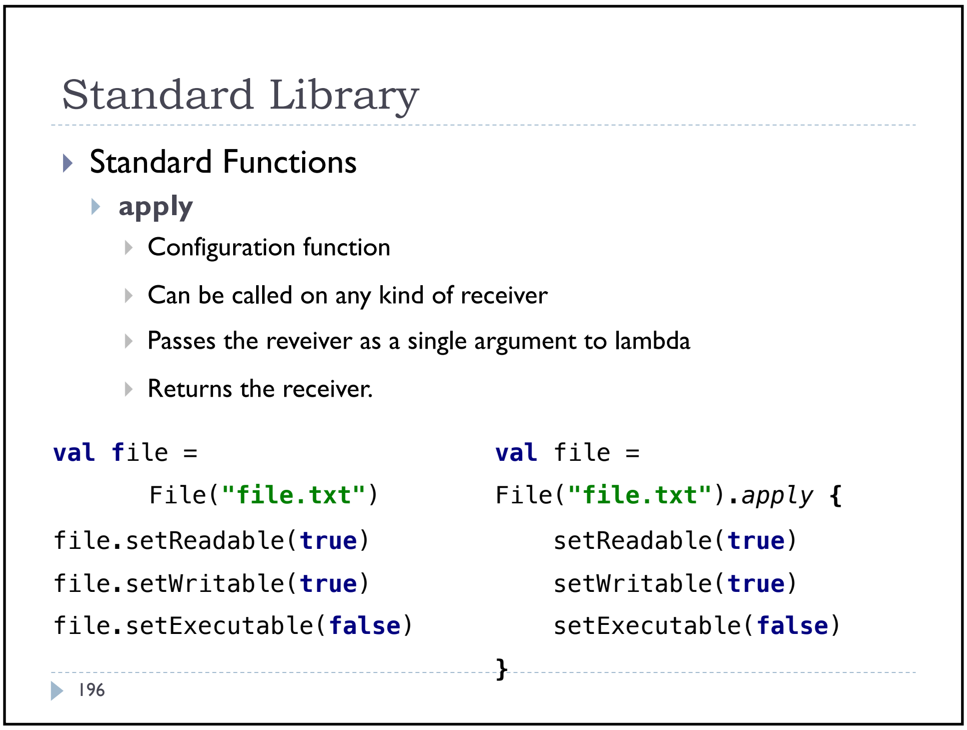
- Transformation Function
  - **map**
    - Applies the given transform function on each item in the collection

```
val numbers = listOf(1, 2, 3, 4, 5)
val strings = numbers.map { n -> n.toString() }
```



Standard Library

- Standard Functions
  - Part of Kotlin Standard Library
  - Utility functions that accept lambdas to specify work.
  - Commonly Used Standard functions
    - apply
    - let
    - run
    - with
    - also

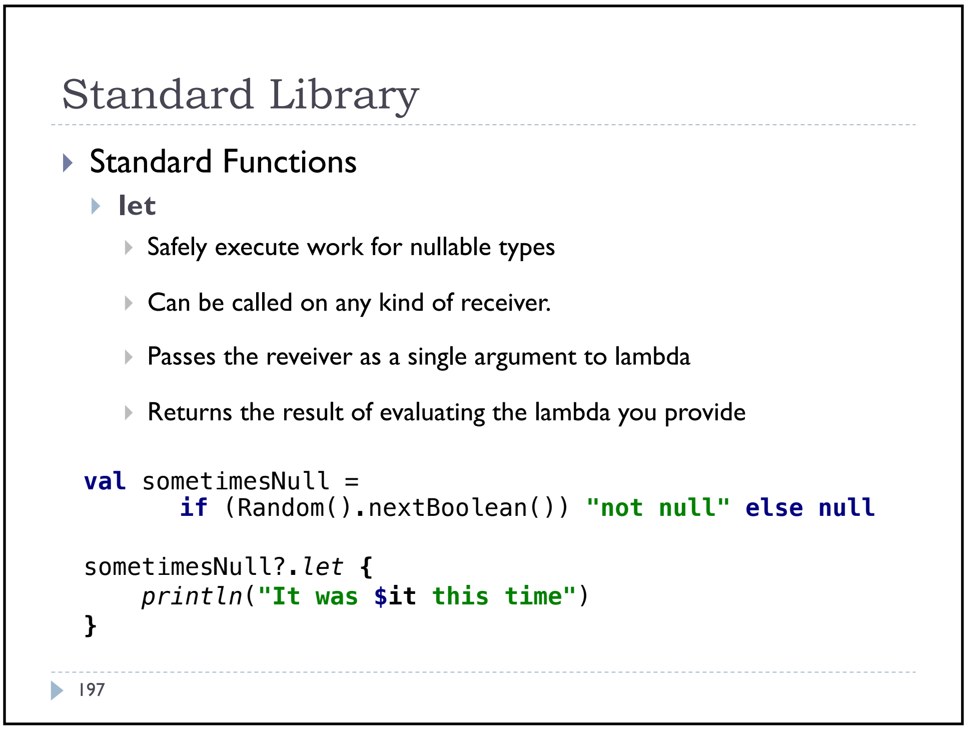


Standard Library

- Standard Functions
  - **apply**
    - Configuration function
    - Can be called on any kind of receiver
    - Passes the reveiver as a single argument to lambda
    - Returns the receiver.

```
val file =
        File("file.txt")
file.setReadable(true)
file.setWritable(true)
file.setExecutable(false)
```

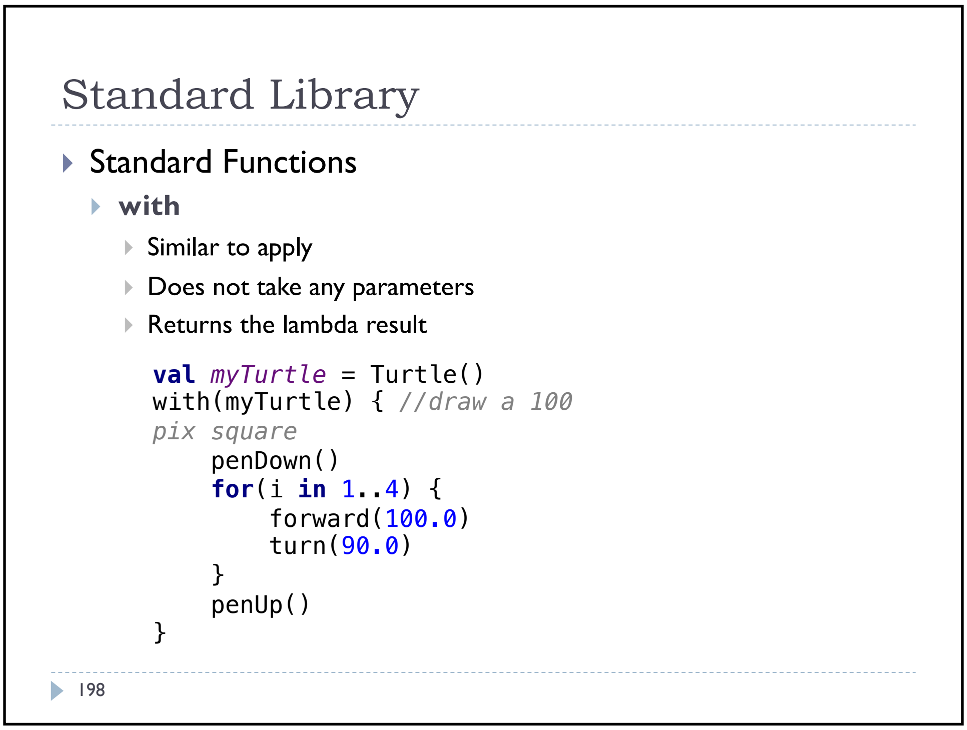
```
val file =
File("file.txt").apply {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```



Standard Library

- Standard Functions
  - **let**
    - Safely execute work for nullable types
    - Can be called on any kind of receiver.
    - Passes the reveiver as a single argument to lambda
    - Returns the result of evaluating the lambda you provide

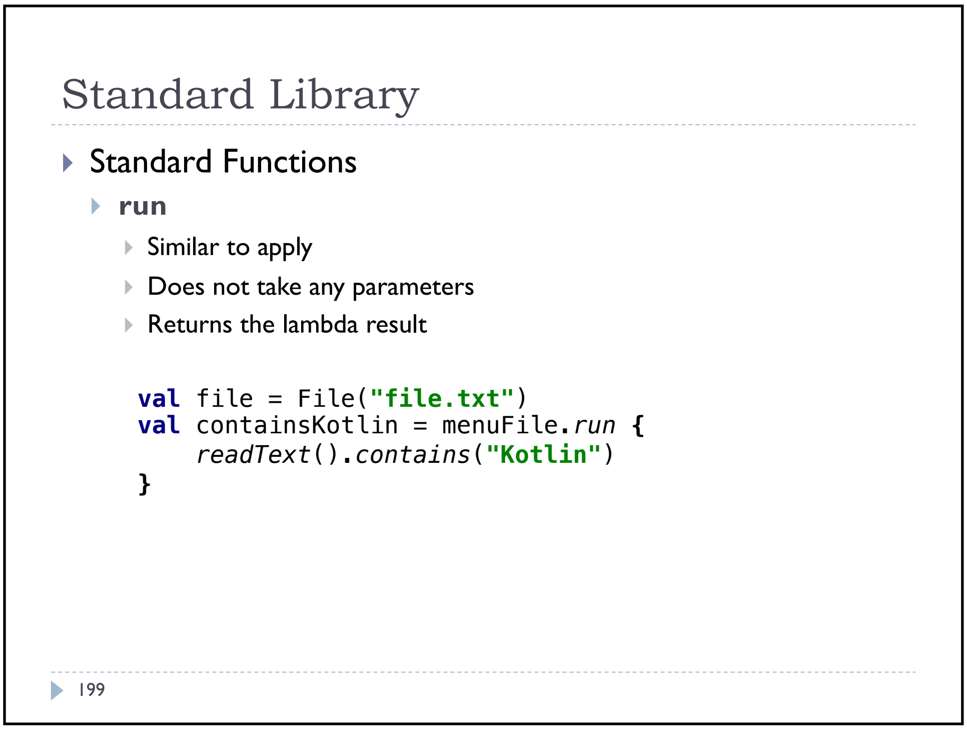
```
val sometimesNull =
        if (Random().nextBoolean()) "not null" else null

sometimesNull?.let {
    println("It was $it this time")
}
```



Standard Library

- Standard Functions
  - **with**
    - Similar to apply
    - Does not take any parameters
    - Returns the lambda result

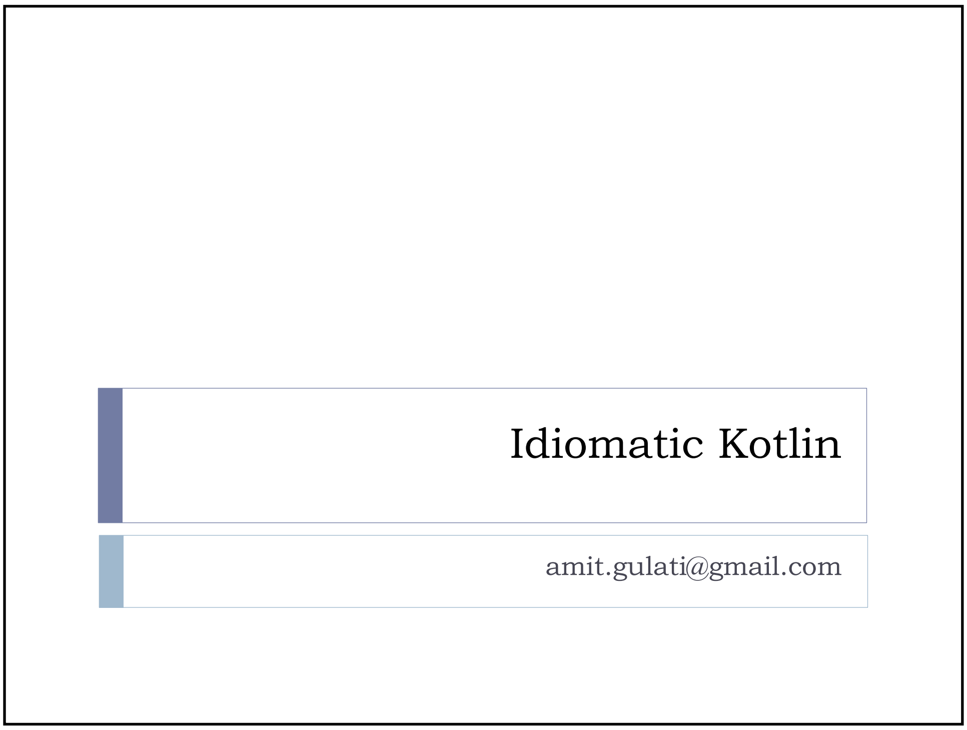
```
val myTurtle = Turtle()
with(myTurtle) { //draw a 100
pix square
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```



Standard Library

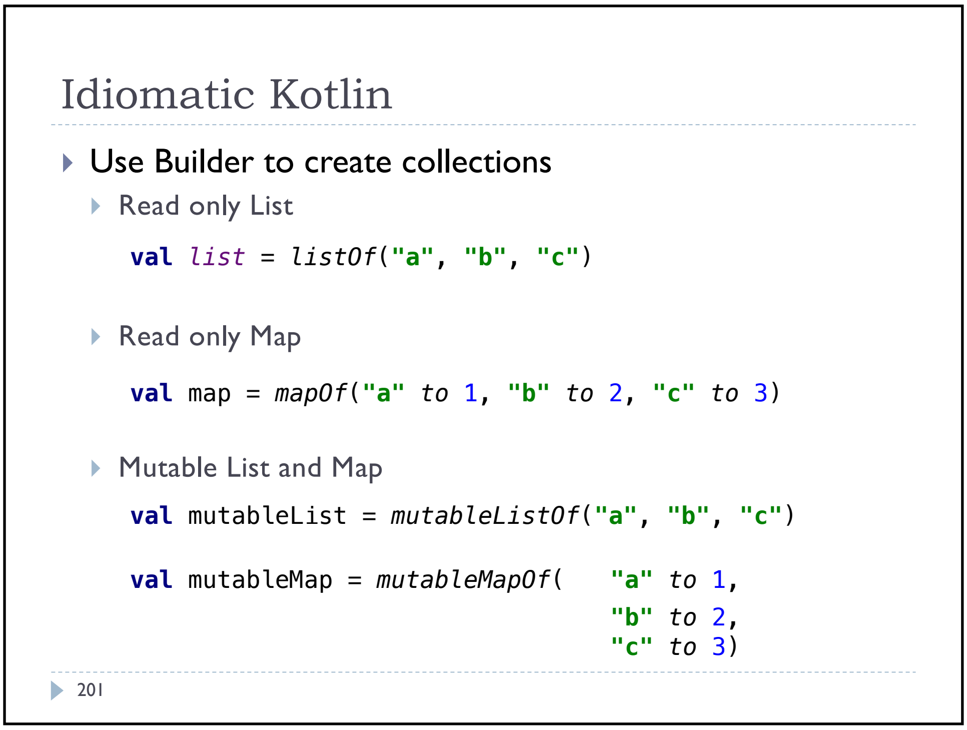
- Standard Functions
  - **run**
    - Similar to apply
    - Does not take any parameters
    - Returns the lambda result

```
val file = File("file.txt")
val containsKotlin = menuFile.run {
    readText().contains("Kotlin")
}
```



Idiomatic Kotlin

amit.gulati@gmail.com



Idiomatic Kotlin

- Use Builder to create collections
  - Read only List

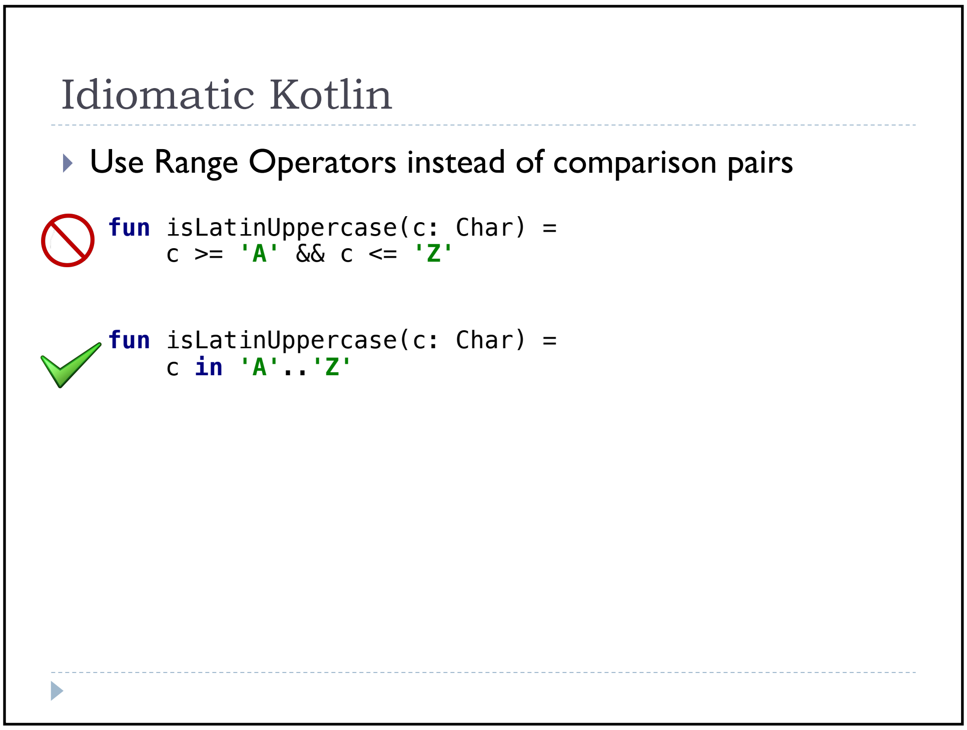
```
val list = listOf("a", "b", "c")
```

  - Read only Map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

  - Mutable List and Map

```
val mutableList = mutableListOf("a", "b", "c")

val mutableMap = mutableMapOf(   "a" to 1,
                                 "b" to 2,
                                 "c" to 3)
```



Idiomatic Kotlin

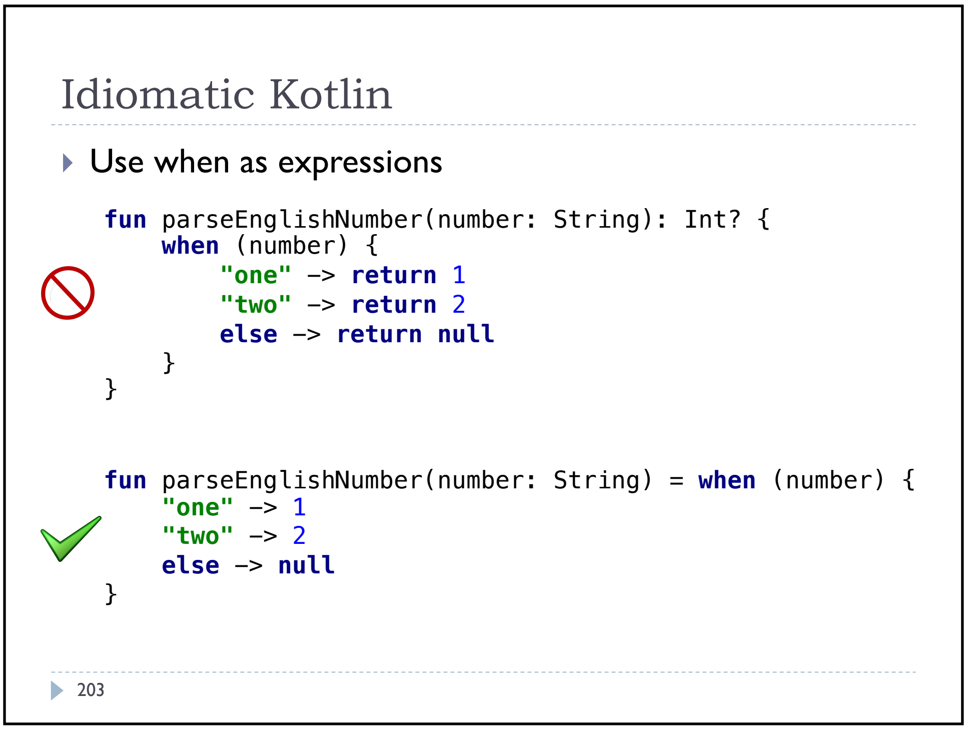
- Use Range Operators instead of comparison pairs

🚫

```
fun isLatinUppercase(c: Char) =
    c >= 'A' && c <= 'Z'
```

✓

```
fun isLatinUppercase(c: Char) =
    c in 'A'..'Z'
```



Idiomatic Kotlin

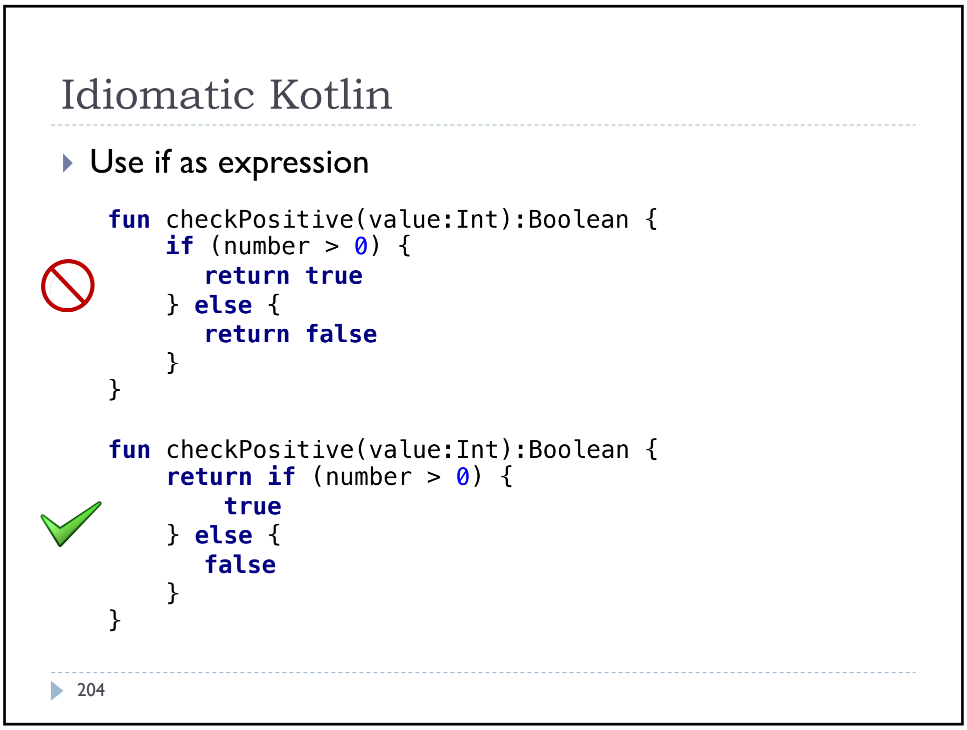
- Use when as expressions

🚫

```
fun parseEnglishNumber(number: String): Int? {
    when (number) {
        "one" -> return 1
        "two" -> return 2
        else -> return null
    }
}
```

✓

```
fun parseEnglishNumber(number: String) = when (number) {
    "one" -> 1
    "two" -> 2
    else -> null
}
```



Idiomatic Kotlin

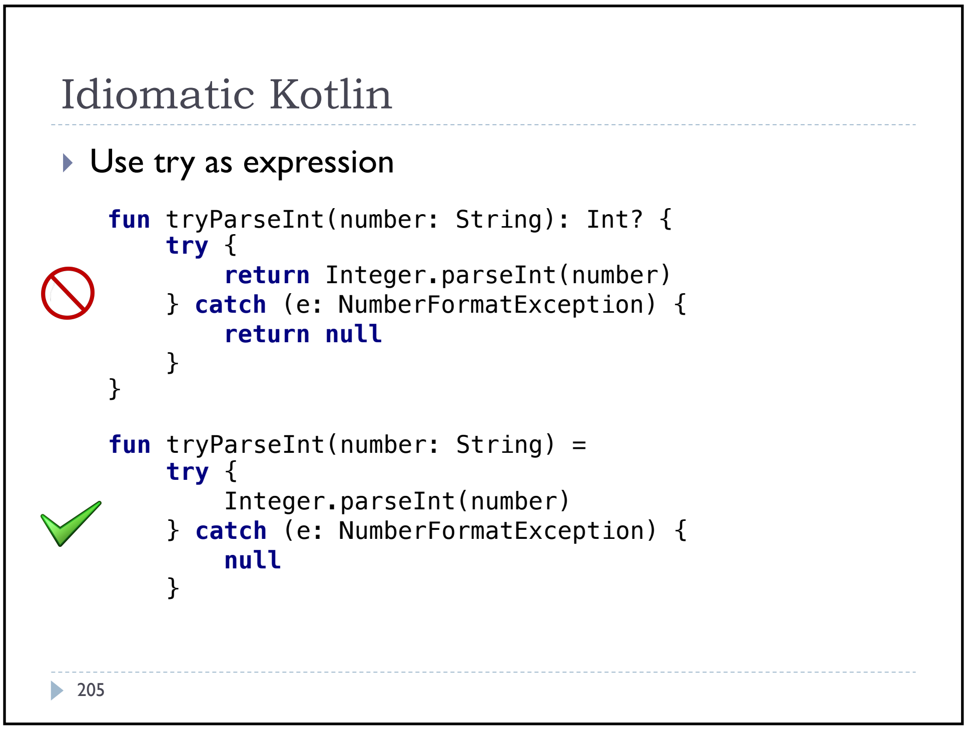
- Use if as expression

🚫

```
fun checkPositive(value:Int):Boolean {
    if (number > 0) {
       return true
    } else {
       return false
    }
}
```

✓

```
fun checkPositive(value:Int):Boolean {
    return if (number > 0) {
        true
    } else {
       false
    }
}
```



Idiomatic Kotlin

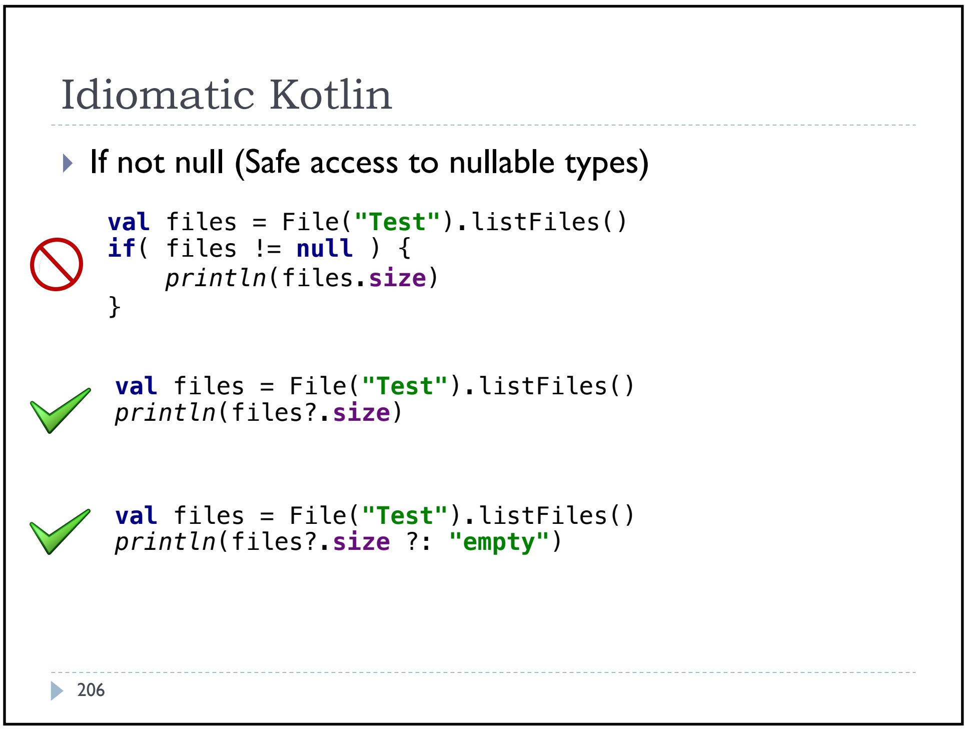
- Use try as expression

🚫

```
fun tryParseInt(number: String): Int? {
    try {
        return Integer.parseInt(number)
    } catch (e: NumberFormatException) {
        return null
    }
}
```

✓

```
fun tryParseInt(number: String) =
    try {
        Integer.parseInt(number)
    } catch (e: NumberFormatException) {
        null
    }
```



Idiomatic Kotlin

- If not null (Safe access to nullable types)

⊘

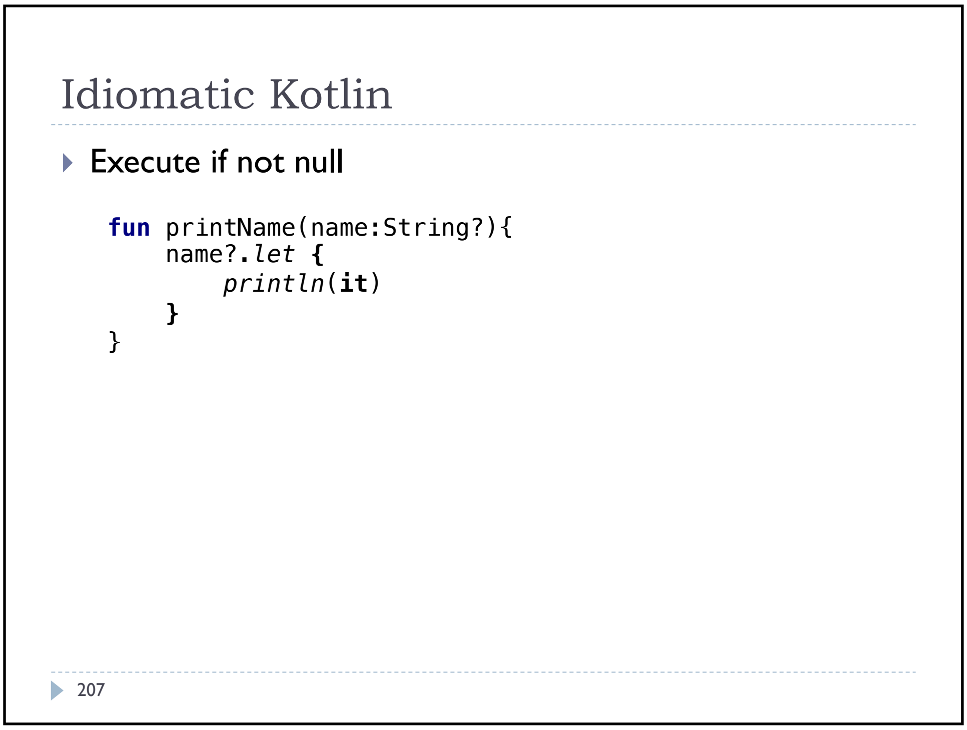
```
val files = File("Test").listFiles()
if( files != null ) {
    println(files.size)
}
```

✓

```
val files = File("Test").listFiles()
println(files?.size)
```

✓

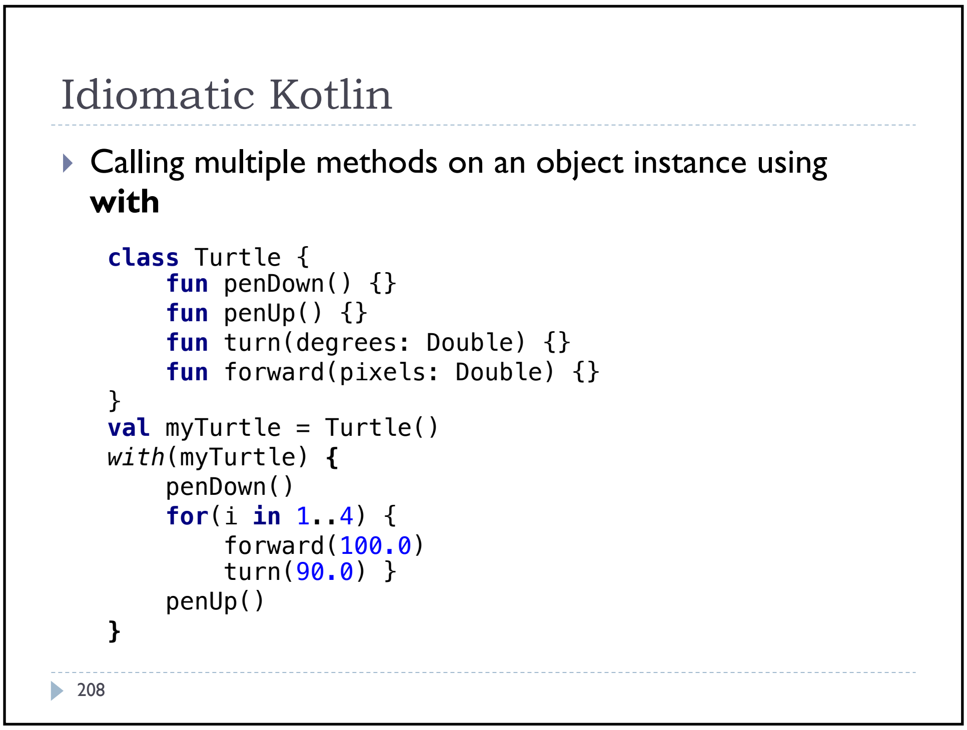
```
val files = File("Test").listFiles()
println(files?.size ?: "empty")
```



Idiomatic Kotlin

- Execute if not null

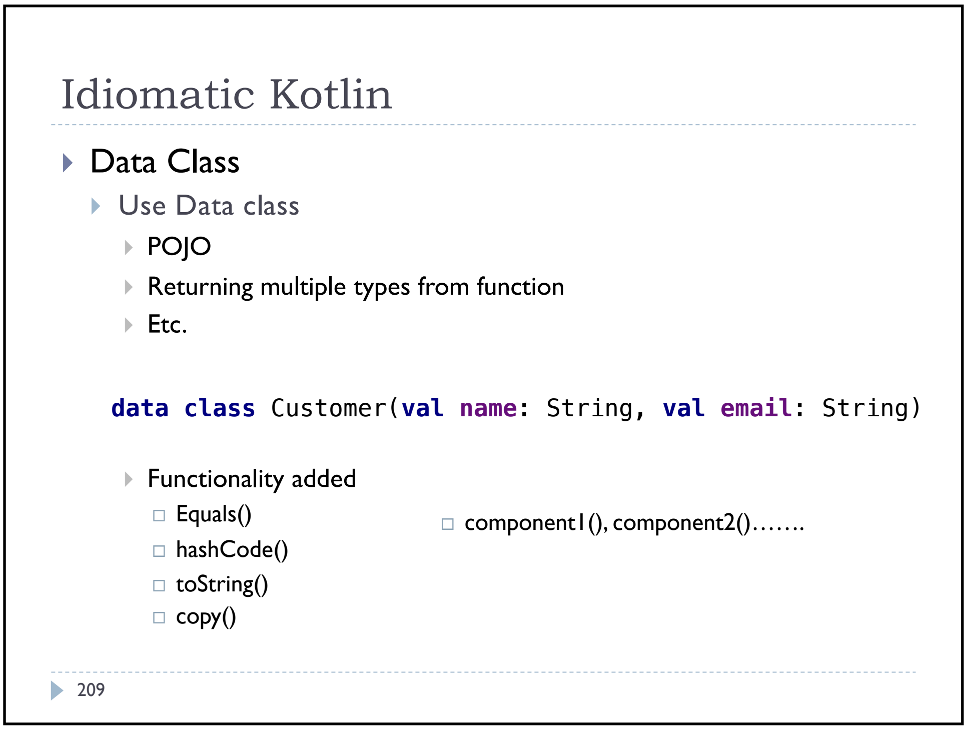
```
fun printName(name:String?){
    name?.let {
        println(it)
    }
}
```



Idiomatic Kotlin

- Calling multiple methods on an object instance using **with**

```
class Turtle {
    fun penDown() {}
    fun penUp() {}
    fun turn(degrees: Double) {}
    fun forward(pixels: Double) {}
}
val myTurtle = Turtle()
with(myTurtle) {
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0) }
    penUp()
}
```

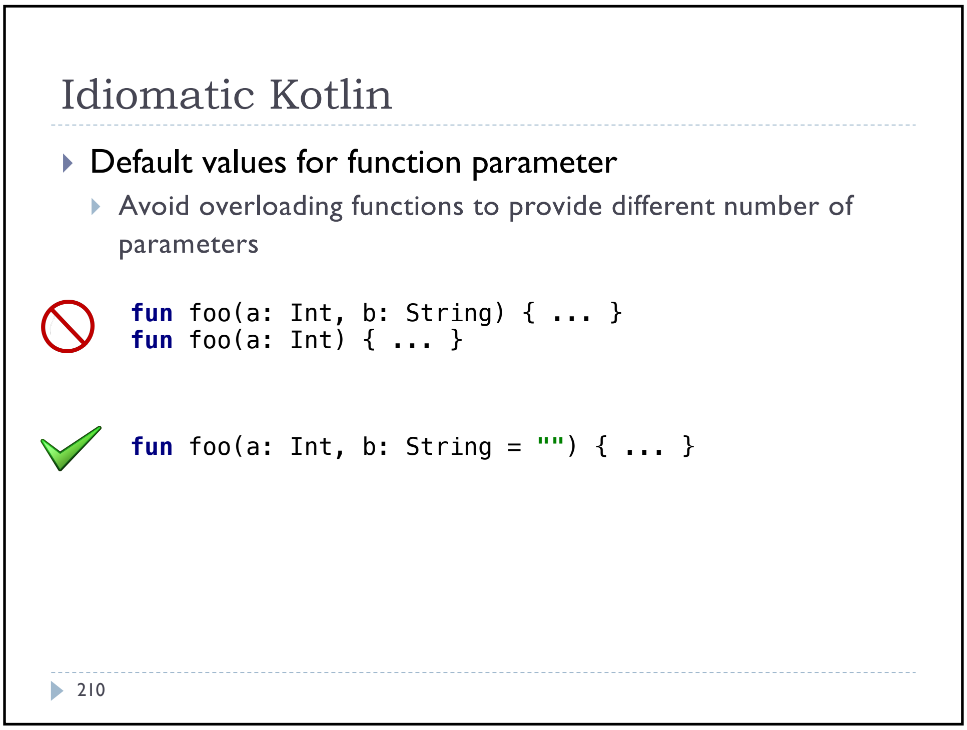


Idiomatic Kotlin

- Data Class
  - Use Data class
    - POJO
    - Returning multiple types from function
    - Etc.

```
data class Customer(val name: String, val email: String)
```

- Functionality added
  - □ Equals()
  - □ hashCode()
  - □ toString()
  - □ copy()
  - □ component1(), component2()…….



Idiomatic Kotlin

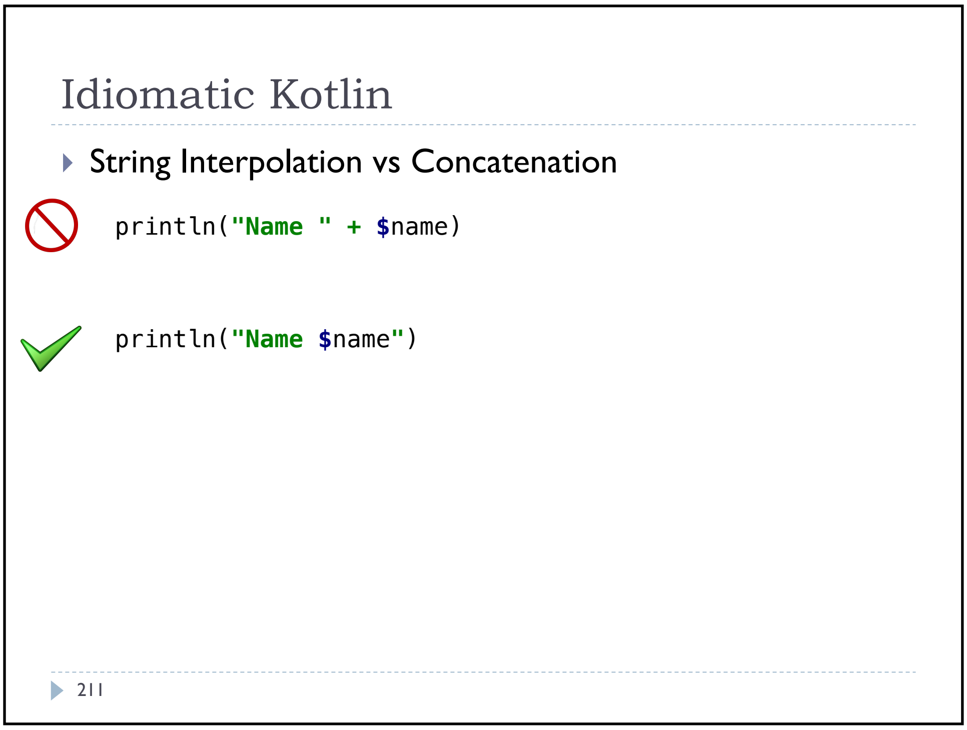
- Default values for function parameter
  - Avoid overloading functions to provide different number of parameters

🚫

```
fun foo(a: Int, b: String) { ... }
fun foo(a: Int) { ... }
```

✓

```
fun foo(a: Int, b: String = "") { ... }
```



Idiomatic Kotlin

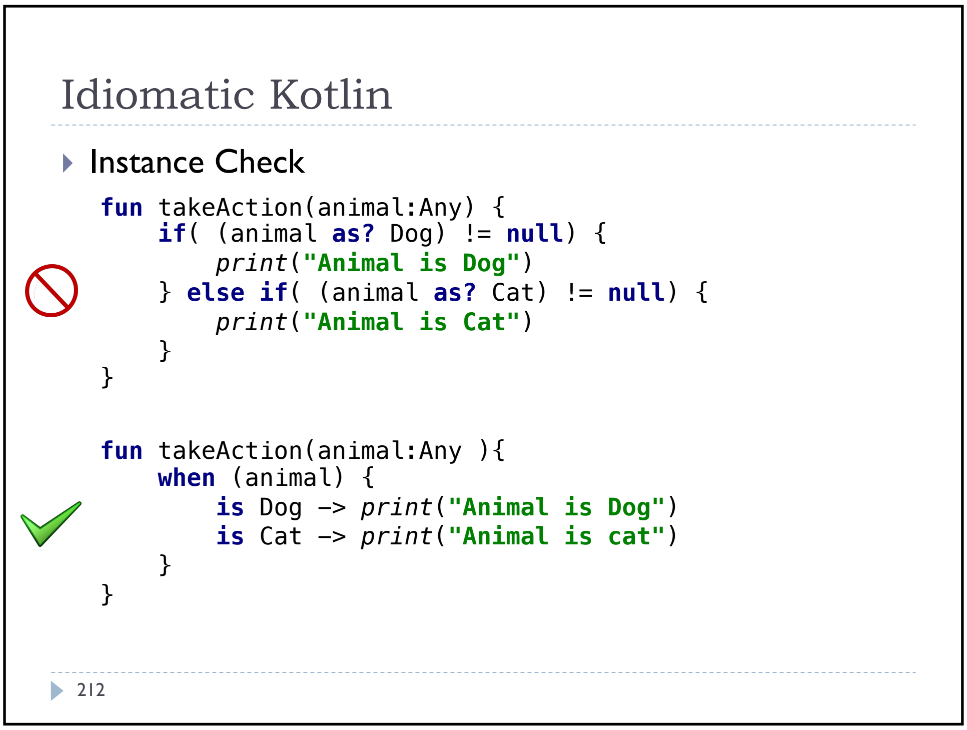
- String Interpolation vs Concatenation

🚫

```
println("Name " + $name)
```

✓

```
println("Name $name")
```



Idiomatic Kotlin

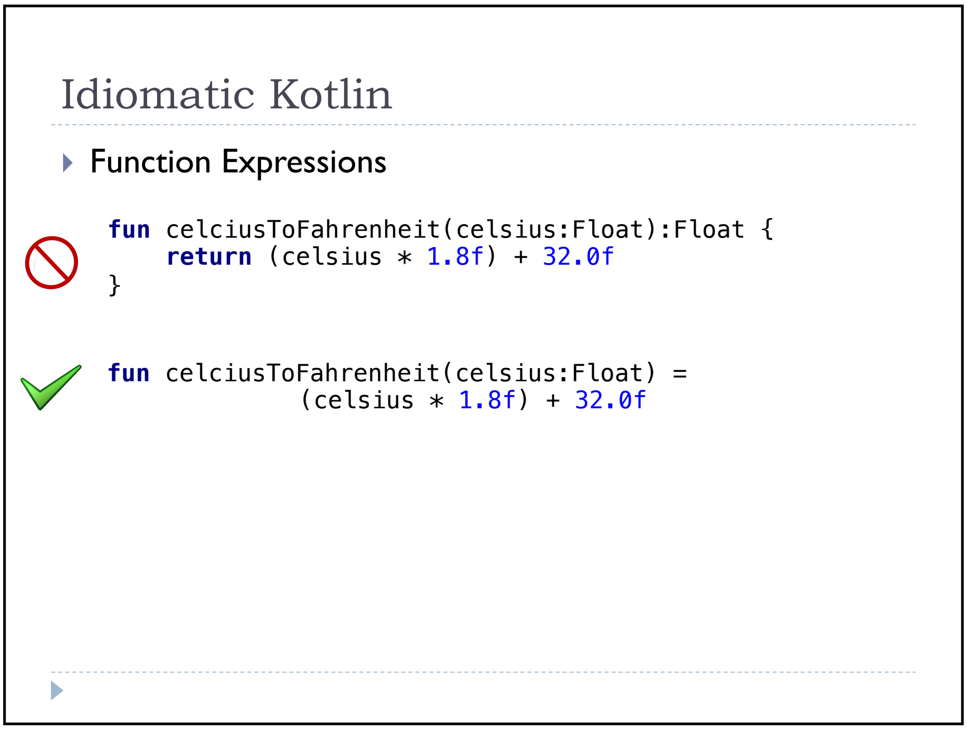
- Instance Check

🚫

```
fun takeAction(animal:Any) {
    if( (animal as? Dog) != null) {
        print("Animal is Dog")
    } else if( (animal as? Cat) != null) {
        print("Animal is Cat")
    }
}
```

✓

```
fun takeAction(animal:Any ){
    when (animal) {
        is Dog -> print("Animal is Dog")
        is Cat -> print("Animal is cat")
    }
}
```



Idiomatic Kotlin

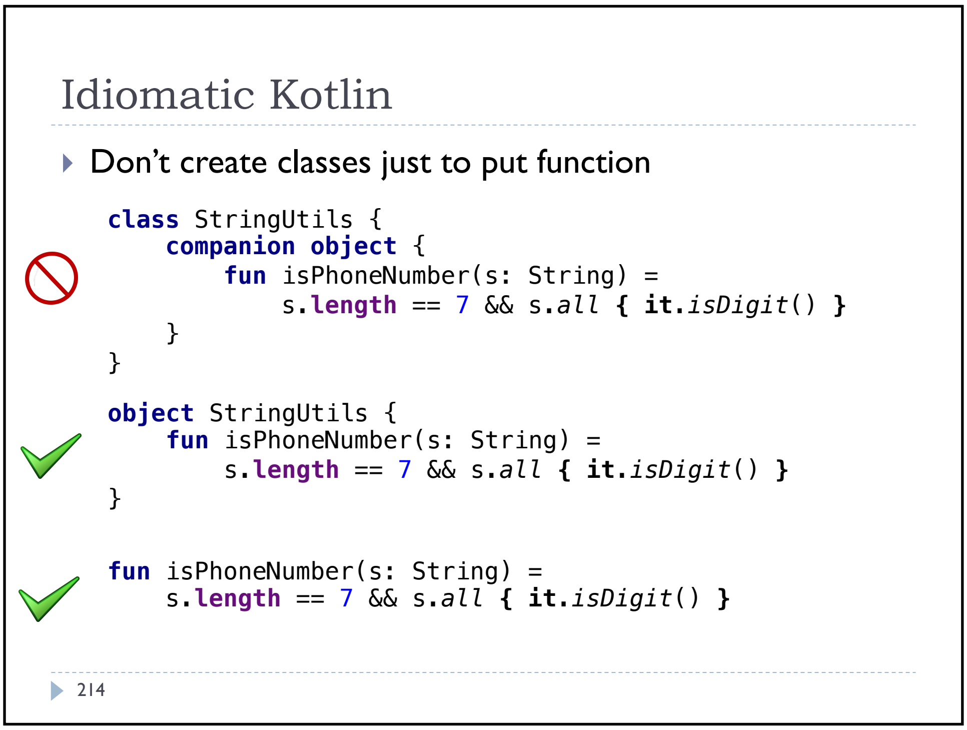
- Function Expressions

🚫

```
fun celciusToFahrenheit(celsius:Float):Float {
    return (celsius * 1.8f) + 32.0f
}
```

✓

```
fun celciusToFahrenheit(celsius:Float) =
              (celsius * 1.8f) + 32.0f
```



Idiomatic Kotlin

- Don't create classes just to put function

🚫

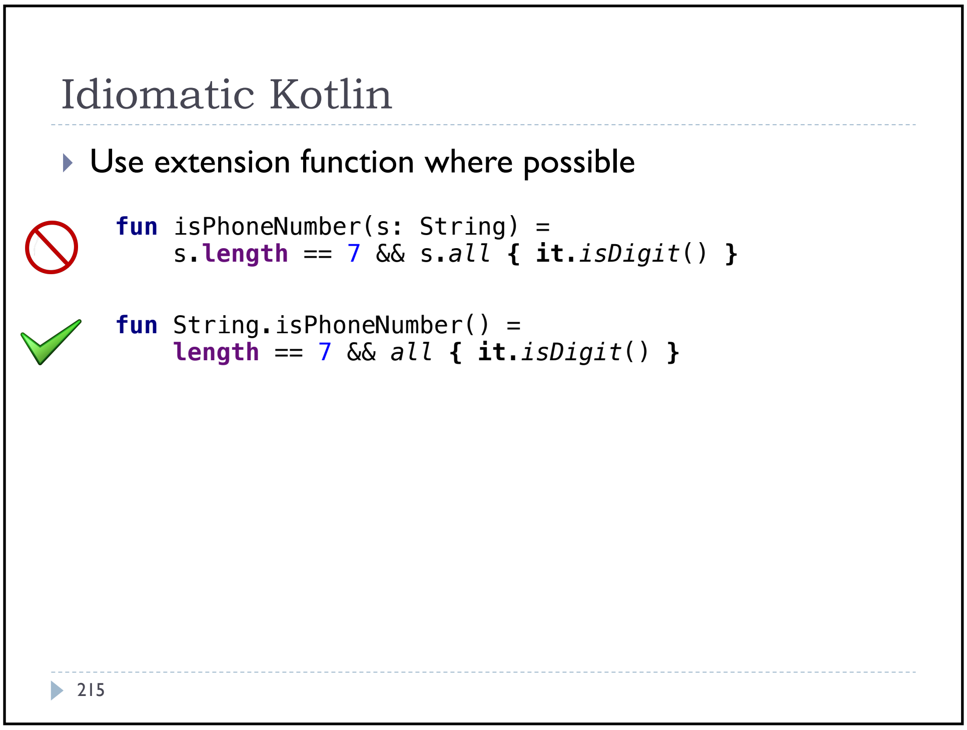
```
class StringUtils {
    companion object {
        fun isPhoneNumber(s: String) =
            s.length == 7 && s.all { it.isDigit() }
    }
}
```

✓

```
object StringUtils {
    fun isPhoneNumber(s: String) =
        s.length == 7 && s.all { it.isDigit() }
}
```

✓

```
fun isPhoneNumber(s: String) =
    s.length == 7 && s.all { it.isDigit() }
```



Idiomatic Kotlin

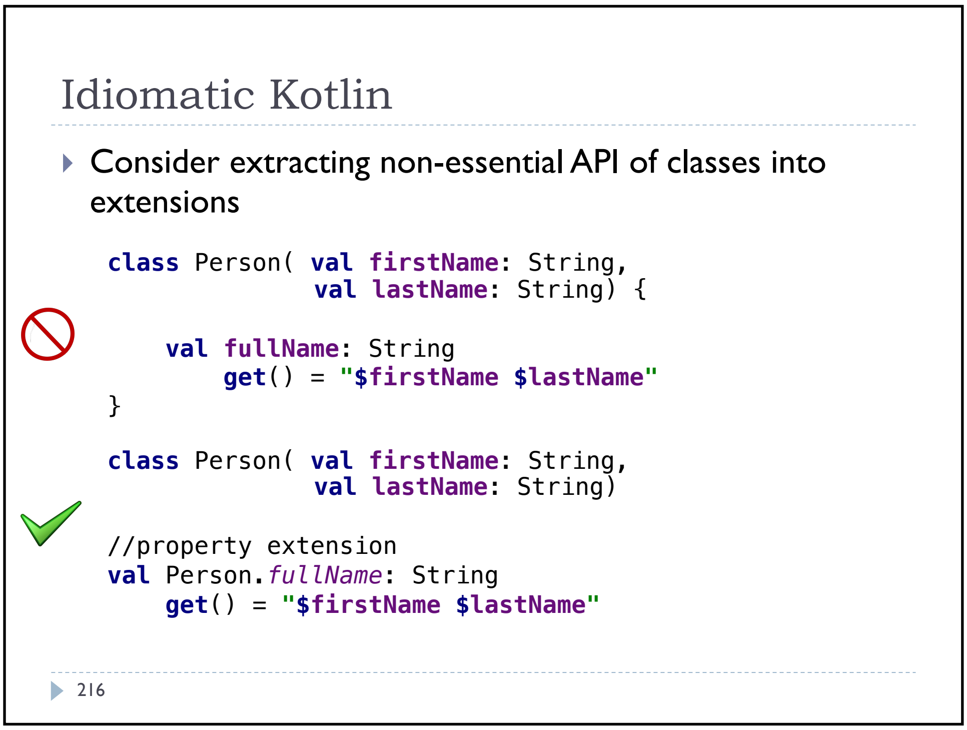
- Use extension function where possible

🚫

```
fun isPhoneNumber(s: String) =
    s.length == 7 && s.all { it.isDigit() }
```

✓

```
fun String.isPhoneNumber() =
    length == 7 && all { it.isDigit() }
```



Idiomatic Kotlin

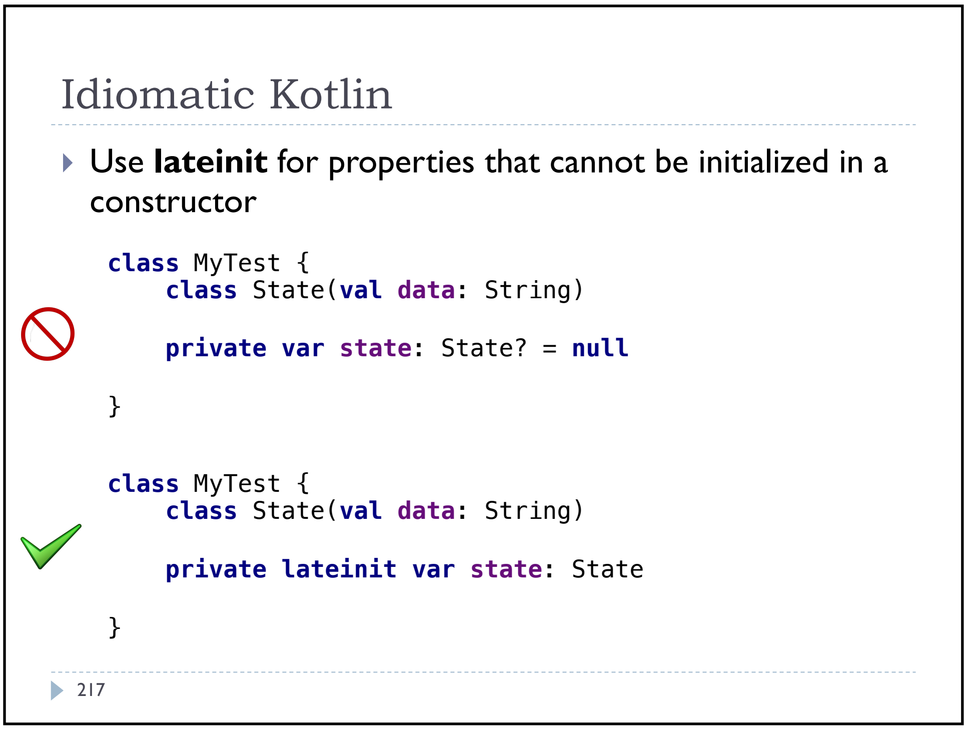
- Consider extracting non-essential API of classes into extensions

🚫

```
class Person( val firstName: String,
              val lastName: String) {

    val fullName: String
        get() = "$firstName $lastName"
}
```

✓

```
class Person( val firstName: String,
              val lastName: String)

//property extension
val Person.fullName: String
    get() = "$firstName $lastName"
```



Idiomatic Kotlin

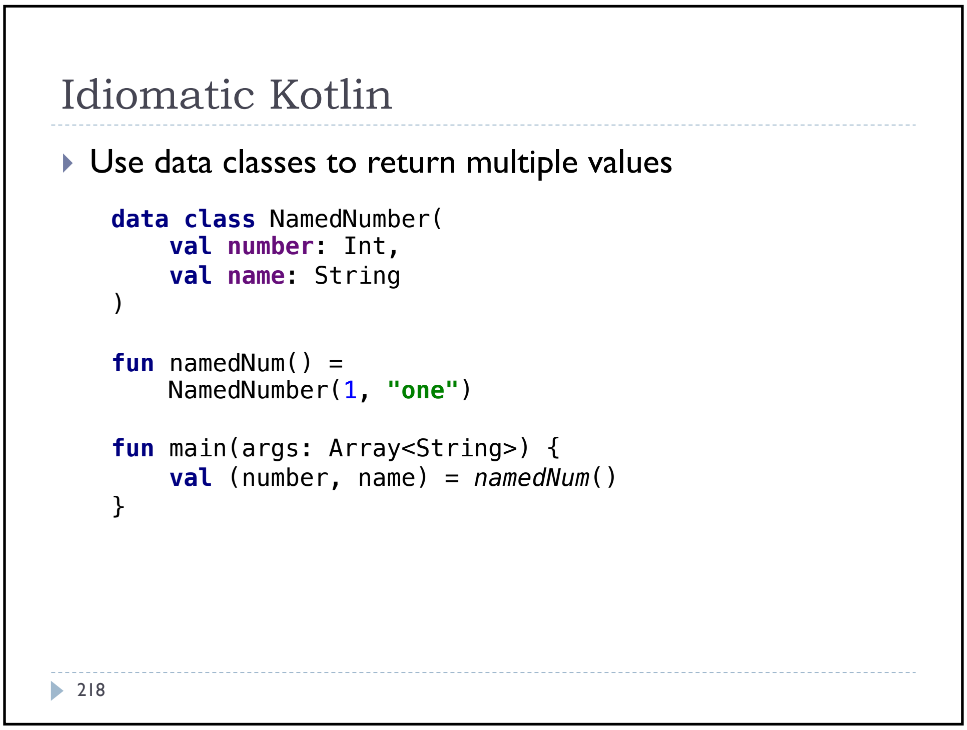
- Use **lateinit** for properties that cannot be initialized in a constructor

🚫

```
class MyTest {
    class State(val data: String)

    private var state: State? = null

}
```

✓

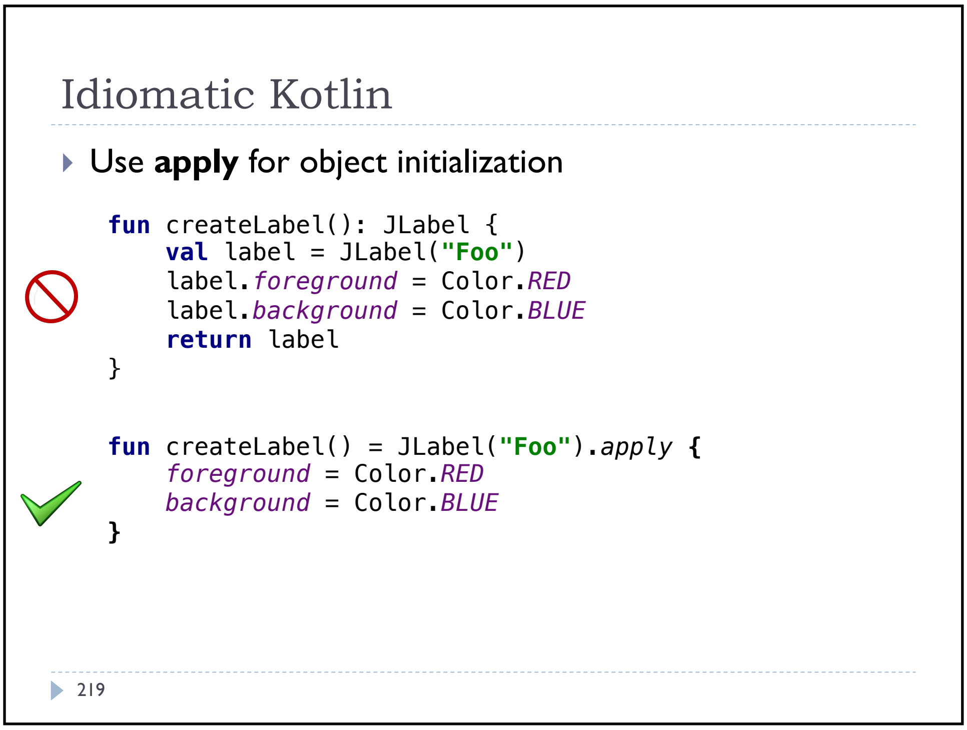
```
class MyTest {
    class State(val data: String)

    private lateinit var state: State

}
```



Idiomatic Kotlin

- Use data classes to return multiple values

```
data class NamedNumber(
    val number: Int,
    val name: String
)

fun namedNum() =
    NamedNumber(1, "one")

fun main(args: Array<String>) {
    val (number, name) = namedNum()
}
```



Idiomatic Kotlin

- Use **apply** for object initialization

🚫

```
fun createLabel(): JLabel {
    val label = JLabel("Foo")
    label.foreground = Color.RED
    label.background = Color.BLUE
    return label
}
```

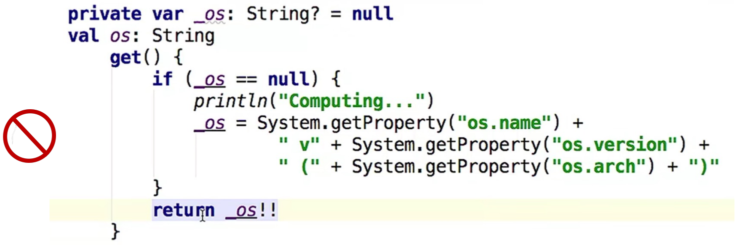
✓

```
fun createLabel() = JLabel("Foo").apply {
    foreground = Color.RED
    background = Color.BLUE
}
```

Idiomatic Kotlin

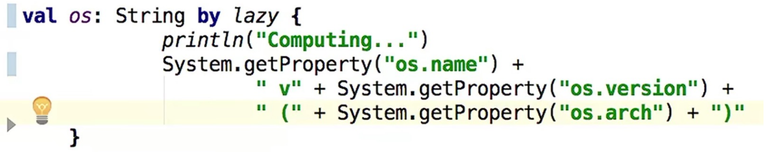
- Easy lazy properties

🚫



```
private var _os: String? = null
val os: String
    get() {
        if (_os == null) {
            println("Computing...")
            _os = System.getProperty("os.name") +
                    " v" + System.getProperty("os.version") +
                    " (" + System.getProperty("os.arch") + ")"
        }
        return _os!!
    }
```

✓



```
val os: String by lazy {
          println("Computing...")
          System.getProperty("os.name") +
                 " v" + System.getProperty("os.version") +
                 " (" + System.getProperty("os.arch") + ")"
    }
```