Kotlin Training Setup Instructions

- Follow the setup instructions prior to the training

- If there are any questions please contact the trainer:
  - Bjorn van der Laan / bvanderlaan@xebia.com
  - Urs Peter / upeter@xebia.com

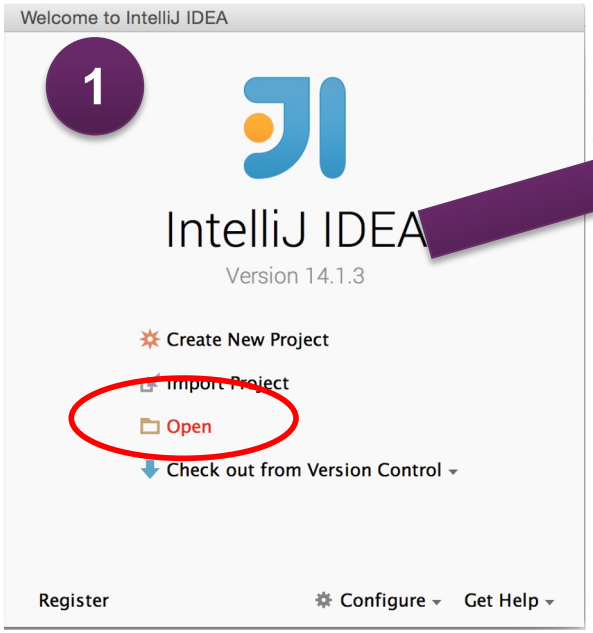
Setup Instructions for Intellij

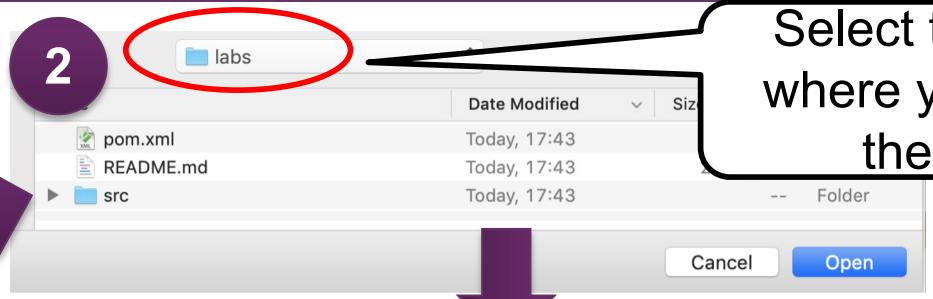
- Ensure you have Admin rights enabled on your PC
- Make sure you have JDK >=17 installed
  - https://www.oracle.com/java/technologies/downloads/
- Make sure you have the latest version (2022.3.2) of Intellij installed
  - https://www.jetbrains.com/idea/download/#section=windows
- Import lab sources *(also see screenshots)*
  - Download and unzip:
    - https://bit.ly/kotlin-training-ah-plsql-labs-2023
  - In the lab root directory run:

    ```
    mvn clean test
    ```

    - All tests will fail
  - Import Project in Intellij:
    - File -> Open-> select lab root directory -> Ok
    - Install the Kotest plugin found under:
      - Preferences -> Plugins -> Marketplace
    - From within the IDE make sure you can run all kotlin and java tests

Screenshots Project Import in Intellij

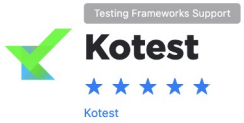




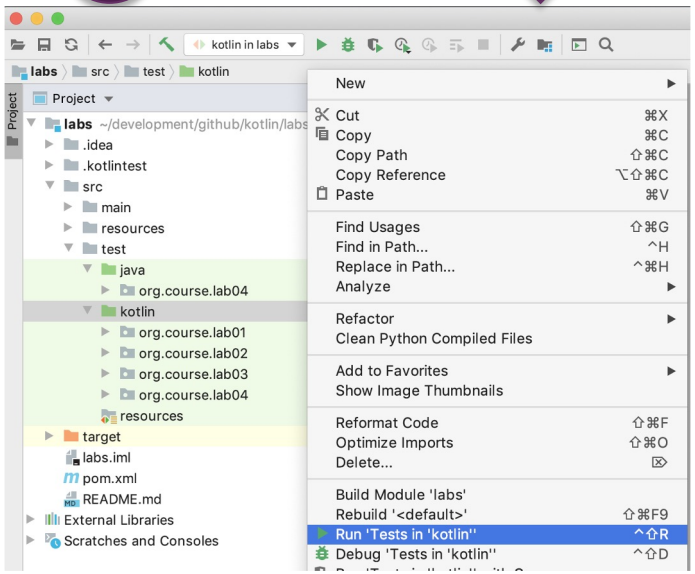
Select the directory where you unzipped the labs file

3

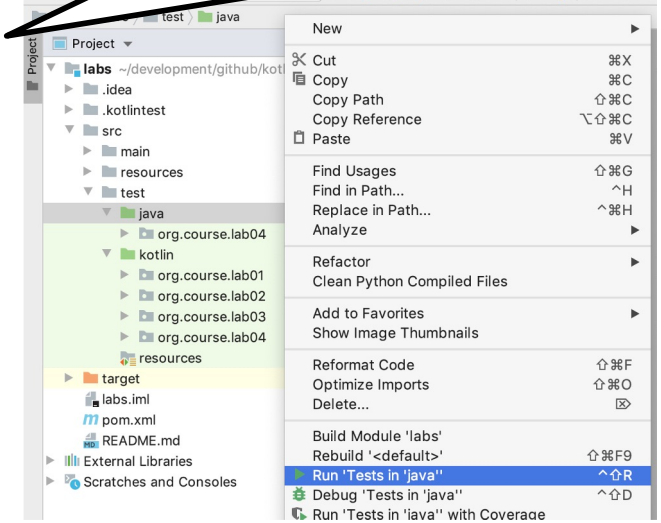
Install the Kotest plugin found under:
Preferences -> Plugins -> Marketplace



4



Make sure you can run the java and kotlin tests (though they fail for now).



Xebia

Kotlin Training
Certified by JetBrains

Xebia Academy

Professional Development with

Kotlin

Urs Peter

upeter@xebia.com

Bjorn van der Laan

bvanderlaan@xebia.com

Sources and Slides

- Slides:
  - https://bit.ly/kotlin-training-ah-plsql-slides-2023

- Sources:
  - https://bit.ly/kotlin-training-ah-plsql-labs-2023

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

Some Facts about Kotlin

Designed by: JetBrains
Developed by: JetBrains and Open source contributors
Run ons: The Java Virtual Machine (JVM) and native platforms
(Android & all major operating systems)

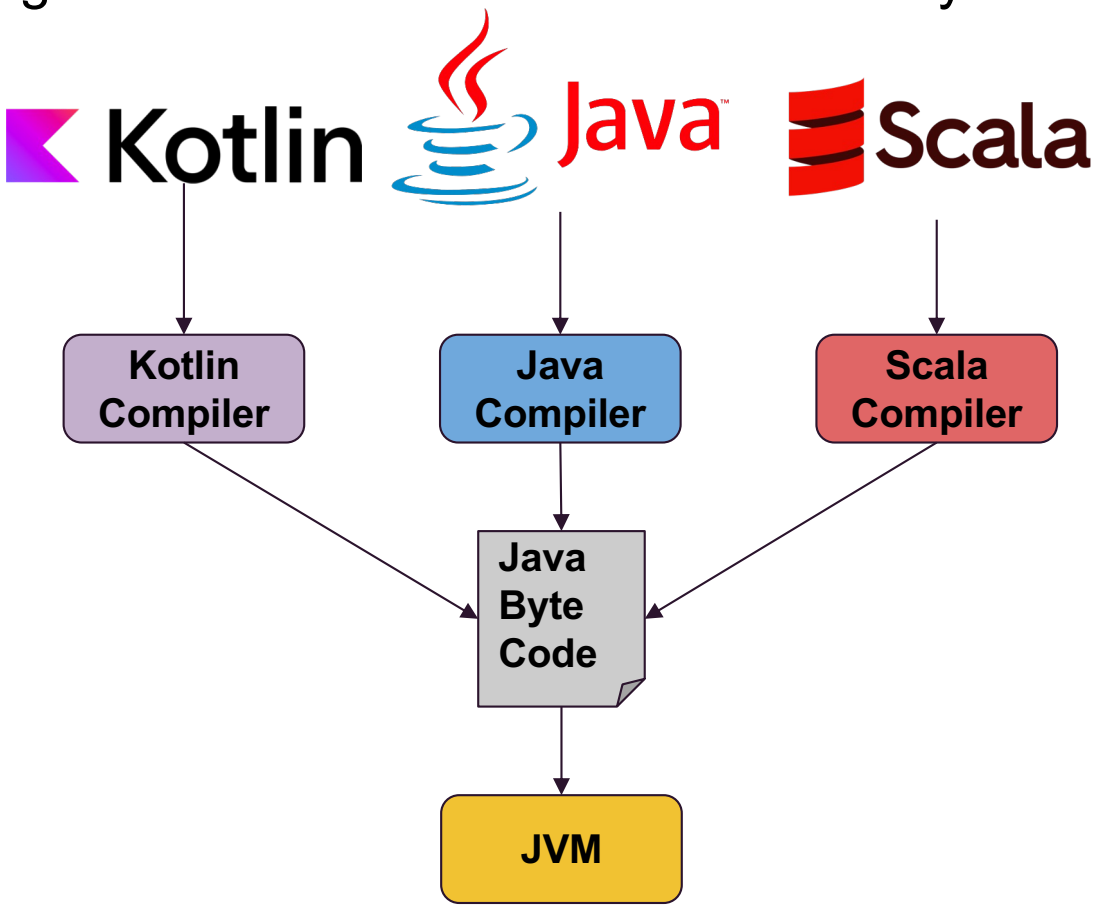
Born in: 2011
1.0 Release in: 2016
Since 2017:

- officially supported by Google for Android Studio 3.0
- official support for Spring framework 5.0

The Kotlin story starts with the JVM

Kotlin's primary platform is the Java Virtual Machine (JVM)
The JVM is a *multilingual* platform:

- Any language that can be translated into Java Byte Code can run on the JVM



Philosophy of Kotlin

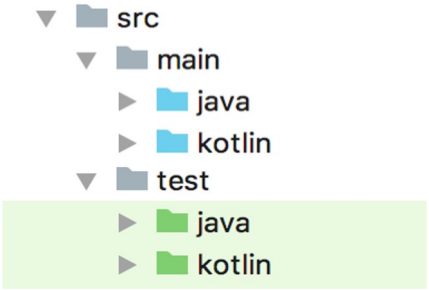
Kotlin is designed with a *better Java* in mind:

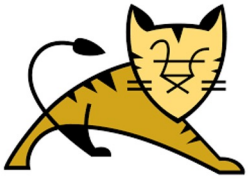
- purely *object-oriented* with *functions* as first class citizens
- aims for maximal *reduction* of *boilerplate*
- safety through *null-types* and *immutability*
- extensibility through *extension functions*
- perfect tooling support
- pragmatic

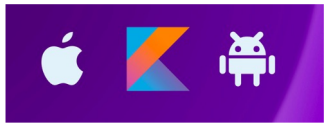
...yet fully interoperable with Java allowing

for gradual migration from Java to Kotlin

Some Facts about Kotlin







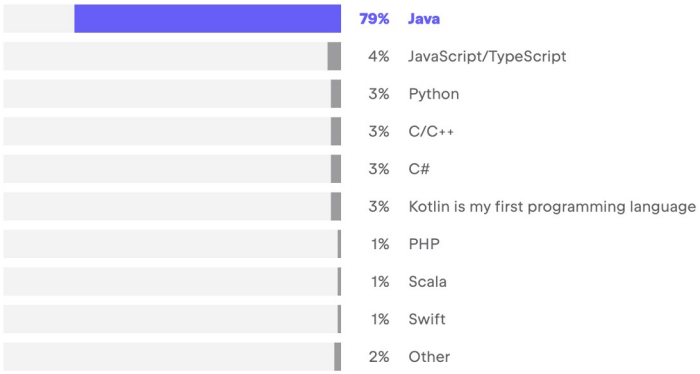
Kotlin is fully interoperable with
Java Frameworks and Tools
… but also provides its own

Some Facts about Kotlin

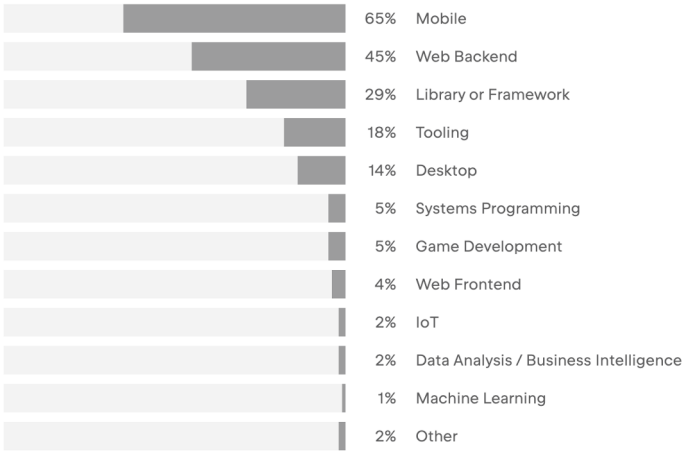
Kotlin is used in many world-class companies

Some Facts about Kotlin

What was your primary programming language before you switched to Kotlin?

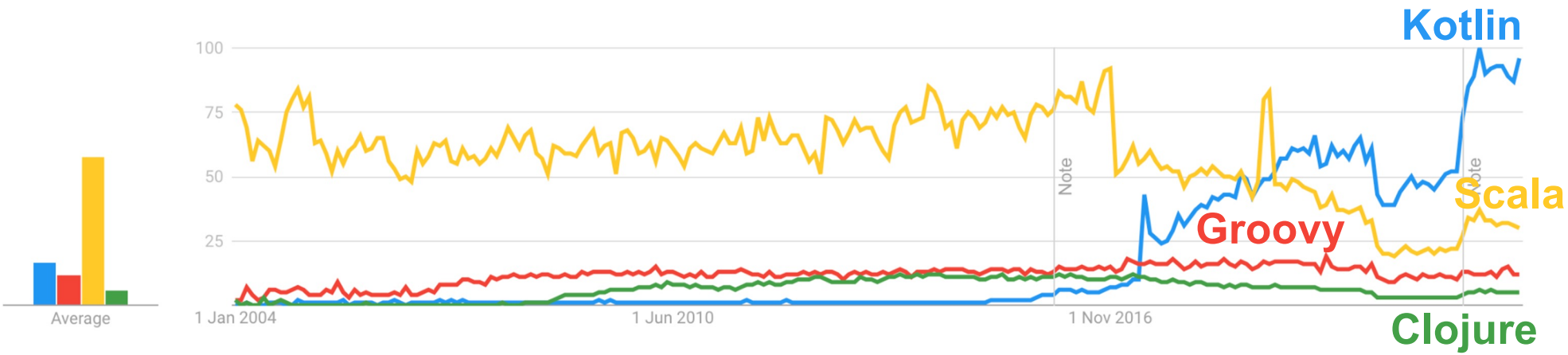


What types of software do you develop with Kotlin?



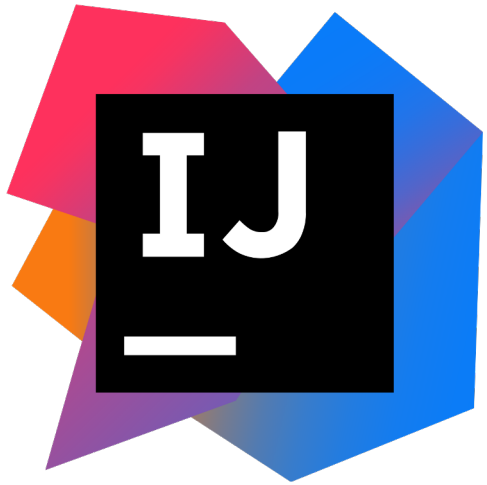
https://www.jetbrains.com/lp/devecosystem-2022/kotlin/

Interest over time



https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0_lcrx4,%2Fm%2F02js86,%2Fm%2F091hdj,%2Fm%2F03yb8hb

Facts about Kotlin



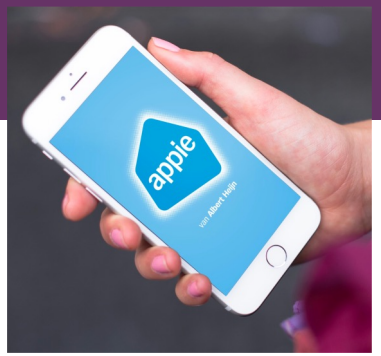
Great IDE support in Intellij:

Java to Kotlin converter

Kotlin REPL

Kotlin Worksheet

Structure of a Kotlin project




```
└── appie-app-backend
    ├── pom.xml
    └── src
        ├── main
        │   ├── kotlin
        │   │   └── com
        │   │       └── aholddelhaize
        │   │           └── appie
        │   │               ├── Main.kt
        │   │               ├── Helpers.kt
        │   │               └── database
        │   │                   └── DatabaseConnection.kt
        │   └── resources
        │       └── configuration.yml
        └── test
            └── kotlin
                └── com
                    └── aholddelhaize
                        └── appie
                            ├── HelpersTest.kt
                            └── database
                                └── DatabaseConnectionTest.kt
```

"Maven" project file that describes the project and how it should be built.

/src/main/kotlin contains all the code of this project

The code is further structured in directories called *packages*

Convention: internet domain of company in reverse + project name

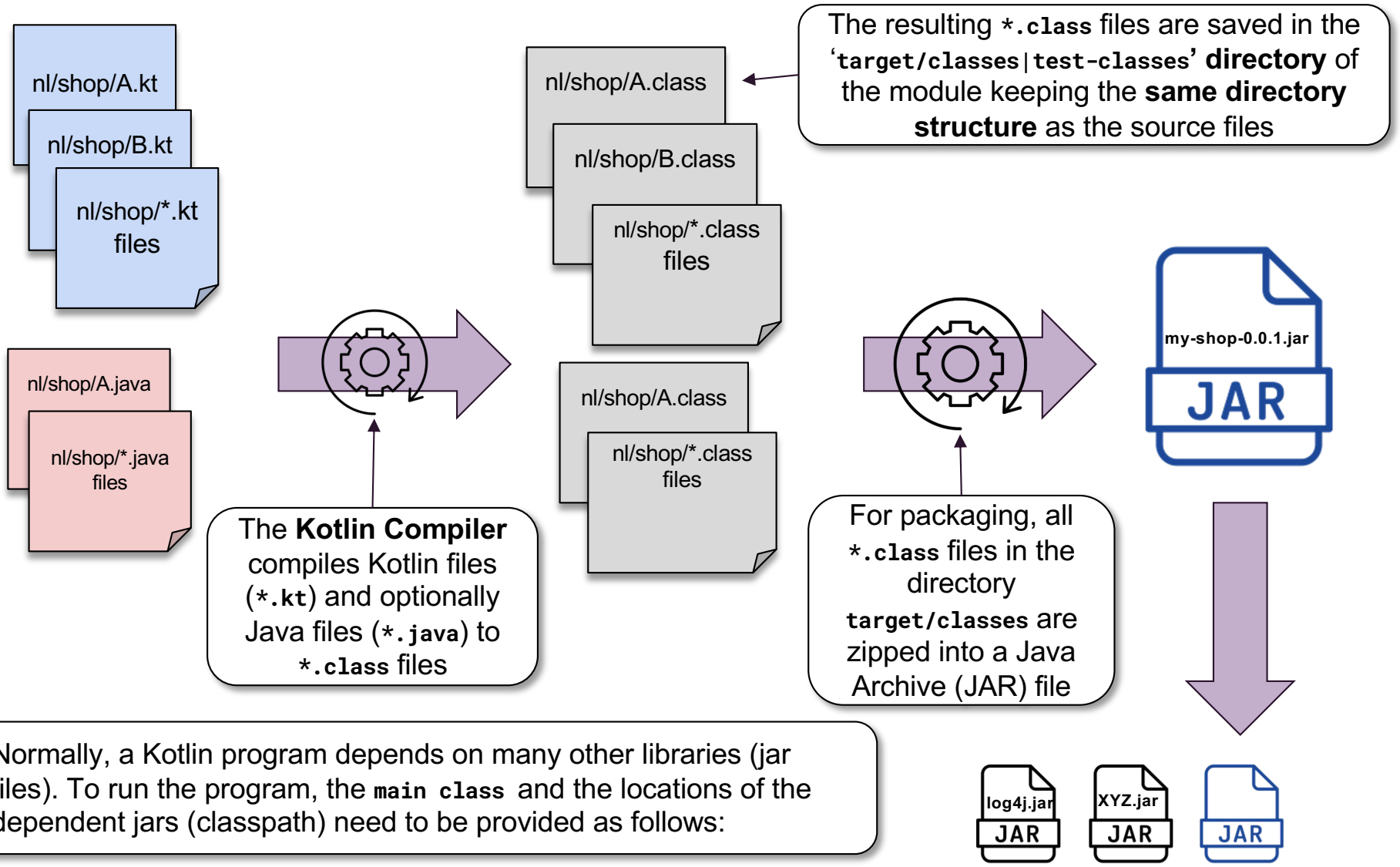
Kotlin uses file extension .kt

/src/resources contains configuration files

/src/test/kotlin contains all the automated tests

Package structure of tests is the same as structure of the code.

Packaging your project: the JAR file



Normally, a Kotlin program depends on many other libraries (jar files). To run the program, the `main class` and the locations of the dependent jars (classpath) need to be provided as follows:

```
java -cp <path>/log4j.jar;<path>/XYZ.jar;<path>/my-shop-0.0.1.jar nl.shop.MyMainClassKt
```

How to manage your JAR

- Building a JAR for your project can be complicated, especially for bigger projects.
- Apache Maven is a build automation tool that handles this for you!
- Every project contains a *pom.xml* file that describes the dependencies your project contains and how to build and package your JAR

In the top section **`groupId`**, **`artifact`** and **`version`** depict the unique identification of your project and resulting artifacts (like the jar)

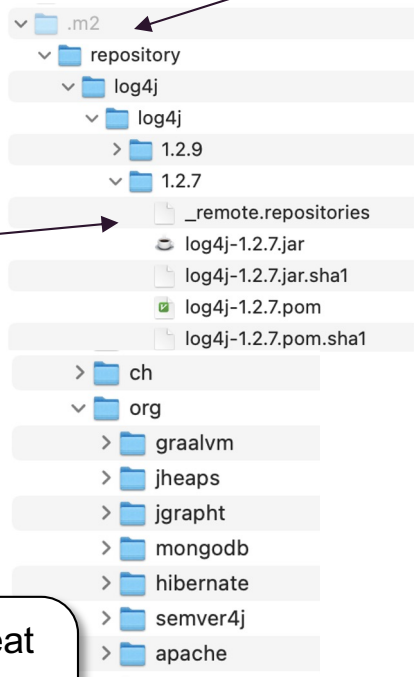
Dependency management is key for every project. It allows (re)-using external libraries.

To use a dependency, simply declare one in the **`dependencies`** section

pom.xml

```
<project>
 <groupId>nl.shop</groupId>
 <artifactId>my-shop</artifactId>
 <version>0.0.1</version>
...

<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.7</version>
  </dependency>
...

<build>
  <plugins>
<groupId>org.apache.maven</groupId>
<artifactId>maven-compiler</artifactId>
<version>${maven-compiler.version}</version>
...
```

Maven stores jar files (your own and all dependencies) under **`~/.m2/repository/…`**



In the **`build`** section, plugins can be configured to do a great variety of tasks such as: *compiling, testing, packaging (jar, zip, docker image and much more), reporting, security checks, code quality checks, bugs analysis etc.*

Some Facts about Kotlin

Designed by: JetBrains
Developed by: JetBrains and Open source contributors
Run ons: The Java Virtual Machine (JVM) and native platforms
(Android & all major operating systems)

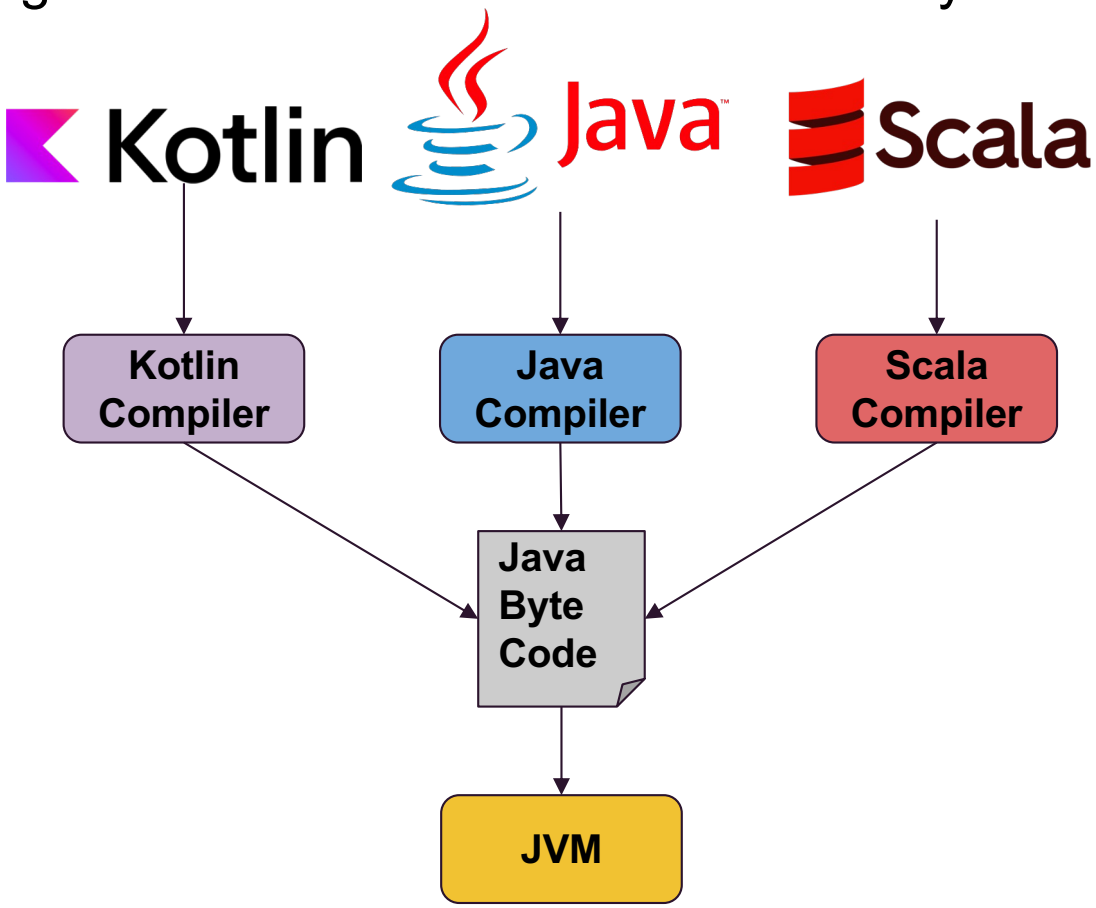
Born in: 2011
1.0 Release in: 2016
Since 2017:

- officially supported by Google for Android Studio 3.0
- official support for Spring framework 5.0

The Kotlin story starts with the JVM

Kotlin's primary platform is the Java Virtual Machine (JVM)
The JVM is a *multilingual* platform:

- Any language that can be translated into Java Byte Code can run on the JVM



Philosophy of Kotlin

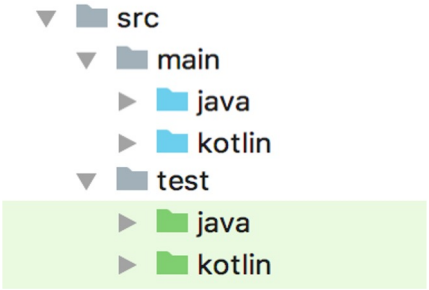
Kotlin is designed with a *better Java* in mind:

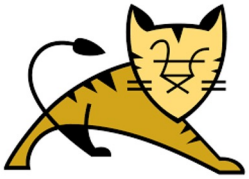
- purely *object-oriented* with *functions* as first class citizens
- aims for maximal *reduction* of *boilerplate*
- safety through *null-types* and *immutability*
- extensibility through *extension functions*
- perfect tooling support
- pragmatic

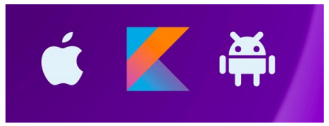
...yet fully interoperable with Java allowing

for gradual migration from Java to Kotlin

Some Facts about Kotlin







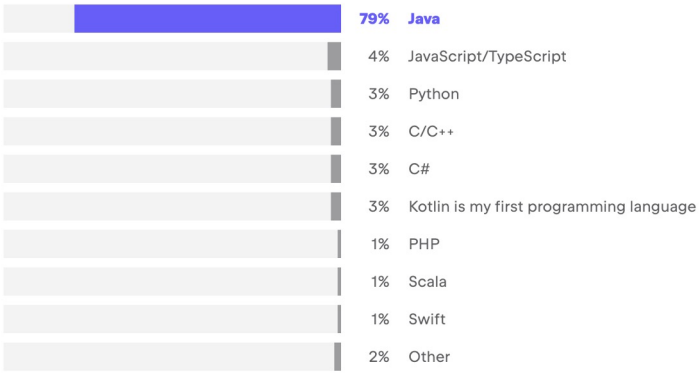
Kotlin is fully interoperable with
Java Frameworks and Tools
… but also provides its own

Some Facts about Kotlin

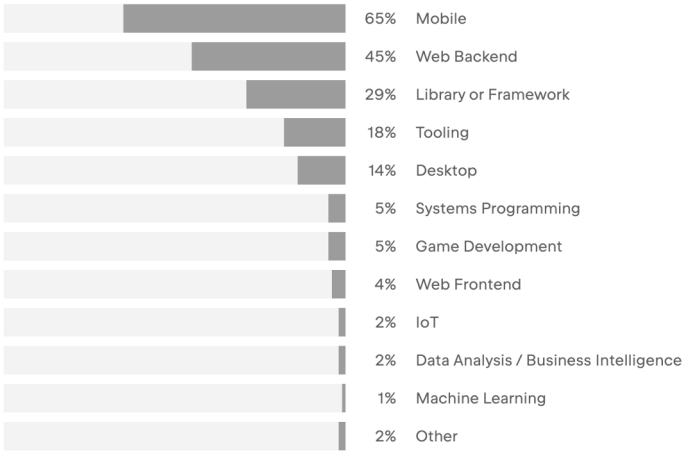
Kotlin is used in many world-class companies

Some Facts about Kotlin

What was your primary programming language before you switched to Kotlin?

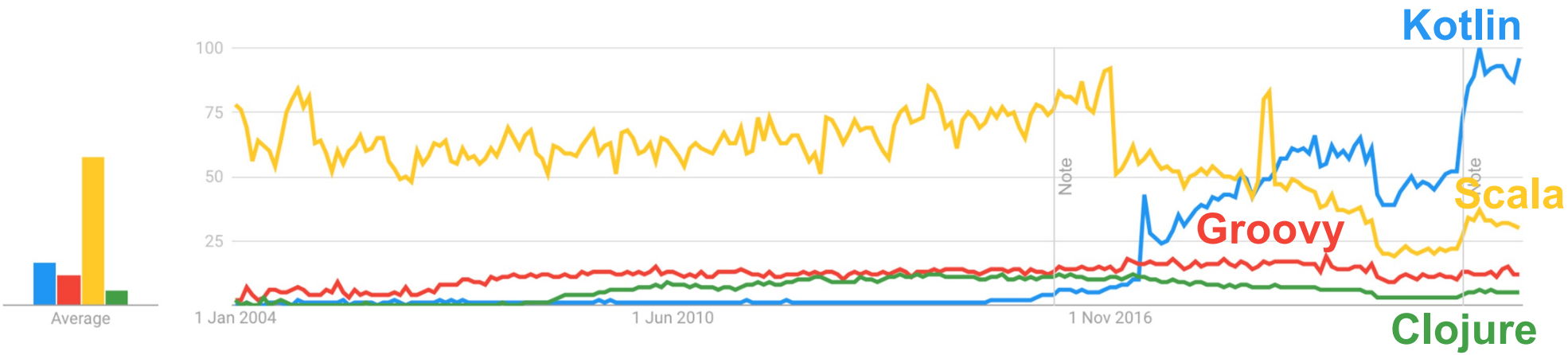


What types of software do you develop with Kotlin?



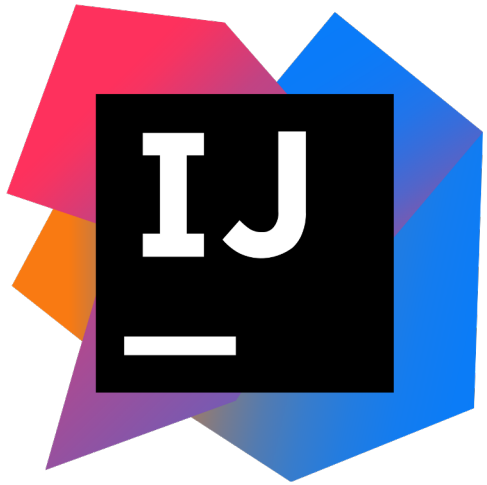
https://www.jetbrains.com/lp/devecosystem-2022/kotlin/

Interest over time



https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0_lcrx4,%2Fm%2F02js86,%2Fm%2F091hdj,%2Fm%2F03yb8hb

Facts about Kotlin



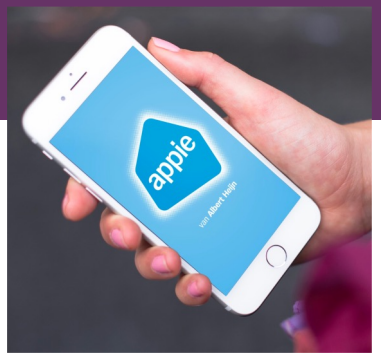
Great IDE support in Intellij:

Java to Kotlin converter

Kotlin REPL

Kotlin Worksheet

Structure of a Kotlin project




```
└── appie-app-backend
    ├── pom.xml
    └── src
        ├── main
        │   ├── kotlin
        │   │   └── com
        │   │       └── aholddelhaize
        │   │           └── appie
        │   │               ├── Main.kt
        │   │               ├── Helpers.kt
        │   │               └── database
        │   │                   └── DatabaseConnection.kt
        │   └── resources
        │       └── configuration.yml
        └── test
            └── kotlin
                └── com
                    └── aholddelhaize
                        └── appie
                            ├── HelpersTest.kt
                            └── database
                                └── DatabaseConnectionTest.kt
```

"Maven" project file that describes the project and how it should be built.

/src/main/kotlin contains all the code of this project

The code is further structured in directories called *packages*

Convention: internet domain of company in reverse + project name

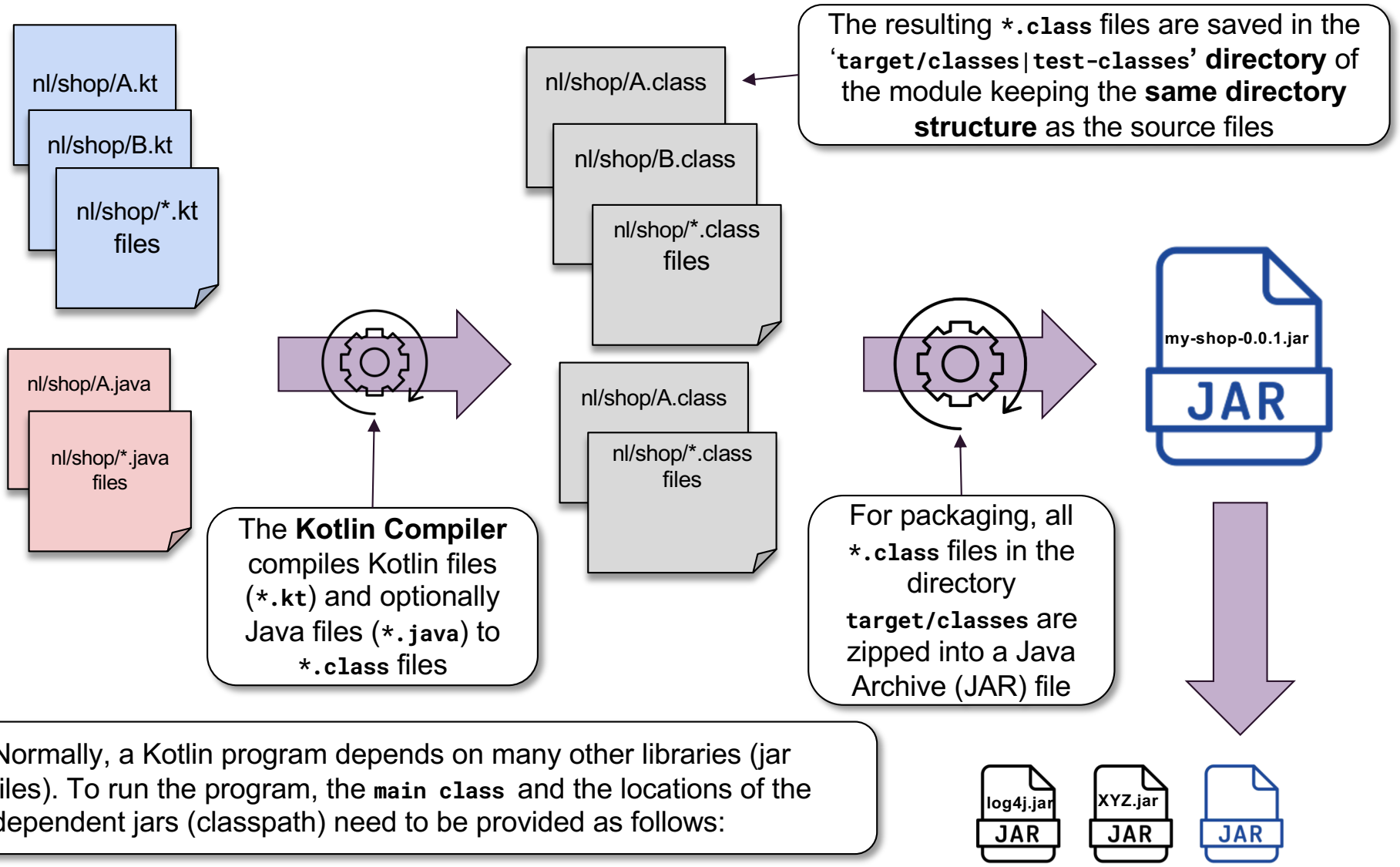
Kotlin uses file extension .kt

/src/resources contains configuration files

/src/test/kotlin contains all the automated tests

Package structure of tests is the same as structure of the code.

Packaging your project: the JAR file



Normally, a Kotlin program depends on many other libraries (jar files). To run the program, the `main class` and the locations of the dependent jars (classpath) need to be provided as follows:

```
java -cp <path>/log4j.jar;<path>/XYZ.jar;<path>/my-shop-0.0.1.jar nl.shop.MyMainClassKt
```

How to manage your JAR

- Building a JAR for your project can be complicated, especially for bigger projects.
- Apache Maven is a build automation tool that handles this for you!
- Every project contains a *pom.xml* file that describes the dependencies your project contains and how to build and package your JAR

In the top section **`groupId`**, **`artifact`** and **`version`** depict the unique identification of your project and resulting artifacts (like the jar)

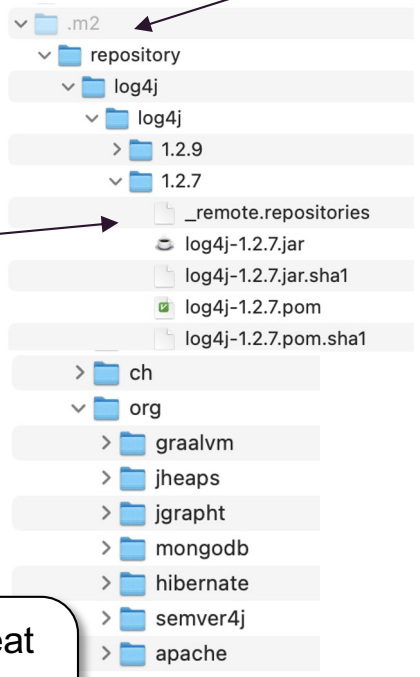
Dependency management is key for every project. It allows (re)-using external libraries.

To use a dependency, simply declare one in the **`dependencies`** section

pom.xml

```
<project>
 <groupId>nl.shop</groupId>
 <artifactId>my-shop</artifactId>
 <version>0.0.1</version>
...

<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.7</version>
  </dependency>
...

<build>
  <plugins>
<groupId>org.apache.maven</groupId>
<artifactId>maven-compiler</artifactId>
<version>${maven-compiler.version}</version>
...
```

Maven stores jar files (your own and all dependencies) under **`~/.m2/repository/…`**



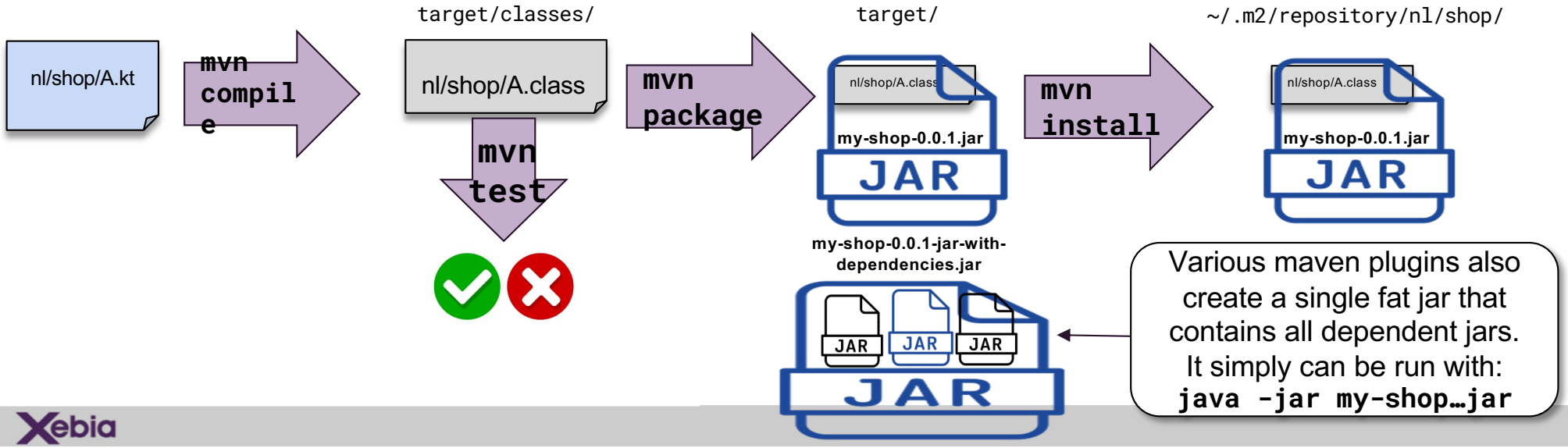
In the **`build`** section, plugins can be configured to do a great variety of tasks such as: *compiling, testing, packaging (jar, zip, docker image and much more), reporting, security checks, code quality checks, bugs analysis etc.*

How to manage your JAR

Maven™

Important Maven commands - which can all be combined:

`mvn clean`	delete all files under the **`/target`** folder
`mvn compile`	compile all classes the **`/target/classes`**
`mvn test`	compile and runs all tests
`mvn test -Dtest=MyTest`	compile and a specific test
`mvn package`	compiles, tests and packages all classes in the project jar file
`mvn package -DskipTests`	same like package but does not execute tests
`mvn install`	copies the packaged jar to the **`.m2`** repository, in the directory computed by: **`groupId/artifactId-<version>.jar`** Example: **`nl/shop/my-shop-0.0.1.jar`**



Exercise: first steps in Kotlin and Maven

https://bit.ly/kotlin-training-ah-plsql-slides-2023

Time to Create, Build and Run our first application!

1. Run your first Kotlin application from the IDE
   - In the project you will find a file named: `SampleMain.kt`
   - Run its main method from your IDE:

```
fun main(args:Array<String>) {
    val today : LocalDateTime! = LocalDateTime.now()
    println("Hello Kotlin! How are you today at $today? Args: ${args.joinToString()}")
}
```

1. Debug the Kotlin application from the IDE
   - Set a breakpoint, run the main in debug mode and inspect the value of `today`

```
fun main(args:Array<String>) {
    val today : LocalDateTime! = LocalDateTime.now()
    println("Hello Kotlin! How are you today at $today? Args: ${args.joinToString()}")
}
```

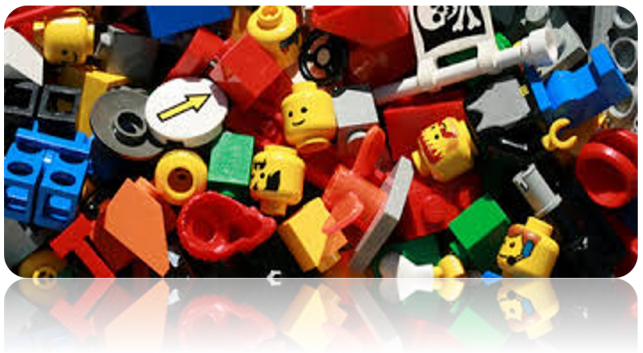
1. Figure out the correct maven command to build the jar file.
   - Note: The build will not succeed because the tests fail. How can you skip the tests?
   - Once the jar is built, try to run it with the `java` command. Ensure that you also add the kotlin-stdlib jar to the classpath, located at: `<path-to-your-userhome-dir>/.m2/repository/org/jetbrains/kotlin/kotlin-stdlib/1.7.20/kotlin-stdlib-1.7.20.jar`.

1. **Bonus:** Execute the runnable jar

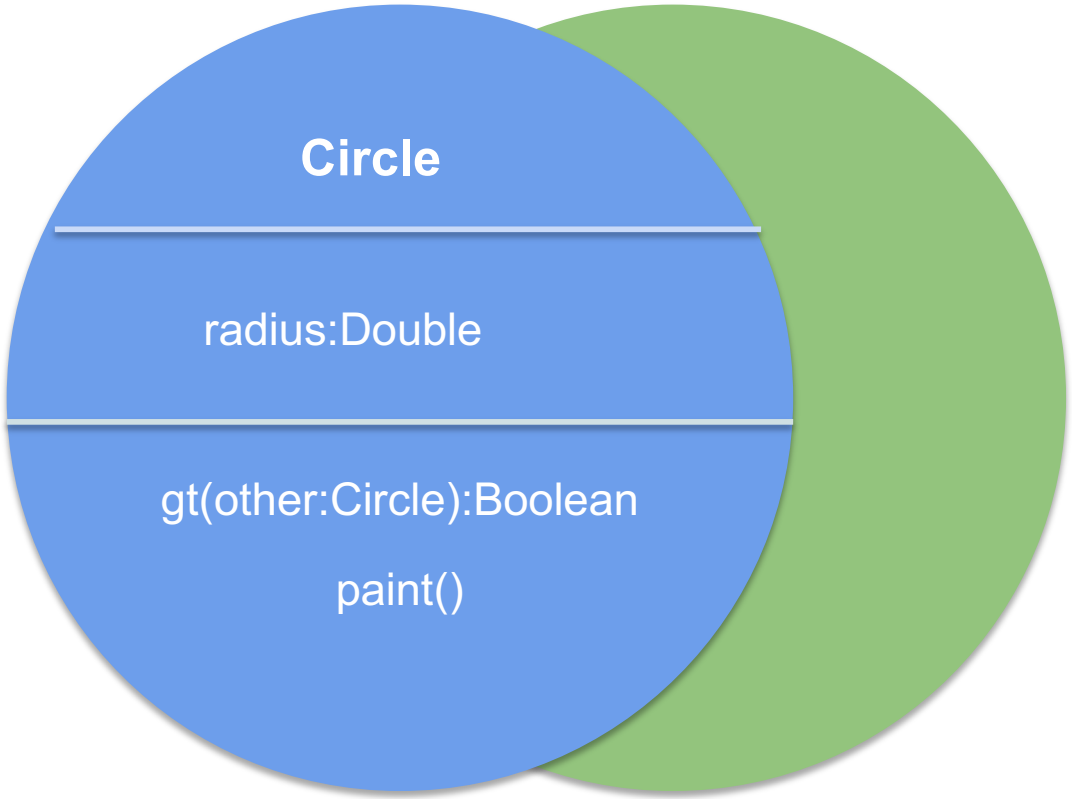
- Inspect the target folder where you will find a jar named: `labs-1.0-jar-with-dependencies.jar`
  - Try to run it with `java -jar` (the classpath (`-cp`) argument can be omitted)

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

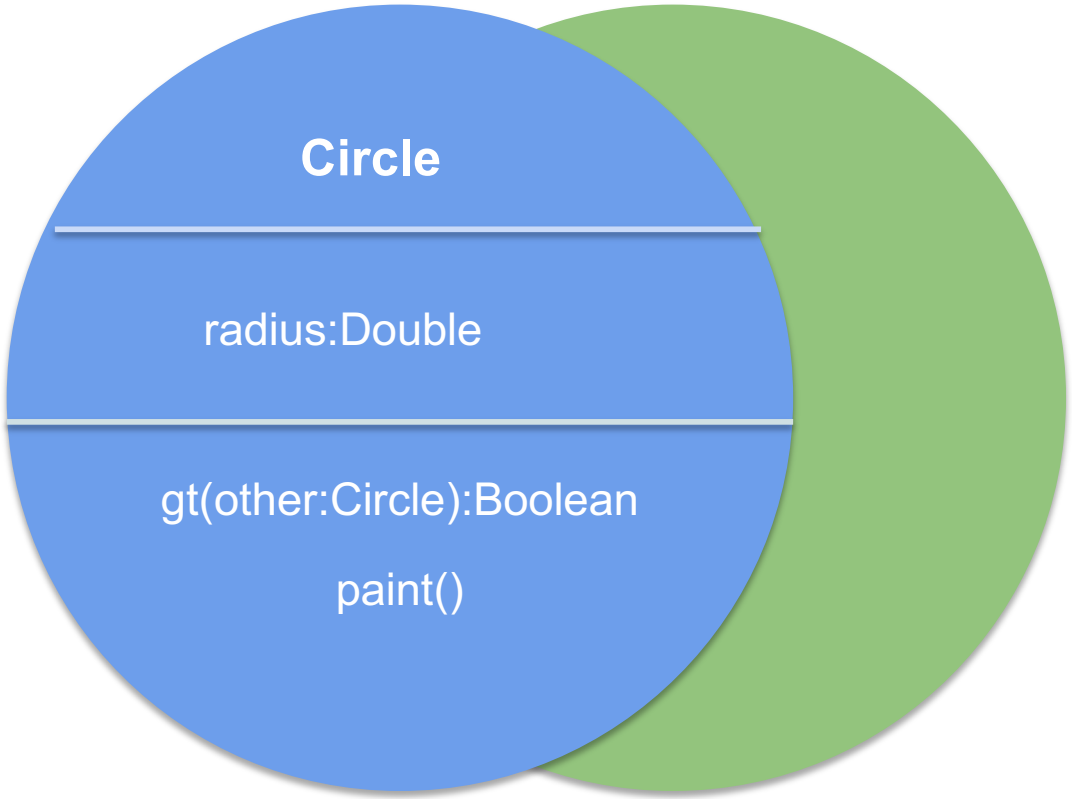


Object Orientation Basics



Kotlin is truly OO

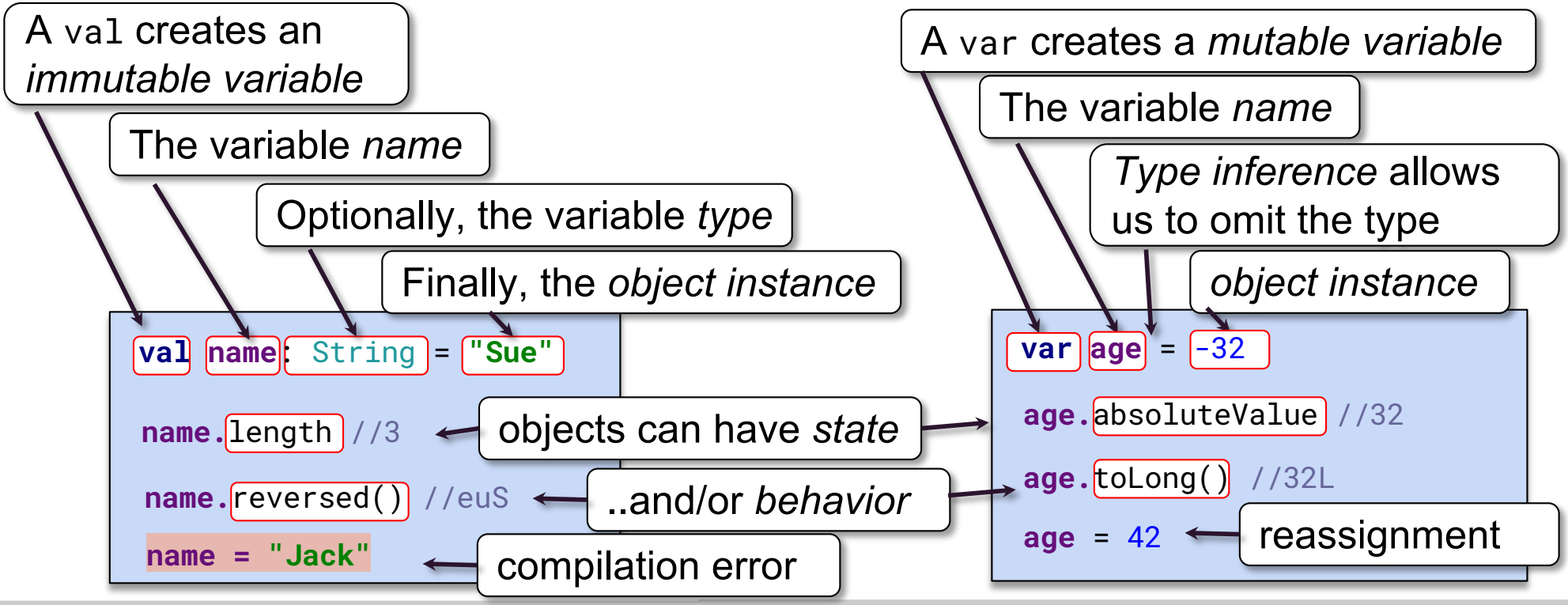
Object Orientation Basics



Kotlin is truly OO

Object Oriented Programming

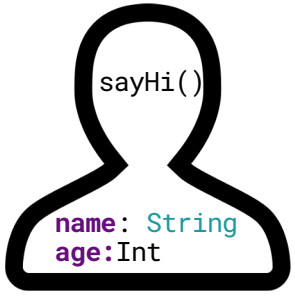
- Kotlin is an *object oriented* programming language.
- In Kotlin *everything* is an object, no matter whether it is a *basic type* like:
  - Numeric Types: `Byte, Short, Int, Long`
  - Floating Point Types: `Float, Double`
  - Characters & Strings: `Char, String`
- …or more complex types like `Date, Collections,` etc.
- Once created, objects can be referenced either by `val` or `var`.



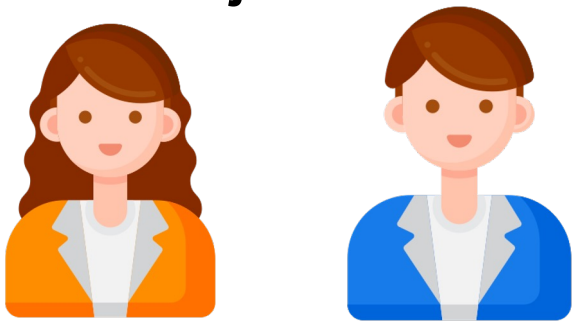
Object Oriented Programming

- Object orientation allows designing software in terms of *objects* that can be related to the 'real world', like a Person.
- The blueprint of an object is called a *class*

Person Class



Person Objects/Instances



state of `Person`

```
class Person(val name: String,
             var age: Int      ){

  fun sayHi() = "Hi I'm $name"
}
```

behaviour of `Person`

To create and pass state to a `Person` *instance*, we need a *constructor*

```
val sue =  Person("Sue", 32)
sue.name //Sue
sue.sayHi() //Hi I'm Sue
```

By calling the constructor a Person instance is created

```
var joe =  Person("Joe", 42)
joe.name //Joe
joe.sayHi() //Hi I'm Joe
joe = sue
```

Since `joe` *is a* `var`*, re-assignment is possible*

Comparing OO between PL/SQL & Kotlin

```
CREATE OR REPLACE TYPE person AS OBJECT (
 name VARCHAR2(30),
 age NUMBER,
 CONSTRUCTOR FUNCTION person
          RETURN SELF AS RESULT,
 CONSTRUCTOR FUNCTION person (
  ( name VARCHAR2, age NUMBER )
          RETURN SELF AS RESULT,
 MEMBER FUNCTION get_name RETURN VARCHAR2,
 MEMBER FUNCTION get_age RETURN NUMBER,
 MEMBER PROCEDURE set_age(pv_age NUMBER),
 MEMBER FUNCTION say_hi RETURN VARCHAR2,
) INSTANTIABLE NOT FINAL;
```

```
CREATE OR REPLACE TYPE BODY Person AS
  MEMBER PROCEDURE set_age( pv_age INTEGER
) IS                BEGIN
          self.age := pv_age;
    END set_age;
  MEMBER FUNCTION get_age RETURN INTEGER IS
     BEGIN
          RETURN self.age;
    END get_age;
  MEMBER FUNCTION get_name RETURN VARCHAR2
IS
     BEGIN
          RETURN self.name;
    END get_name;
  MEMBER FUNCTION say_hi RETURN VARCHAR2 IS
     BEGIN
          RETURN 'Hi';
    END say_hi;
END;
```

A *specification* (*interface*) is optional - more on that later

```
class Person(val name: String,
             var age: Int) {

  fun sayHi() = "Hi"

}
```

A PL/SQL body is identical to a `class`.

Comparing OO between PL/SQL & Kotlin

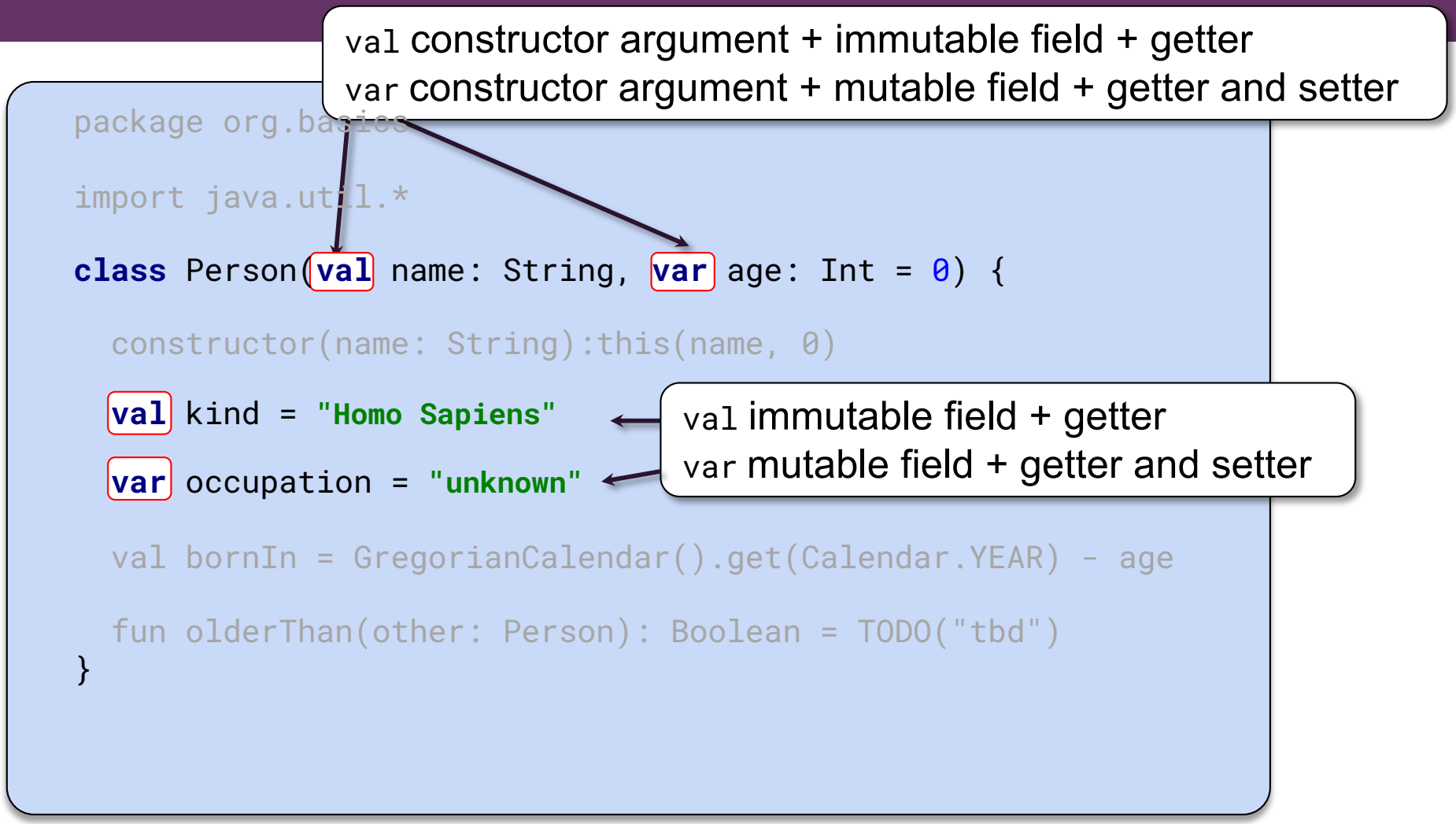
```
DECLARE
  p Person;
BEGIN
  p := Person('Jack', 42);

  dbms_output.put_line(p.say_hi());

  p.set_age(39);

  dbms_output.put_line(p.age);
END;
```

```
val p = Person("Jack", 42)
println(p.sayHi())
p.age = 39
println(p.age)
```

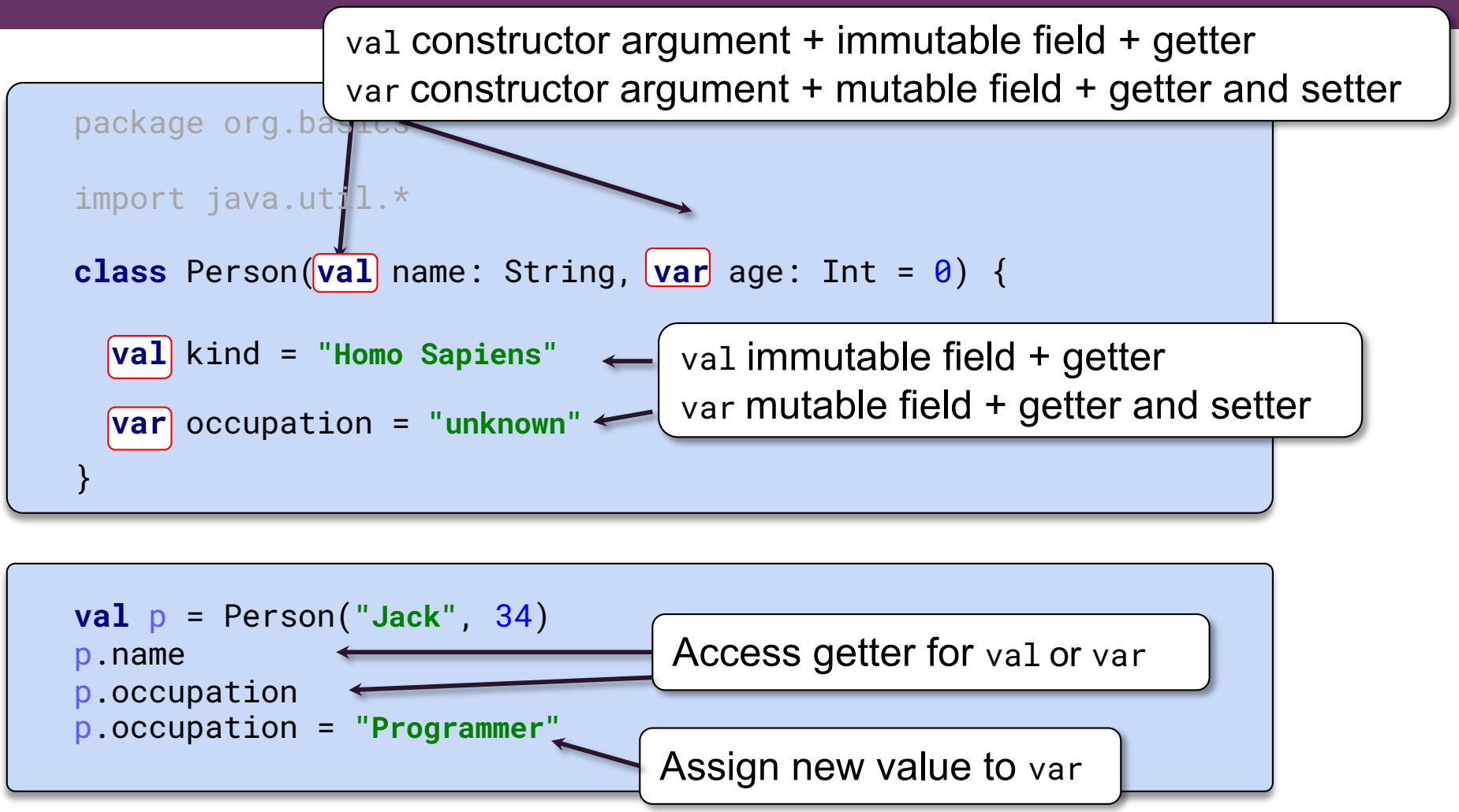
Object Orientation Basics

```
package org.basics

import java.util.*

class Person(val name: String, var age: Int) {

  constructor(name: String):this(name, 0)

  val kind = "Homo Sapiens"

  var occupation = "unknown"

  val bornIn = GregorianCalendar().get(Calendar.YEAR) - age

  fun olderThan(other: Person): Boolean = TODO("tbd")
}
```

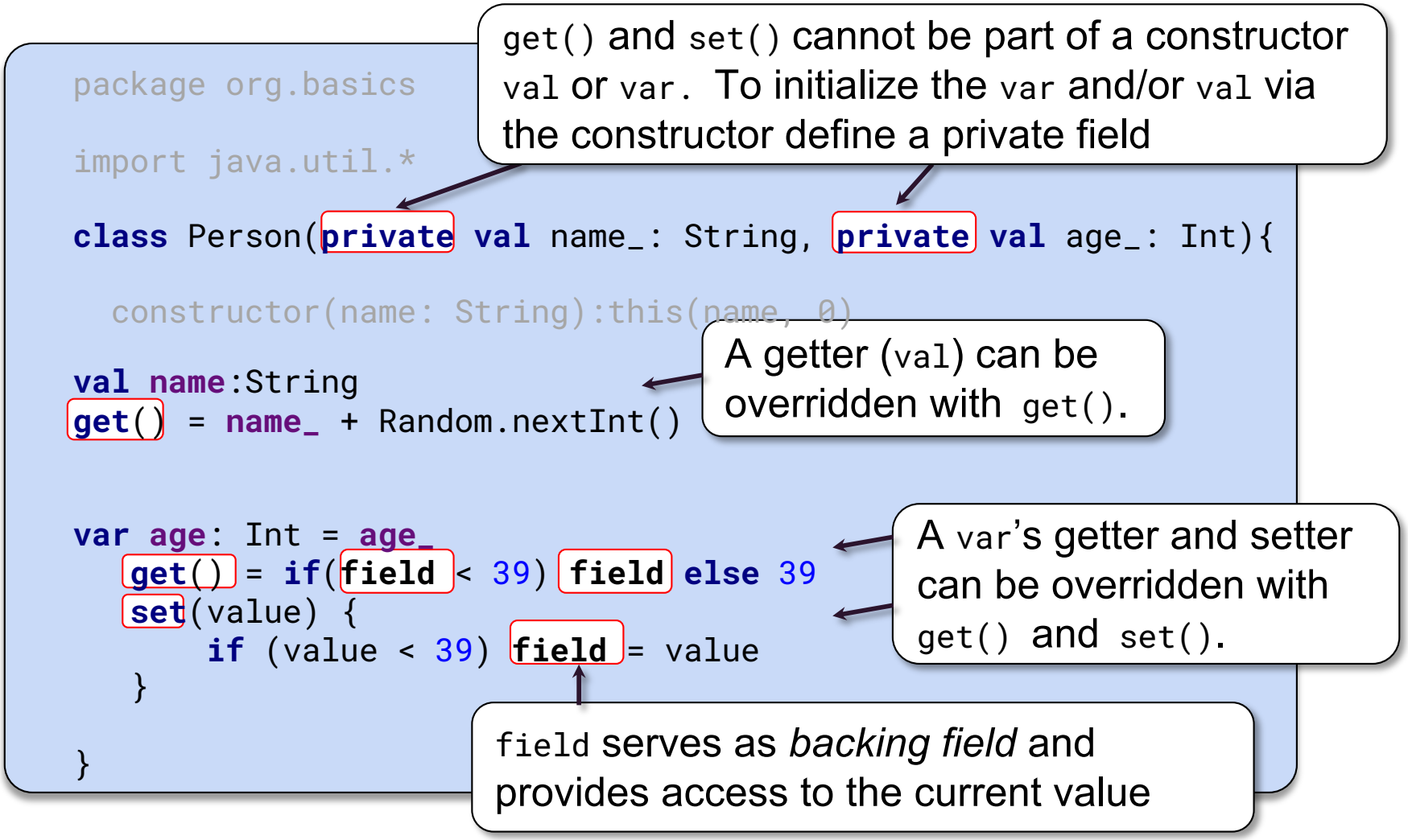
Fields, Getters and Setters: val's and var's



Fields, Getters and Setters: val's and var's



Fields, Getters and Setters: getters and setters



Fields, Getters and Setters: getters and setters

```
package org.basics

import java.util.*

class Person(private val name_: String, private val age_: Int){

  constructor(name: String):this(name, 0)

val name:String
get() = name_ + Random.nextInt()


var age: Int = age_
   get() = if(field < 39) field else 39
   set(value) {
       if (value < 39) field = value
   }
}
```

```
val p = Person("Jack", 34)
p.name      //Jack9349348934
p.age       //34
p.age = 48
p.age       //39
```

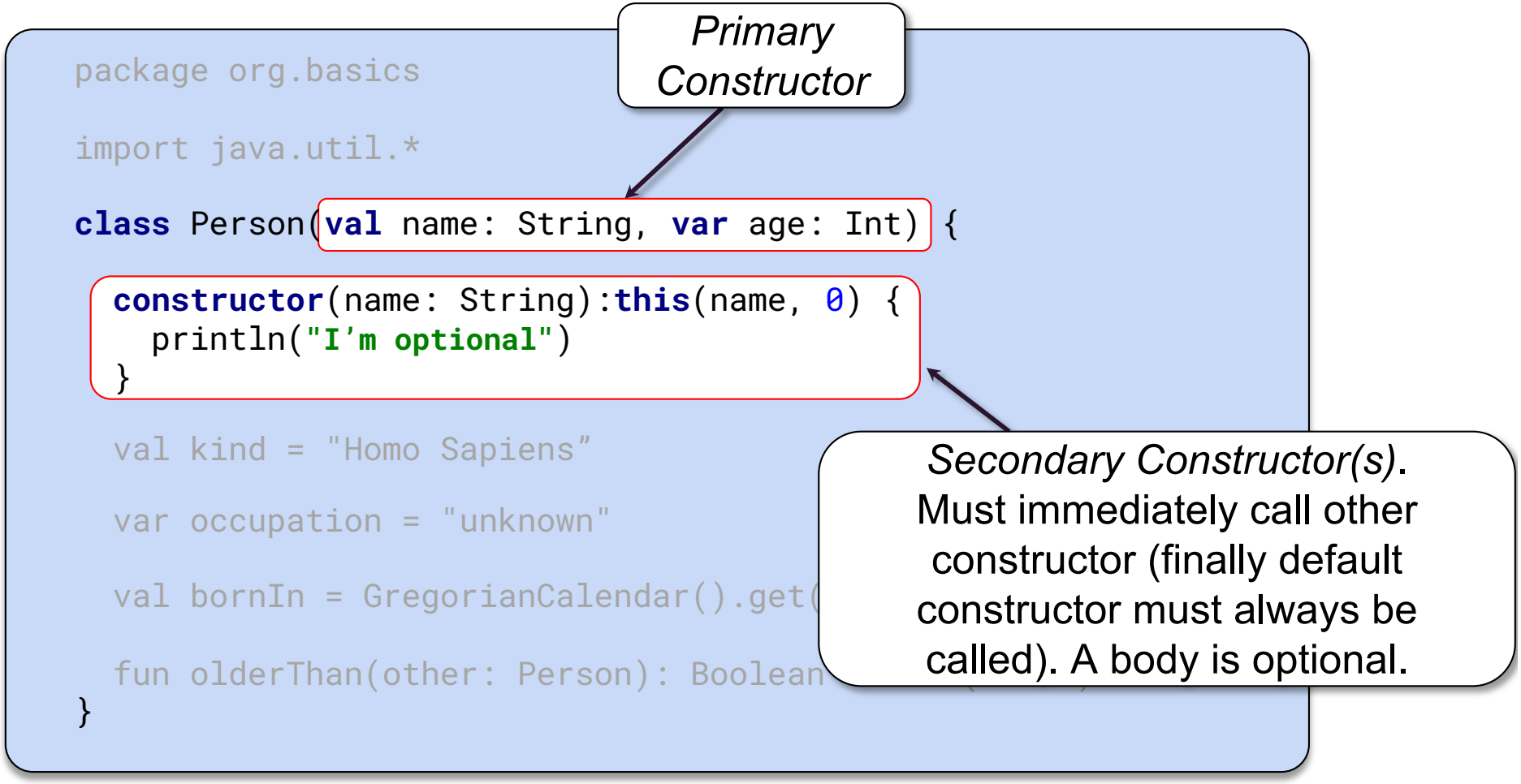
Default Arguments

Constructor arguments and methods can have default arguments

```
package org.basics

import java.util.*

class Person(val name: String, var age: Int = 0) {
  constructor(name: String):this(name, 0)
  val kind = "Homo Sapiens"
  var occupation = "unknown"
  val bornIn = GregorianCalendar().get(Calendar.YEAR) - age
  fun olderThan(other: Person): Boolean = TODO("tbd")
}
```

```
val p = Person("Jack")
p.name //Jack
p.age //0
```

Constructor



Constructor Scope

```
package org.basics

import java.util.*

class Person(val name: String, var age: Int = 0) {

  constructor(name: String):this(name, 0)

  val kind = "Homo Sapiens"

  var occupation = "unknown"

  val bornIn = GregorianCalendar().get(Calendar.YEAR) - age

  fun olderThan(other: Person): Boolean = TODO("tbd")

}
```

All assignments in *class scope* belong to the *primary constructor scope*

```
val p = Person("Jack")
p.bornIn  //2023
```

Initializer Block

```
package org.basics

import java.util.*

class Person(val name: String, var age: Int = 0) {

  constructor(name: String):this(name, 0)

  val kind = "Homo Sapiens"

  var occupation = "unknown"

  val bornIn:Int
  init {
     bornIn = GregorianCalendar().get(Calendar.YEAR) - age
  }

  fun olderThan(other: Person): Boolean = TODO("tbd")
}
```

In addition, an initializer block can be used for initialization logic. `init` is part of the *primary constructor*

Methods: single expression functions

```
package org.basics

import java.util.*

class Person(val name: String, var age: Int = 0) {

  constructor(name: String):this(name, 0)

  val kind = "Homo Sapiens"

  val unknown = "unknown"

  val birthYear = Calendar.getInstance().get(Calendar.YEAR) - age

  fun olderThan(other: Person): Boolean = true

}
```

Method declaration. Public by default (as all fields)

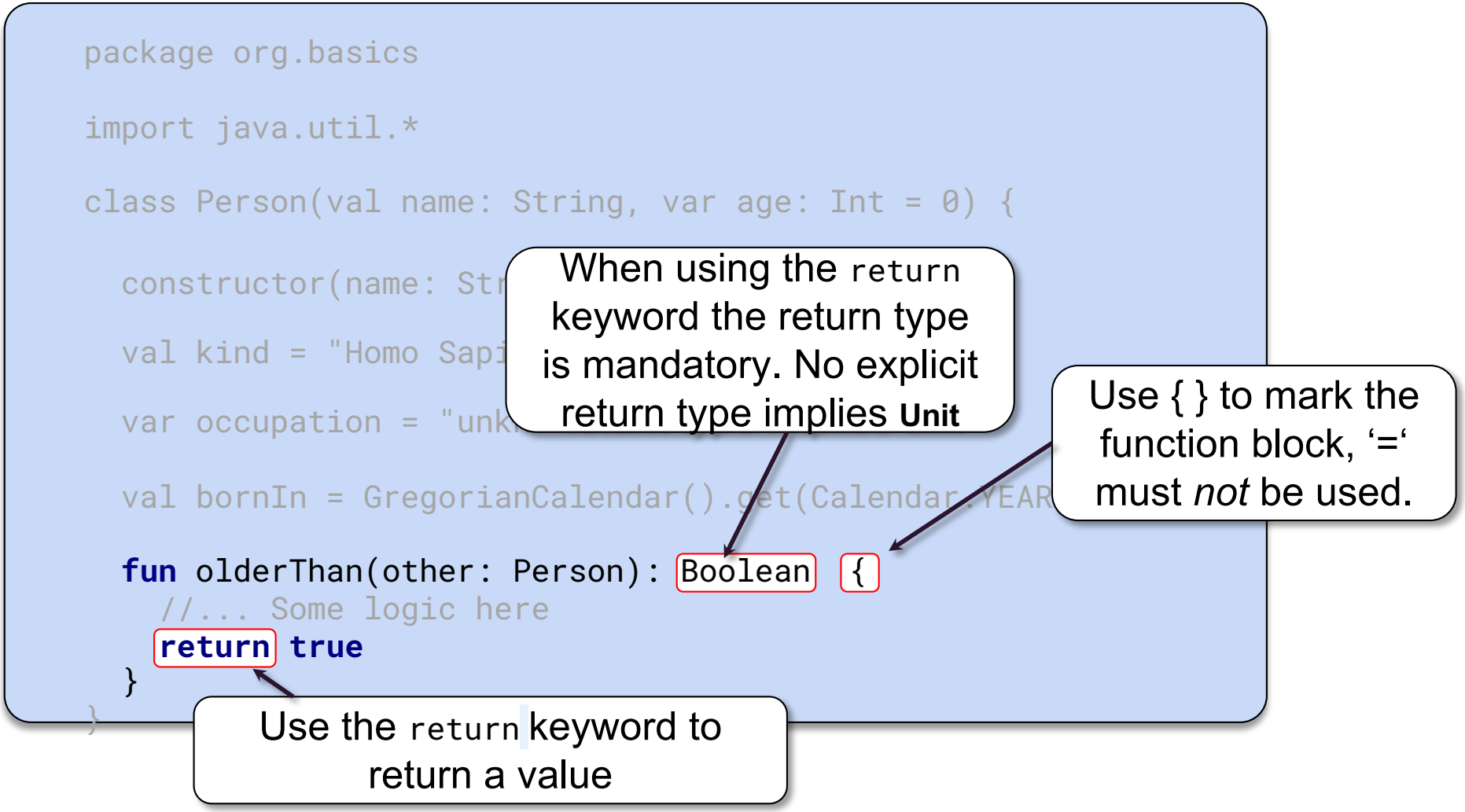
If return value can be inferred it is optional.

For single expression functions '=' is mandatory

No `return` keyword needed for single expression functions

```
val jack = Person("Jack", 42)
jack.olderThan(Person("Fred", 26))   //true
```

Methods: multiline functions

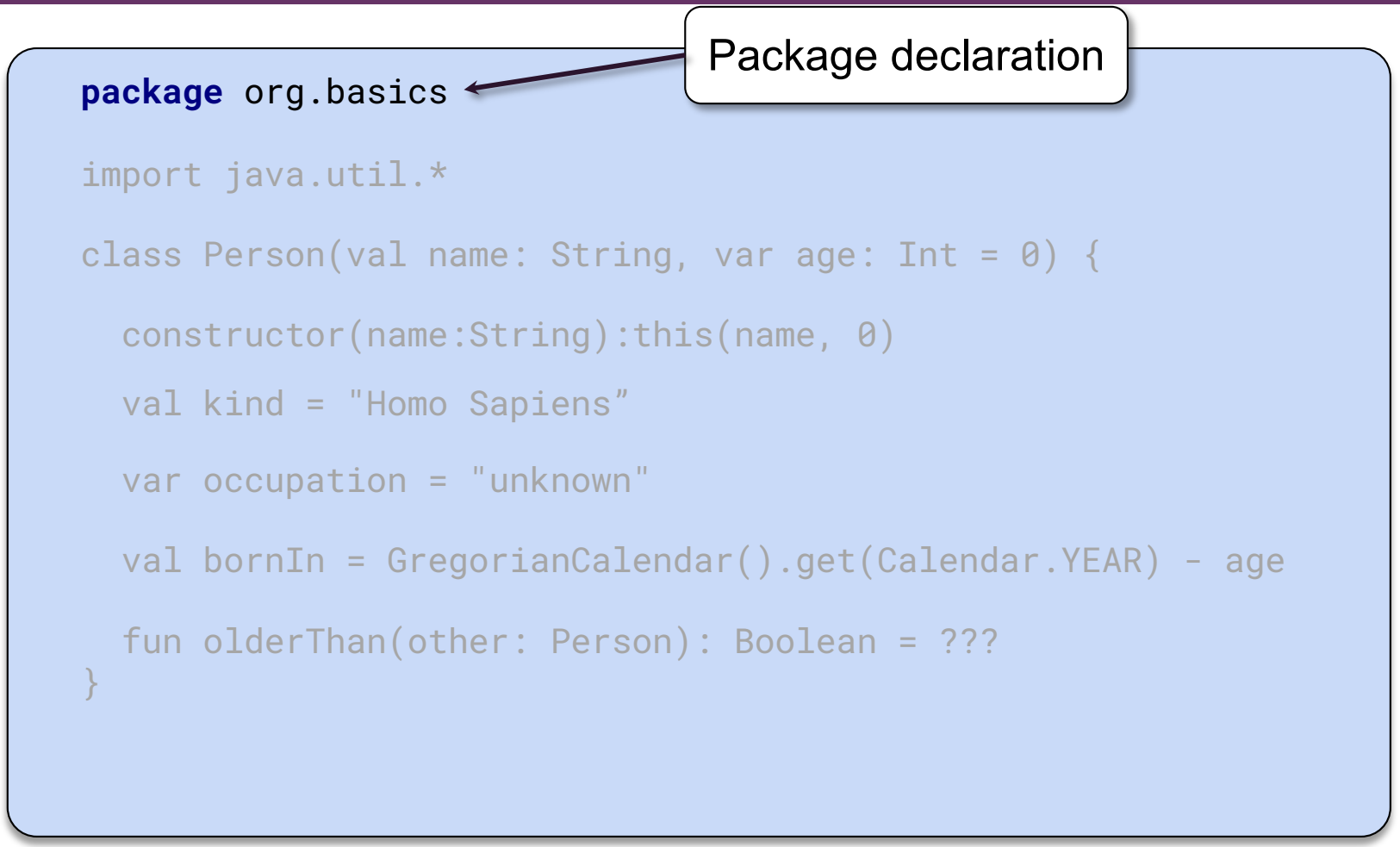


If expression

```
package org.basics

import java.util.*

class Person(val name: String, var age: Int = 0) {

  constructor(name: St

  val kind = "Homo Sap

  var occupation = "un

  val bornIn = GregorianCalendar().get(Calendar.YEAR) - age

  fun olderThan(other: Person): Boolean {
    return if(age > other.age) true else false
}
```

If is an *expression*.
Therefore, it returns a value.
The ternary operator (? :)
does not exist in Kotlin

Package declaration

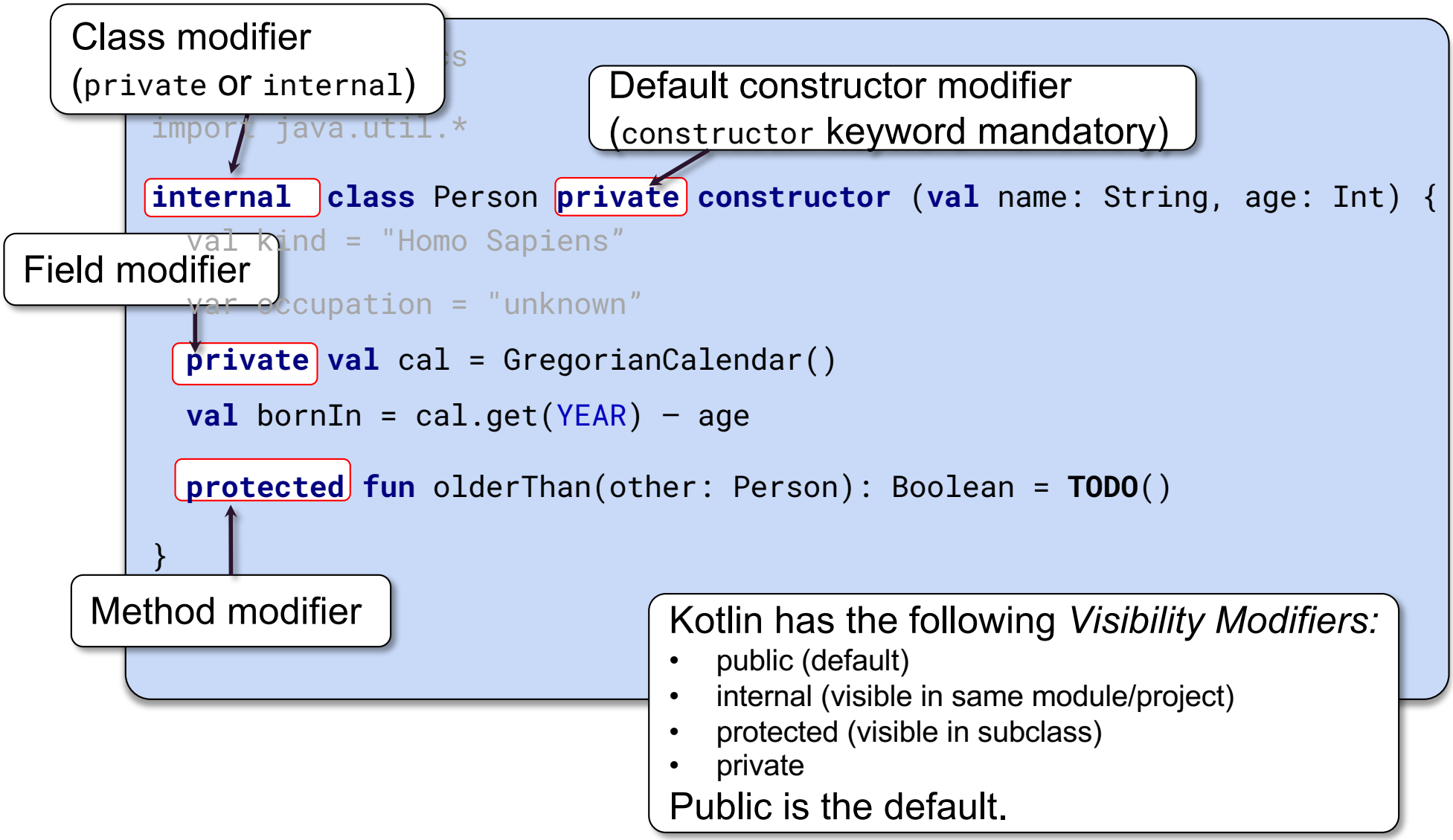


Imports Statement: disambiguate keyword

```
package org.basics

import java.util.GregorianCalendar as GregCal

class Person(val name: String, var age: Int = 0) {

  val kind = "Homo Sapiens"

  var occupation = "unknown"

  val bornIn = GregCal().get(Calendar.YEAR) - age

}
```

Use ‘as’ to rename the import locally

Visibility Modifiers



Kotlin has the following *Visibility Modifiers:*

- public (default)
- internal (visible in same module/project)
- protected (visible in subclass)
- private

Public is the default.

What is this?

```
package org.basics

import java.util.*

class Person(val name: String, age: Int) {
  val kind = "Homo Sapiens"

  var occupation = "unknown"

  private val cal = GregorianCalendar()

  val bornIn = this.cal.get(YEAR) - age

  fun olderThan(other: Person): Boolean = TODO()

}
```

With the `this` keyword we can refer to a class' *instance variables* or *methods*. `this` is implicit, so most of the time it is not required.

Objects and static members

Member functions are member of the **object**

```
CREATE OR REPLACE TYPE PERSON AS OBJECT
(name VARCHAR2(30),
 age NUMBER,
 kind VARCHAR2(30),
 MEMBER FUNCTION get_name RETURN VARCHAR2,
 MEMBER PROCEDURE set_name (pv_name VARCHAR2),
 STATIC FUNCTION new_born (pv_name VARCHAR2)
) INSTANTIABLE NOT FINAL;
/
```

```
DECLARE
 person_1 PERSON;
BEGIN
 person_1 := PERSON("Albert", 36);
 name_1 := person_1.get_name;

 person_2 := PERSON.new_born("Heijn");
END;
/
```

Static functions are member of the **object type**

Objects

```
class Person(val name: String, var age: Int = 0) { }
```

An object represents a singleton instance. It can have *fields* and *methods* like a class.

```
object Anonymizer {
  val anonymousName = "*******"
  fun anonymous(p:Person) = Person(anonymousName, p.age)
}
```

Fields and methods are accessed using the *name* of the *object*.

```
Anonymizer.anonymous(Person("Jack", 37))
```

Companion Objects

```
class Person(val name: String, var age: Int = 0) {
  private val cal = GregorianCalendar()

  companion object {
    fun newBorn(name: String):Person {
      val newBorn = Person(name)
      println("Born @ " + newBorn.cal)
      return newBorn
    }
  }
}
```

Inside a class an *object* declaration can be prefixed with the *companion* keyword.

Companion objects can be used as *factories* for their companion classes or as placeholder of *static members* like a logger.

They can access private fields of their *companion class*.

Companion fields and methods are accessed using the *name* of the *class.*
(object name == class name)

```
Person.newBorn("Jack")
```

Toplevel Functions and Properties

HelloKotlin.kt

```
package org.basics
import java.time.*

val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")

fun today() = LocalDateTime.now().format(formatter)


fun main(args: Array<String>) {
        println("you executed me at:" + today())
}
```

Functions and properties can be declared '*top-level*' (outside of a class or object)

Main methods also need to be declared top level.

The main *class* has the following name: *<package>.<filename>Kt*

```
java org.basics.HelloKotlinKt
```

Exercise: Implement a Class

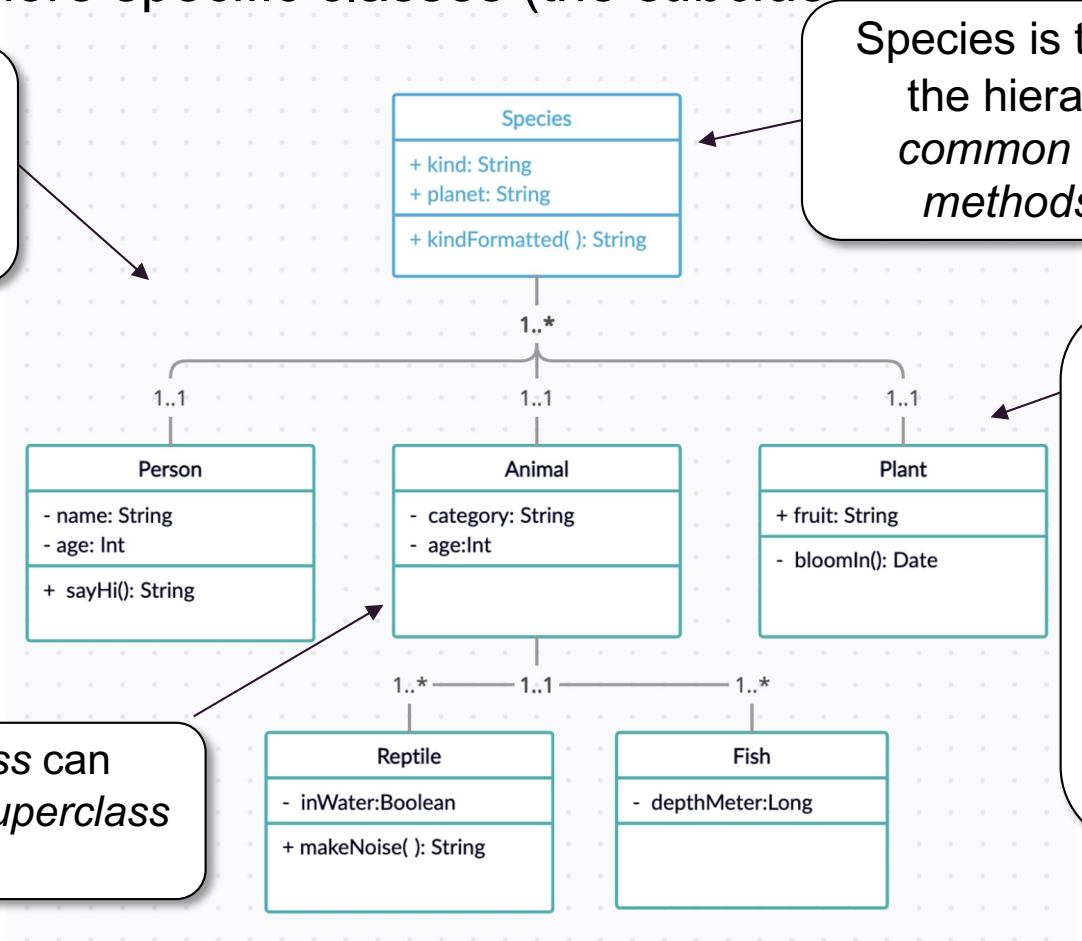
- Download Slides: https://bit.ly/kotlin-training-ah-plsql-slides-2023
- Open: OOExercise01.kt and OOExercise01Test
- **Solve Exercise: 1**
  - Create a `class Euro`
  - Provide it with two *immutable* constructor/field parameters:
    `euro:Int, cents:Int`
  - Provide the `cents` field with default value: `0`
  - Provide an immutable field named: `inCents` that converts euro + cents into cents.
  - Add a `companion object` to `Euro` with a factory method named: `fromCents` that creates an `Euro` based on cents.
    - E.g. calling `fromCent(150)` must result in: `Euro(1,50)`
  - Create a method named: `plus` that adds another `Euro`
  - Create a method named: `times` that multiplies an `Euro`
  - Make sure all tests succeed

Object Orientation Basics - Inheritance

A powerful feature of object orientation is *inheritance.*
Inheritance allows you to put *common code* in one class (the *superclass*), and then allow other more specific classes (the *subclass*) to inherit this code.

The hierarchical structure we see here is called an *inheritance tree*

Species is the *base class* of the hierarchy where all *common properties* and *methods* are defined.



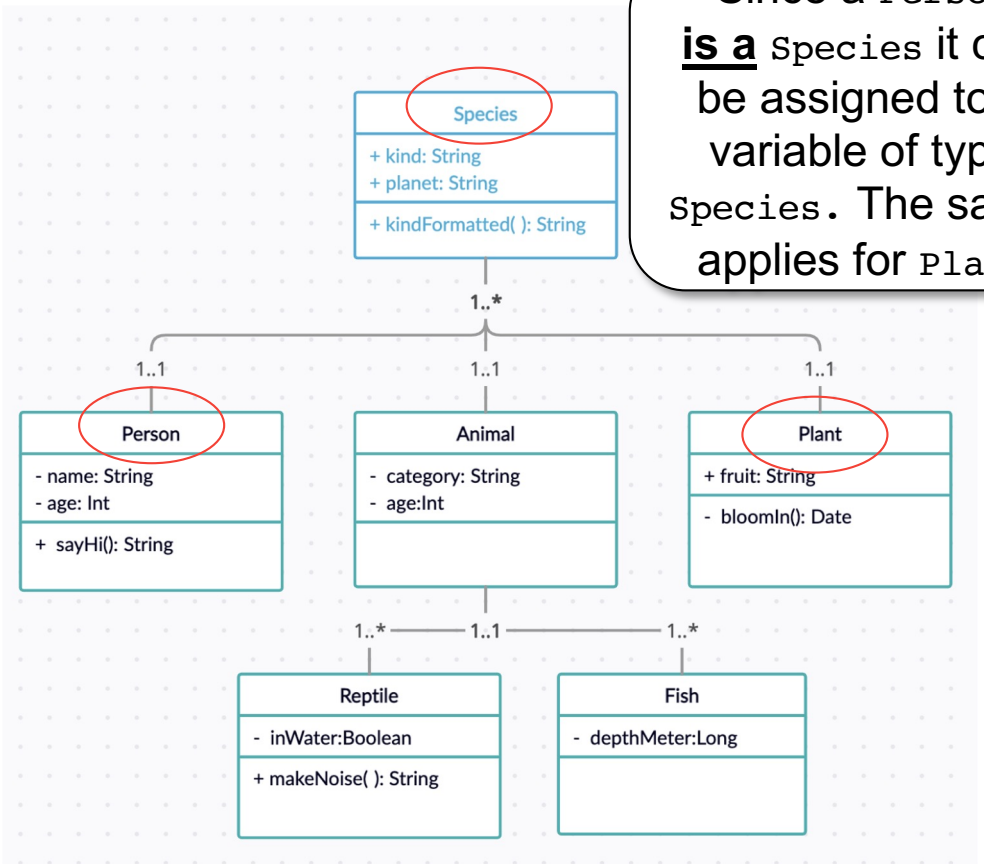
These are the *subclasses* that *extend* the *superclass* `Species`. *Subclasses inherit* the *properties* and methods from its *superclass* and allow new ones to be added.

A *subclass* can become a *superclass* too

A key advantage of inheritance is that if you have to modify code you have to do it in only one place - the *superclass*. *Subclasses* automatically inherit this code.

Inheritance & Polymorphism

Another advantage of inheritance is Polymorphism: Polymorphism allows you to deal with *different kinds* of *subclasses* in a *common way*.



Since a `Person` **is a** `Species` it can be assigned to a variable of type `Species`. The same applies for `Plant`

```
val personSue:Species =  Person(
        kind = "Homo Sapiens",
        planet = "Earth",
        name =  "Sue",
        age = 32)
```

```
val appleTree:Species =  Plant(
        kind = "Malus Domestica",
        planet = "Earth",
        fruit =  "Apple")
```

```
val species:Array<Species> =
     arrayOf(personSue, appleTree)

species[0].kind //Homo Sapiens
species[1].kind //Malus Domestica
```

This is *polymorphism* in action: since `Person` and `Plant` share the properties and/or methods from their *superclass* `Species`, they both can be addressed from the perspective of `Species`

Inheritance comes in many shapes

There are different ways to apply Inheritance:

- Extend from a normal class
- Extend from an abstract class
- Inheritance with interfaces

The invisible Inheritance

```
open class Any{
    open fun equals(other:Any?):Boolean = …
    open fun hashCode():Int = …
    open fun toString():String = …
}
```

Allows comparing objects. The default is by reference (their memory address). Alias: `==`

Identity of an object used in hash tables

Spring representation of the object

So automatically every Kotlin class inherits `equals(...)`, `hashCode()` and the `toString()` method from `Any`

Every class in Kotlin *implicitly* extends from `Any`

```
class Person(val name: String, val age: Int):Any
```

```
val jack = Person("Jack", 42)
val joe = Person("Joe", 24)
jack == joe                 //false
jack == jack                //true
jack.hashCode()             //501263526
jack.toString()             //org.course.Person@1de0aca6
println(jack)               //org.course.Person@1de0aca6
```

Whenever a *String* representation is required, `toString()` is called

Inheritance

By default classes are *final*. To allow others to inherit from a class it must me marked *open*

```
open class Species(val kind:String) {
    open fun kindFormatted() = kind.uppercase()

    open val planet = "Earth"
}
```

Fields and methods are *final* by default too. To make them overridable they must be marked *open* as well.

To extend from a class, use the syntax *MyClass(...):SuperClass*

When extending a class the superclass' constructor must always be called

```
class Person(val name: String, val age: Int):Species("Alien"){

    override fun kindFormatted() = super.kindFormatted() + "!"

    override val planet = "Mars"
}
```

Use *super* to call the superclass' implementation

When overriding a field or a method *override* is *mandatory*

Abstract Classes

Abstract class are classes that cannot be instantiated, only their implementing subclasses can. They are ideal to define a *contract and/or common behavior*, which can be completed by their subclasses.

Use the `abstract` keyword to mark a class *abstract*. An abstract class is `open` by default

```
abstract class Species(val kind:String) {
    abstract fun kindFormatted():String
    abstract val planet:String
}
```

The `abstract` keyword is needed to mark a field (`val`/`var`) or a method abstract.

For abstract field or method implementations `override` is *mandatory*

```
open class Person(val name: String, val age: Int):Species("Homo Sapiens"){
    override fun kindFormatted() = kind.toLowerCase()
    final override val planet = "Earth"
}
```

If the overridden class is `open`, *overridden* fields or methods are *open* too. To prevent further overriding they must be marked `final`

When to use abstract classes

```
abstract class Species(val kind:String) {
    abstract fun kindFormatted():String

    abstract val planet:String
}
```

What do you think: Is `Species` a good candidate for an `abstract` class or should it be an `open` class?

- When the class *should not be instantiable*
- When the class contains *common behavior* and *state* that is to be shared for different implementations (subclasses)
  (state = `val`'s and/or `var`'s with pre-configured values)

data classes: The ideal value object

Classes or Objects prefixed with `data` gain special abilities

```
data class Person(val name:String, var age:Int) { ... }
```

data classes: The ideal value object

```
data class Person(val name:String, var age:Int) { ... }
```

A data class automatically gets a natural implementation of `toString`, `equals` and `hashCode`,

… and a `copy` method to create copies based on the original object

```
val jack = Person("Jack", 23)
jack: Person(name=Jack,age=23)

jack == Person("Jack", 23)
res1: true

val fred = jack.copy(name = "Fred")
```

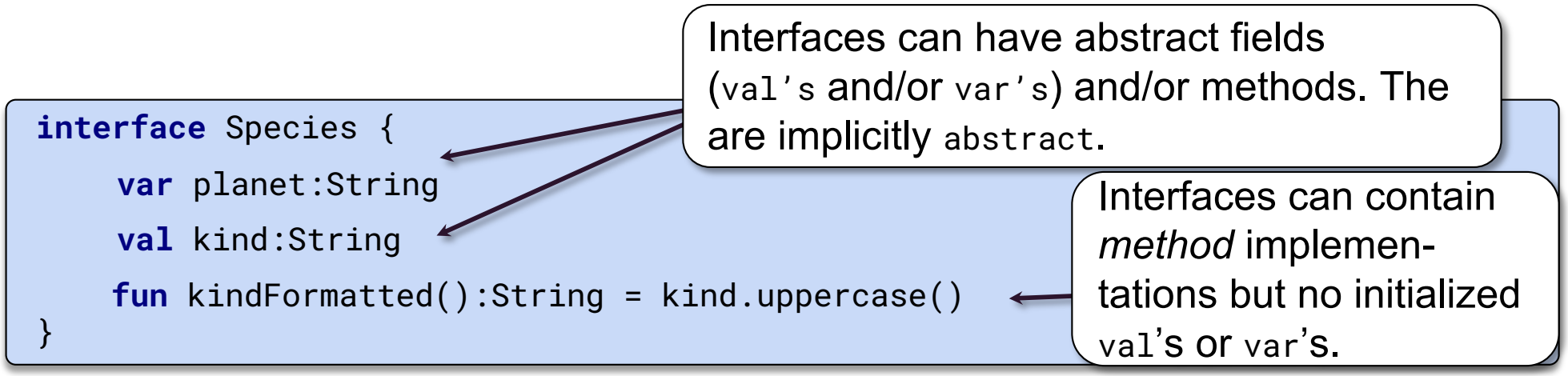
When to use data classes

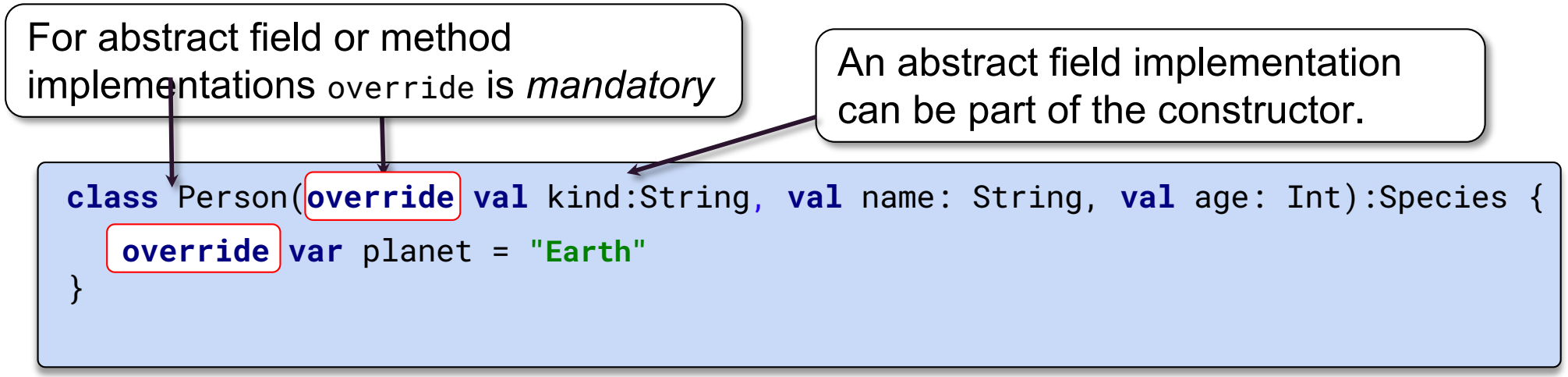
```
data class Person(val name:String, var age:Int) { ... }
```

- data classes should be used as *immutable value objects*
- They are not intended to be used as *service objects*
- They cannot inherit from other data classes

Interfaces

Interfaces are similar to a PL/SQL *specification*. Their main purpose is to define a *contract* that classes / (bodies in PL/SQL) have to implement.





When to use interfaces

```
interface Species {
    var planet:String
    val kind:String
    fun kindFormatted():String = kind.uppercase()
}
```

- When it *should not be instantiable*
- To define a *contract of behavior* that *does not have state*
- When *multiple-inheritance* is required
  (-> one class implements multiple interfaces)

Enums

Use the **enum** keyword to define an enumerated class

```
enum class SimpleKind {
    ANIMAL,HUMAN,ALIEN;
}
```

Declare the Enum constants

```
SimpleKind.ALIEN.name
//-> "ALIEN"
SimpleKind.ALIEN.ordinal
//-> 2
SimpleKind.values()
//-> [ANIMAL, HUMAN, ALIEN]

SimpleKind.valueOf("ALIEN")
//-> SimpleKind.ALIEN
```

Name of Enum constant as String

Position in Enum declaration, 0 based

Get all Enum constants

Get Enum constant by name. Throw `IllegalArgumentException` if name does not match an Enum.

Enums Advanced

```
enum class AdvancedKind(val taxonomy:String, var birthYear:Long) {
    ANIMAL("Animalia", -640000000),
    HUMAN("Homo sapiens", -194000),
    ALIEN("Genus Hammerhead", 2020);
    fun asString() = "$this ($taxonomy) since $birthYear"
}
```

enum 's can have fields and methods

```
enum class EvolutionaryKind {
    ANIMAL {
        override fun evolvesTo() = HUMAN
    },
    HUMAN{
        override fun evolvesTo() = ALIEN
    },
    ALIEN{
        override fun evolvesTo() = ANIMAL
    };
    abstract fun evolvesTo():EvolutionaryKind
}
```

An Enum can also declare *abstract methods* that need to be implemented in the Enum constants' own anonymous class

When to use enums

```
enum class SimpleKind {
   ANIMAL,HUMAN,ALIEN;
}
```

When we want to represent a *finite set of options*

String Interpolation

```
class Person(val name: String, var age: Int = 0) {

  val asStr1 = "$name is $age years old"

  val asStr2 = "$name is ${if (age == 0) "a newborn" else "$age years old"}"

}
```

String Interpolation

Use the **$** to directly reference a variable in the surrounding scope

```
class Person(val name: String, var age: Int = 0) {

  val asStr1 = "$name is $age years old"

  val asStr2 = "$name is ${if (age == 0) "a newborn" else "$age years old"}"

}
```

String Interpolation

```
class Person(val name: String, var age: Int = 0) {

  val asStr1 = "$name is $age years old"

  val asStr2 = "$name is ${if (age == 0) "a newborn" else "$age years old"}"

}
```

Interpolators accept arbitrary expression between ${}

Exercise: Implement a Class

- Download Slides: https://bit.ly/kotlin-training-ah-plsql-slides-2023
- Open: OOExercise01.kt and OOExercise01Test
- **Solve Exercise: 1**
  - Create a `class Euro`
  - Provide it with two *immutable* constructor/field parameters:
    `euro:Int, cents:Int`
  - Provide the `cents` field with default value: `0`
  - Provide an immutable field named: `inCents` that converts euro + cents into cents.
  - Add a `companion object` to `Euro` with a factory method named: `fromCents` that creates an `Euro` based on cents.
    - E.g. calling `fromCent(150)` must result in: `Euro(1,50)`
  - Create a method named: `plus` that adds another `Euro`
  - Create a method named: `times` that multiplies an `Euro`
  - Make sure all tests succeed

Object Orientation Basics

Object types and their members are FINAL by default. By marking them NOT FINAL a subtype can override them.

```
CREATE OR REPLACE TYPE SPECIES AS OBJECT
(kind VARCHAR2(30),
 MEMBER FUNCTION get_kind RETURN VARCHAR2,
 NOT FINAL MEMBER FUNCTION kind_formatted RETURN NUMBER
) NOT FINAL
/
```

Special keywords for inheritance

```
CREATE OR REPLACE TYPE PERSON UNDER SPECIES
(name VARCHAR2(30),
 age NUMBER,
 MEMBER FUNCTION get_name RETURN VARCHAR2,
 MEMBER PROCEDURE set_name (pv_name VARCHAR2),
 OVERRIDING MEMBER FUNCTION kind_formatted
)
/
```

Inheritance

By default classes are *final*. To allow others to inherit from a class it must me marked *open*

```
open class Species(val kind:String) {
    open fun kindFormatted() = kind.uppercase()

    open val planet = "Earth"
}
```

Fields and methods are *final* by default too. To make them overridable they must be marked *open* as well.

When extending a class the superclass' constructor must always be called

```
class Person(val name: String, val age: Int):Species("Alien"){

   override fun kindFormatted() = super.kindFormatted() + "!"

   override val planet = "Mars"

}
```

Use *super* to call the superclass' implementation

When overriding a field or a method *override* is *mandatory*

Abstract Classes

```
CREATE OR REPLACE TYPE SPECIES AS OBJECT
(kind VARCHAR2(30),
 MEMBER FUNCTION get_kind RETURN VARCHAR2,
 NOT INSTANTIABLE NOT FINAL MEMBER FUNCTION kind_formatted RETURN NUMBER
) NOT INSTANTIABLE NOT FINAL
/
```

In PL/SQL `NOT INSTANTIABLE` implies that only a subtype can provide an implementation. In OO terms, this is called an `abstract class`.

Abstract Classes

Use the `abstract` keyword to mark a class *abstract*. An abstract class is `open` by default

```
abstract class Species(val kind:String) {
    abstract fun kindFormatted():String
    abstract val planet:String
}
```

The `abstract` keyword is needed to mark a field (`val`/`var`) or a method abstract.

For abstract field or method implementations `override` is *mandatory*

```
open class Person(val name: String, val age: Int):Species("Homo Sapiens"){
    override fun kindFormatted() = kind.toLowerCase()
    final override val planet = "Earth"
}
```

If the overridden class is `open`, *overridden* fields or methods are *open* too. To prevent further overriding they must be marked `final`

data classes: The ideal value object

Classes or Objects prefixed with `data` gain special abilities

```
data class Person(val name:String, var age:Int) { ... }
```

data classes: The ideal value object

```
data class Person(val name:String, var age:Int) { ... }
```

A data class automatically gets a natural implementation of `toString`, `equals` and `hashCode`,

… and a `copy` method to create copies based on the original object

as well as `componentN(...)` methods for *destructuring* all fields declared in the constructor

```
val jack = Person("Jack", 23)
jack: Person(name=Jack,age=23)

jack == Person("Jack", 23)
res1: true

val fred = jack.copy(name = "Fred")

val (name, age) = fred
name: "Fred"
age: 23
```

When to use data classes

```
data class Person(val name:String, var age:Int) { ... }
```

- data classes should be used as *immutable value objects*
- They are not intended to be used as *service objects*
- They cannot inherit from other data classes

Interfaces

Interfaces can have abstract fields (`val`'s and/or `var`'s) and/or methods. The are implicitly `abstract`.

Interfaces can contain *method* implementations but no initialized `val`'s or `var`'s.

```
interface Species {
    var planet:String
    val kind:String
    fun kindFormatted():String = kind.uppercase()
}
```

For abstract field or method implementations `override` is *mandatory*

An abstract field implementation can be part of the constructor.

```
class Person(override val kind:String, val name: String, val age: Int):Species {
    override var planet = "Earth"
}
```

Enums

Use the **enum** keyword to define an enumerated class

```
enum class SimpleKind {
    ANIMAL,HUMAN,ALIEN;
}
```

Declare the Enum constants

```
SimpleKind.ALIEN.name
//-> "ALIEN"
SimpleKind.ALIEN.ordinal
//-> 2
SimpleKind.values()
//-> [ANIMAL, HUMAN, ALIEN]

SimpleKind.valueOf("ALIEN")
//-> SimpleKind.ALIEN
```

Name of Enum constant as String

Position in Enum declaration, 0 based

Get all Enum constants

Get Enum constant by name. Throw `IllegalArgumentException` if name does not match an Enum.

Enums Advanced

```
enum class AdvancedKind(val taxonomy:String, var birthYear:Long) {
    ANIMAL("Animalia", -640000000),
    HUMAN("Homo sapiens", -194000),
    ALIEN("Genus Hammerhead", 2020);
    fun asString() = "$this ($taxonomy) since $birthYear"
}
```

enum 's can have fields and methods

```
enum class EvolutionaryKind {
    ANIMAL {
        override fun evolvesTo() = HUMAN
    },
    HUMAN{
        override fun evolvesTo() = ALIEN
    },
    ALIEN{
        override fun evolvesTo() = ANIMAL
    };
    abstract fun evolvesTo():EvolutionaryKind
}
```

An Enum can also declare *abstract methods* that need to be implemented in the Enum constants' own anonymous class

String Interpolation

```
class Person(val name: String, var age: Int = 0) {

  val asStr1 = "$name is $age years old"

  val asStr2 = "$name is ${if (age == 0) "a newborn" else "$age years old"}"

}
```

String Interpolation

Use the **$** to directly reference a variable in the surrounding scope

```
class Person(val name: String, var age: Int = 0) {

  val asStr1 = "$name is $age years old"

  val asStr2 = "$name is ${if (age == 0) "a newborn" else "$age years old"}"

}
```

String Interpolation

```
class Person(val name: String, var age: Int = 0) {

  val asStr1 = "$name is $age years old"

  val asStr2 = "$name is ${if (age == 0) "a newborn" else "$age years old"}"

}
```

Interpolators accept arbitrary expression between ${}

Exercise: Apply Inheritance & String Interpolation

- Open: OOExercise02.kt and OOExercise02Test
- **Solve Exercise: 2**
  - Create an `abstract class Currency`
  - Provide it with one immutable constructor/field:

    `symbol:Symbol`
  - Symbol is an Enum with constants for EUR and USD and an immutable field:

    `sign:String`
  - Extend the previously created **`Euro`** class from **`Currency`**
  - Override the **`toString`** method of **`Euro`** to represent the following String by making use of *String interpolation*:
    - symbol.sign + ': ' + euro + '.' + cents. E.g: €: 200.05
      To format the cents use: `String.format("%02d", cents)`
  - In case the cents are 0 use the following representation:
    - symbol.sign + ': ' + euro + '.--. E.g.: €: 200.--

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

Kotlin test Frameworks

- **Unittest frameworks**
  - JUnit4/5
    - Works as is
  - Kotest - framework + Assertion DSL
    - https://github.com/kotest/kotest (multiplatform support)
  - Spek - framework
    - https://spekframework.org/ (multiplatform support)
- **Test Utils**
  - Assertion DSL:
    - Kotest
    - Strikt: https://strikt.io/
  - Mocking libraries
    - Mockk https://mockk.io/
    - Mockito-Kotlin: https://github.com/mockito/mockito-kotlin

Testing with Kotest

```
import io.kotest.assertions.throwables.shouldThrow
import io.kotest.core.spec.style.WordSpec
import io.kotest.matchers.*
import io.kotest.matchers.collections.*
import io.kotest.matchers.string.*
import io.kotest.matchers.comparables.beLessThan

class KotestSampleTest : WordSpec() {
    init {
        var state = 0
        "Kotest" should {
            "have a great variety of matchers" {
                state = 2
                state shouldBe 2

                9 should beLessThan(10)
                shouldThrow<NumberFormatException> { "NaN".toInt() }

                "Kotlin" should startWith("K")
                "12:30" should match("""\d{2}:\d{2}""")

                listOf(1, 2) should containExactlyInAnyOrder(listOf(2, 1))
                listOf(1, 2, 3) should beSorted()
            }

            "!correctly handle state" {
                state shouldBe 2
            }
        }
    }
}
```

Testing with Kotest

```
import io.kotest.assertions.throwables.shouldThrow
import io.kotest.core.spec.style.WordSpec

class KotestSampleTest : WordSpec() {
    init {
        var state = 0
        "Kotest" should {
            "have a great variety of matchers" {
                state = 2
                state shouldBe 2
                state should beLessThan(10)
                shouldThrow<NumberFormatException> { "NaN".toInt() }

                "Kotlin" should startWith("K")
                "12:30" should match("""\d{2}:\d{2}""")

                listOf(1, 2, 3) should containExactlyInAnyOrder(listOf(2, 3, 1))
                listOf(1, 2, 3) should beSorted()
            }

            "!correctly handle state" {
                state shouldBe 2
            }
        }
    }
}
```

State for all tests is placed here (like JUnit setup)

Define the unit to test, like a class

Define the part of the unit to test

Use matchers for test assertions

Use ! to ignore a test

Testing with Kotest

```
import io.kotest.assertions.throwables.shouldThrow
import io.kotest.core.spec.style.WordSpec
import io.kotest.matchers.*
import io.kotest.matchers.collections.*
import io.kotest.matchers.string.*
import io.kotest.matchers.comparables.beLessThan

class KotestSampleTest : WordSpec() {
    init {
        var state = 0
        "Kotest" should {
            "have a great variety of matchers" {
                state = 2
                state shouldBe 2
                state should beLessThan(10)
                shouldThrow<NumberFormatException> { "NaN".toInt() }

                "Kotlin" should startWith("K")
                "12:30" should match("""\d{2}:\d{2}""")

                listOf(1, 2, 3) should containExactlyInAnyOrder(listOf(2, 3, 1))
                listOf(1, 2, 3) should beSorted<Int>()
            }

            "!correctly handle state" {
                state shouldBe 2
            }
        }
    }
}
```

Kotest offers a great variety of *Matchers* to do all kinds of sophisticated test assertions

All possible test styles and matchers can be found here

Exercise: Write a unittest with kotest

- Open: Euro in OOExercise02.kt and KotestExerciseTest
- **Instructions**
  - 1. Implement the `divide` method in `Euro` that has the following signature: `fun div(divider:Int):Euro`
    - If the divider is <=0 throw an `IllegalArgumentException`
  - 2. Write a Kotest covering the following testcases:
    - Happy flow (divider is > 0)
    - Alternative flow (divider is <= 0)

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

The Root Cause of Defensive Programming

The root cause of defensive programming is a *bad contract*

In order to be really safe you have to be 'defensive', always checking for NULL

Is there no better way??

```
DECLARE
 TYPE person IS RECORD
  (name VARCHAR2(30) DEFAULT NULL,
   age NUMBER DEFAULT NULL,
   occupation VARCHAR2(30) DEFAULT NULL);
 person_1 PERSON;
 age_is_null EXCEPTION;
START
 <... some operations here ...>
 assert.is_not_null(person_1.name, 'Name cannot be NULL');
 IF person_1.age IS NULL THEN
  RAISE age_is_null;
 END IF;
 IF person_1.occupation IS NOT NULL THEN
  DBMS_OUTPUT.put_line('Occupation is ' || person_1.occupation)
 ELSE
  DBMS_OUTPUT.put_line('Occupation unknown');
 END IF;
EXCEPTION
 WHEN age_is_null THEN
  DBMS_OUTPUT.put_line(-20000, 'Age cannot be NULL');
END;
/
```

Nullable Types to the rescue

```
class ProductService {
  fun maxRating(productName:String):Int? =
    if(found) rating else null
}
```

Kotlin

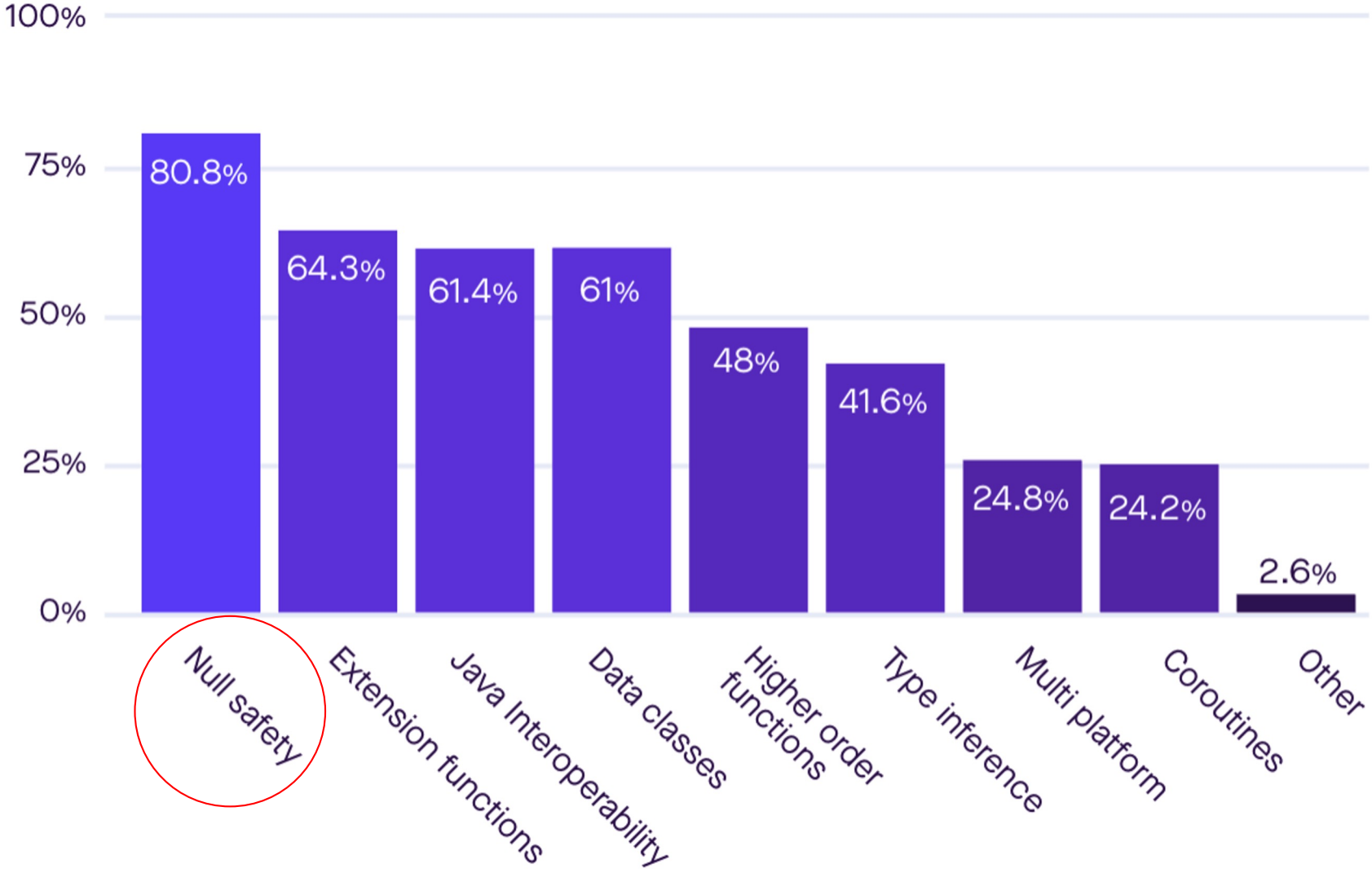
Nullable types make the API contract explicit: *you must handle the case when there is no return value*

```
class Client {
   fun showMaxRating(pName:String):Int =
       productService.maxRating(pName) ?: 0
 }
```

Kotlin

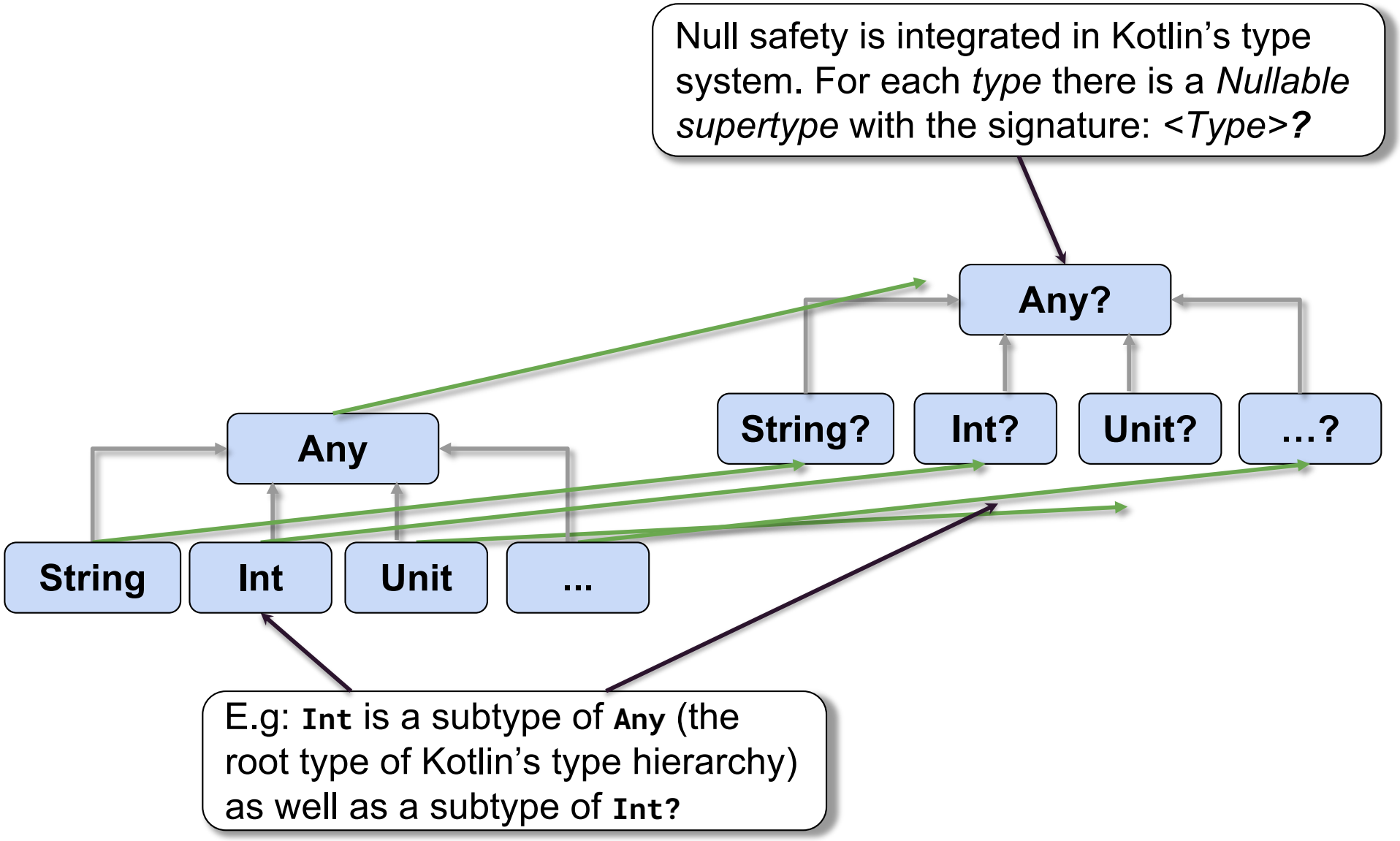
Nullable Types make a real difference

Kotlin’s favourite features



Source: https://pusher.com/state-of-kotlin

Nullable Types in Kotlin's Type System



Working with Nullable Types

Declare a nullable type by appending a `?` to the actual type.

```
var hobby:String? = null
hobby = "programming"
```

A nullable type can have **`null`** or an instance of its own type assigned

When checking a *nullable type* in an **`if`** expression it can be accessed like a *non-null* variable inside **`if`**

```
val spareTimeActivity = if(hobby != null) hobby else "lazy"

val spareTimeActivity = hobby ?: "lazy"
```

Instead of an **`if`** expression the *Elvis operator* **`?:`** can be used to provide a default for the null case

```
val noChoice = hobby!!.uppercase()
```

A value can be forced out of a nullable type using **`!!`**. If the variable was **`null`**, a **`kotlin.KotlinNullPointerException`** is thrown.

Nullable Types in Nested Structures

```
class Hobby(val desc:String, var hoursPerWeek:Int? = null)
val myHobby:Hobby? = Hobby("Programming", 60)
```

`?` can also be *chained* to safely access a property in a nested data structure

```
val hobbyTime = myHobby?.hoursPerWeek?.toString() ?: "UNKNOWN"
```

A safe call can also be placed on the assignment side: the variable will only be assigned if all properties in the nested structure are not null.

```
myHobby?.hoursPerWeek = 40
```

Nullable types in API's

Safely get the first element of a List

```
val res1:Int? = emptyList<Int>().firstOrNull()
-> r1: null
```

Remove null values from a List

```
val res2:List<Int> = listOf(1,2,null,4).filterNotNull()
-> r2: [1,2,4]
```

Transform a null variable of a List to an empty List

```
val noList:List<Int>? = null
val res3:List<Int> = noList.orEmpty()
-> res3: []
```

Same for String

```
val noString:String? = null
val res4:String = noString.orEmpty()
-> res4: ""
```

Safely get a value from a Map

```
val res5:String? = mapOf(7 to "eleven").get(7)
-> res5: "eleven"
```

Exercise: Nullable types

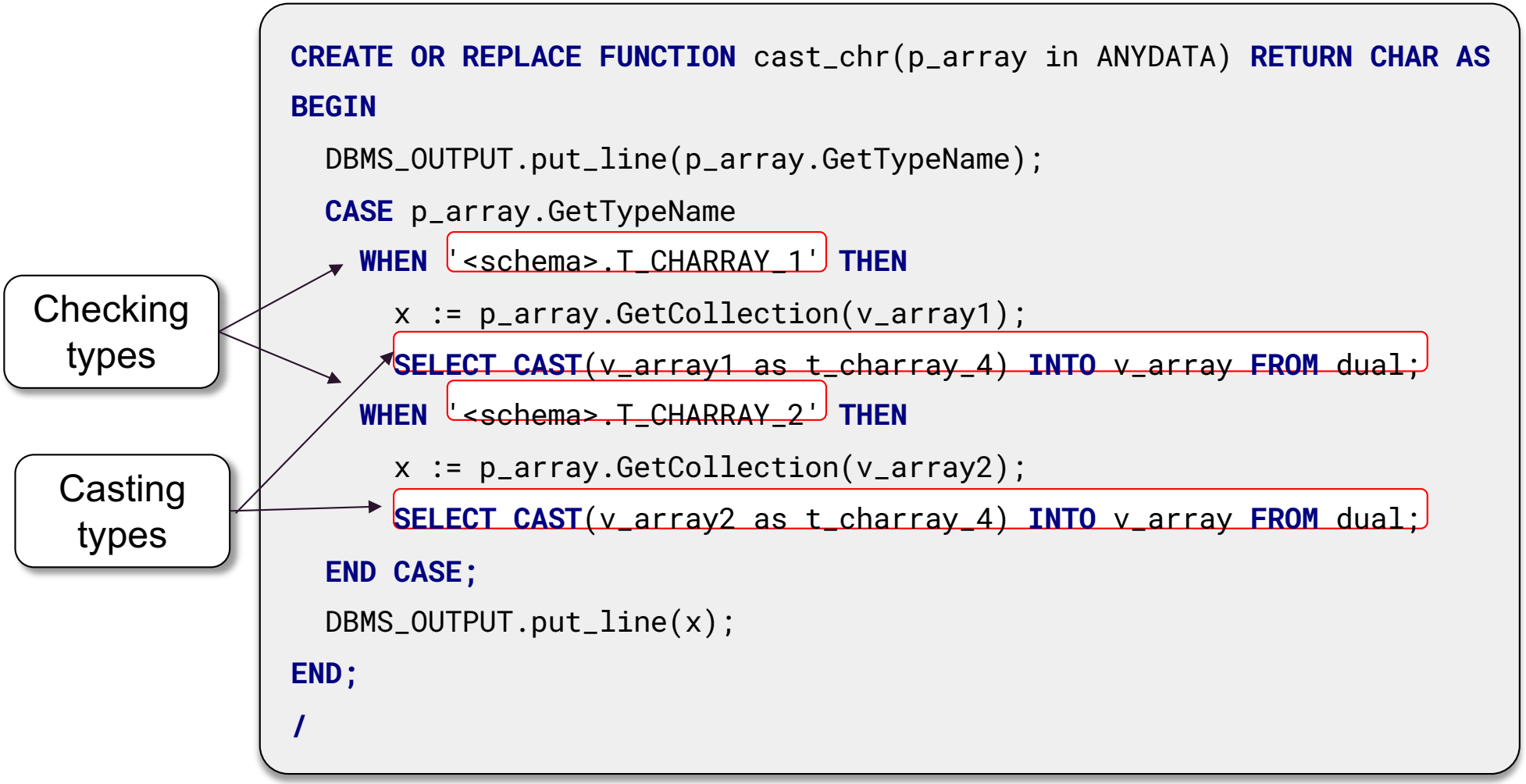
- **Before you start**
  - Take a look at the **SQL file**: **challenge04.sql**
  - The method implementation of `confirmationMessage` makes heavy use of null checks since we have no guarantee whether a value is null or not in PL/SQL.

- Open: NullSafetyExercise and NullSafetyExerciseTest
- **Exercise: port from PL/SQL to Kotlin with Null-Safety**
  - The aim of this exercise is to re-implement the logic of the PL/SQL method `confirmationMessage(...)` in Kotlin by making use of Kotlin's *Null Safety* features.
  - In the same class you find the domain classes that are already ported to Kotlin and provided with nullable types where applicable. Take a look at them.
  - Because PL/SQL does not have built-in nullability, the PL/SQL code is stuffed with null checks.
  - By using Kotlin's null-safety features you are able to rewrite this PL/SQL code in Kotlin with a *single* `if` expression *that checks for null*.
  - How can you do that? (Hint: smart casting is your friend)
  - You completed this exercise when the corresponding unittest succeeds.

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

Checking and Casting Types



Checking and Casting Types

The `is` operator can used to check the given type at runtime

```
val aStr = "hi"
val isStr:Boolean = aStr is String
-> isStr: true
```

The `as` operator *unsafely* casts a variable to a given type.

```
val whoAmI:Any = "hi"
val aStr:String = whoAmI as String
-> aStr: "hi"
```

Safe Casts

The **as?** operator *safely* casts a variable to a given type if it matches, otherwise it returns null

```
val whoAmI:Any = "hi"
val aStr:String? = whoAmI as? String
-> aStr: "hi"
val anInt:Int? = whoAmI as? Int
-> anInt: null
```

`whoAmI` is auto-cast to `Int`

```
val whoAmI:Any = "hi"
if(whoAmI is Int) {
        println(whoAmI * whoAmI)
} else if(whoAmI is String) {
        println(whoAmI.reversed())
}
-> "ih"
```

`whoAmI` is auto-cast to `String`

The compiler supports *smart casts* when **is** is used in an **if** *expression*: The variable is automatically cast to the type used in the condition of the **if**.

Generics

The generics syntax is the same as in Java: `<GENERIC_TYPE>`

By default a generic type (here `T`) extends from `Any?` which is Kotlin's root type

```
class Container<T>(var value:T)
```

```
val withInt = Container(42)
withInt.value is Int //-> true

val withString = Container("hi")
withString.value is String //-> true

val noValue:Any? = null
val withNullableAny = Container(noValue)
withNullableAny.value is Any? //-> true
```

Generics: Constraints

Types can be constrained using the syntax: `<GENERIC_TYPE:SUPERTYPE>`

```
class Container<T:Number>(var value:T)
```

```
val withInt = Container(42.0)
withInt.value is Double  //-> true
val withString = Container("hi")
```

Compilation Error: *inferred type* `String` *is not a subtype of* `Number`

The constrained type can also be another generic type

```
class Container<T:Comparable<T>>(var value:T)
```

```
val withString = Container("hi")
withString.value is String  //-> true
withString.value is Comparable<String>  //-> true
val withURL = Container(URL("http://abc.com"))
```

Compilation Error

Generics: Star Projection

```
class Container<T>(var value:T)
```

```
val c:Container<Any> = Container("hi")
c is Container<String>
```

Due to type erasure, checking generic types (*other than the declared one)* causes a Compilation Error: *Cannot check for instance of erased type* `Container<String>`

```
val c:Container<Any> = Container("hi")
c is Container<*>
```

For a generic instance check to succeed the *star-projection* syntax: `MyType<*>` needs to be used

Generics: Reification

```
class Container<T>(var value:T)
```

Due to type erasure it is not possible to figure out the class of a *generic type*. Compilation Error: *Cannot use T as reified type parameter*

```
fun <T> classOfType(c:Container<T>):Class<T> = T::class.java
```

For this to work we first need to declare the method as `inline`, causing its *code* to be directly *inlined* rather than being compiled to a separate method

```
inline fun <reified T> classOfType(c:Container<T>):Class<T> = T::class.java
```

Second, we have to qualify the *type parameter* with the `reified` *modifier* making the compiler preserve the type information.

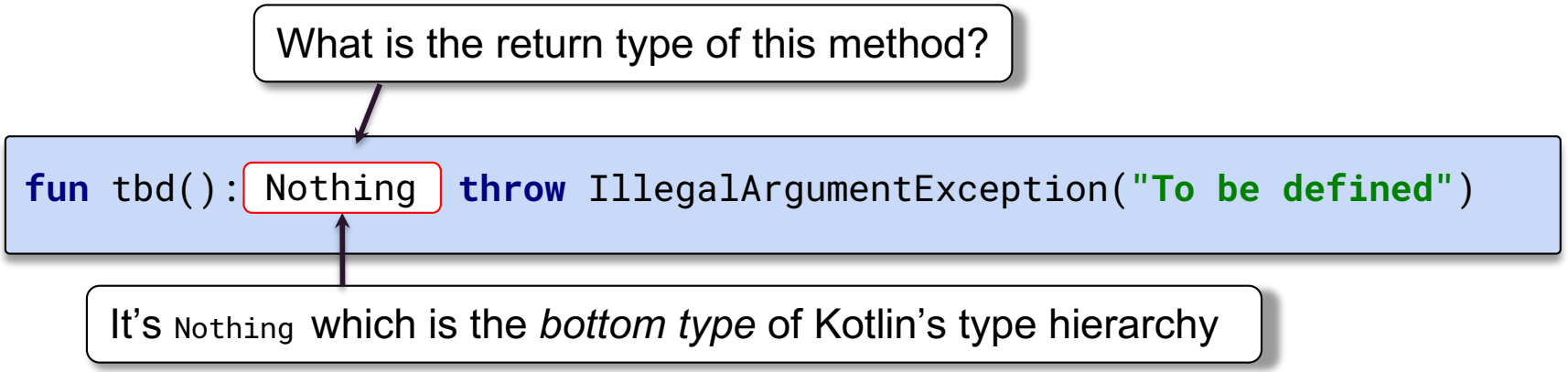
Generics: Methods

Generic methods are accessible using different data types.

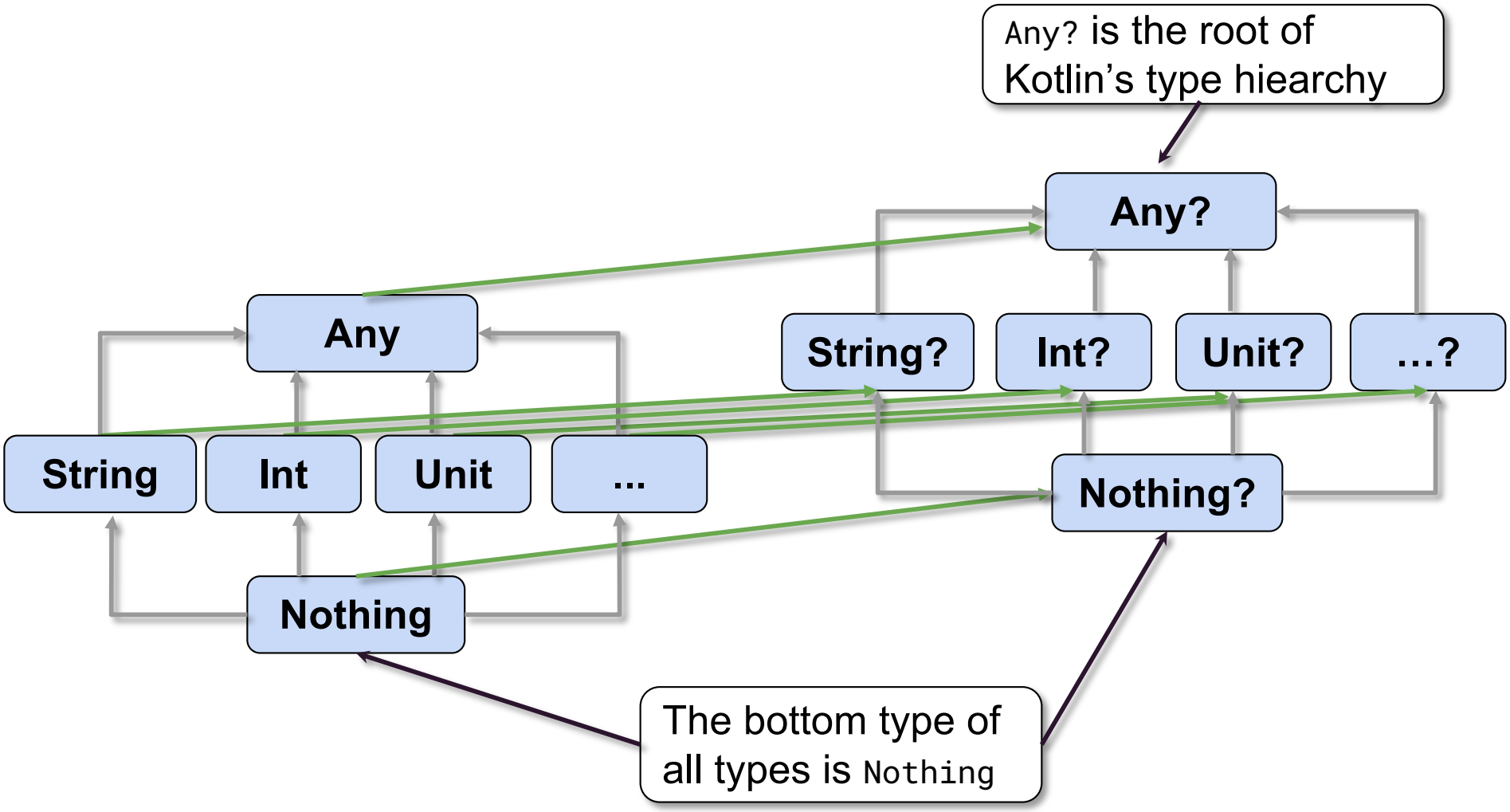
```
fun <T> addToList(item: T, list: List<T>): List<T>
```

```
val brands = listOf("Ahold", "Delhaize")
val moreBrands = addToList("Albert", brands)

val numbers = listOf(1, 2, 3)
val moreNumbers = addToList(4, numbers)
```

Nothing


What is the return type of this method?
fun tbd(): Nothing throw IllegalArgumentException("To be defined")
It's Nothing which is the bottom type of Kotlin's type hierarchy

Kotlin Type Hierarchy



Exercise: Generics

- Open: GenericsExercise.kt and GenericsExerciseTest
- **Exercise 1:**

  In *GenericsExercise.kt* familiarize yourself with the domain classes `Animal, Dog, Printer, AnimalPrinter`. Introduce *generics* in the `Printer` class so that test Exercise 1a in *GenericsExerciseTest* succeeds.

- **Exercise 2:**

  The extension method `ObjectMapper.readFromJsonString(json:String, clazz:Class<T>)` in *GenericsExercise.kt* uses the parameter clazz to deserialize a json string to a class. Rewrite this extension method by using a reified type, so that the clazz parameter can be removed and the method can be called with a type instead:
  `mapper.readFromJsonString<Dog>(json)`

  Uncomment test Exercise 5 in GenericsExerciseTest once you have provided the implementation.

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

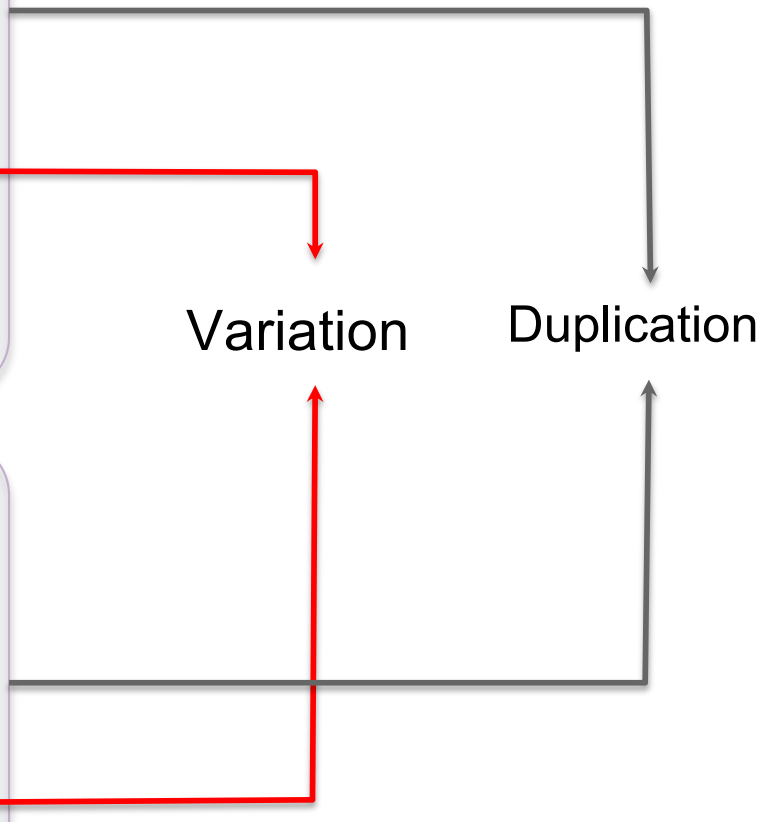
Fun

(-ctional)

Imperative programming in PL/SQL

```
DECLARE
  type NT_TYPE is table of NUMBER;
  nt NT_TYPE := NT_TYPE (1, 3, 5);
BEGIN
  FOR i IN 1..nt.count LOOP
       IF MOD(i) = 0 THEN

         dbms_output.put_line(nt(i));
    END IF;
  END LOOP;
END;
/
```

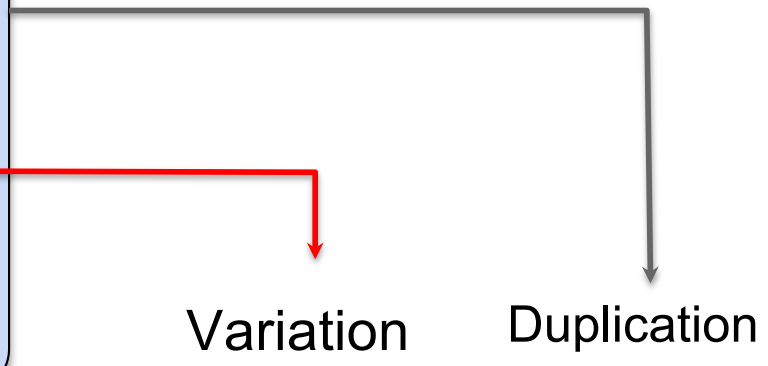
```
DECLARE
  type NT_TYPE is table of NUMBER;
  nt NT_TYPE := NT_TYPE (1, 3, 5);
BEGIN
  FOR i IN 1..nt.count LOOP
       IF MOD(i) = 1 THEN

         dbms_output.put_line(nt(i));
    END IF;
  END LOOP;
END;
/
```



Imperative programming in Kotlin

```
val res = mutableListOf<Int>()
for (i in res) {
    if ( i % 2 == 0 ) {
        res.add(i)
    }
}
return res
```

```
val res = mutableListOf<Int>()
for (i in res) {
    if ( i % 2 == 1 ) {
        res.add(i)
    }
}
return res
```



From Imperative to Functional in Kotlin

The Function: Varying computation

```
val res = mutableListOf<Int>()
for (i in res) {
    if ( i % 2 == 0 ) {
        res.add(i)
    }
}
return res
```

```
val isEvenNumber = fun(item: Int): Boolean {
    return item % 2 == 0
}
```

```
fun filter(items: List<Int>, predicate: (Int) -> (Boolean)): List<Int> {
    val res = mutableListOf<Int>()
    for (item in items) {
        if (predicate.invoke(item)) {
            res.add(item)
        }
    }
    return res
}
```

The Loop: Generic Control Structure

```
val listOfNumbers = listOf(1,2,3,4)

filter(listOfNumbers, isEvenNumber) // [2, 4]
```

Passing functions to other functions in Kotlin

```
filter(numbers, fun(item: Int): Boolean { return item % 2 == 0 })
```

Syntactic sugar

```
filter(numbers,{item:Int -> item % 2 == 0})
```

Syntactic sugar

```
filter(numbers) {item:Int -> item % 2 == 0}
```

With type inference

```
filter(numbers){item -> item % 2 == 0}
```

More sugar

```
filter(numbers){it % 2 == 0}
```

Functions as Objects

Kotlin Functions are objects.
Therefore, they can be assigned to variables and referenced like any other object.

```
//square has type: Int -> Int
val square = { i:Int -> i * i }
square(2)
 > 4

//multiply has type: (Int,Int) -> String
val multiply = {i:Int, j:Int -> (i * j).toString()}
multiply(2, 21)
 > "42"
```

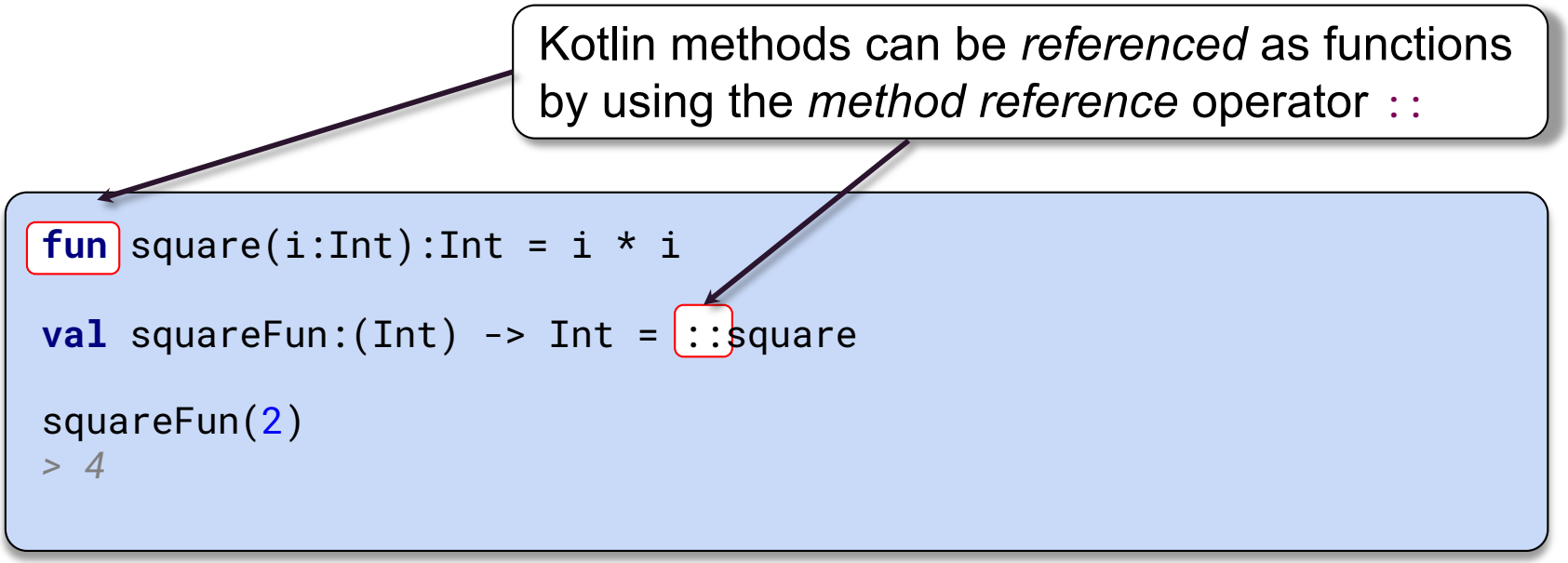
Kotlin requires the function body to be placed between curly braces.

Methods vs Functions

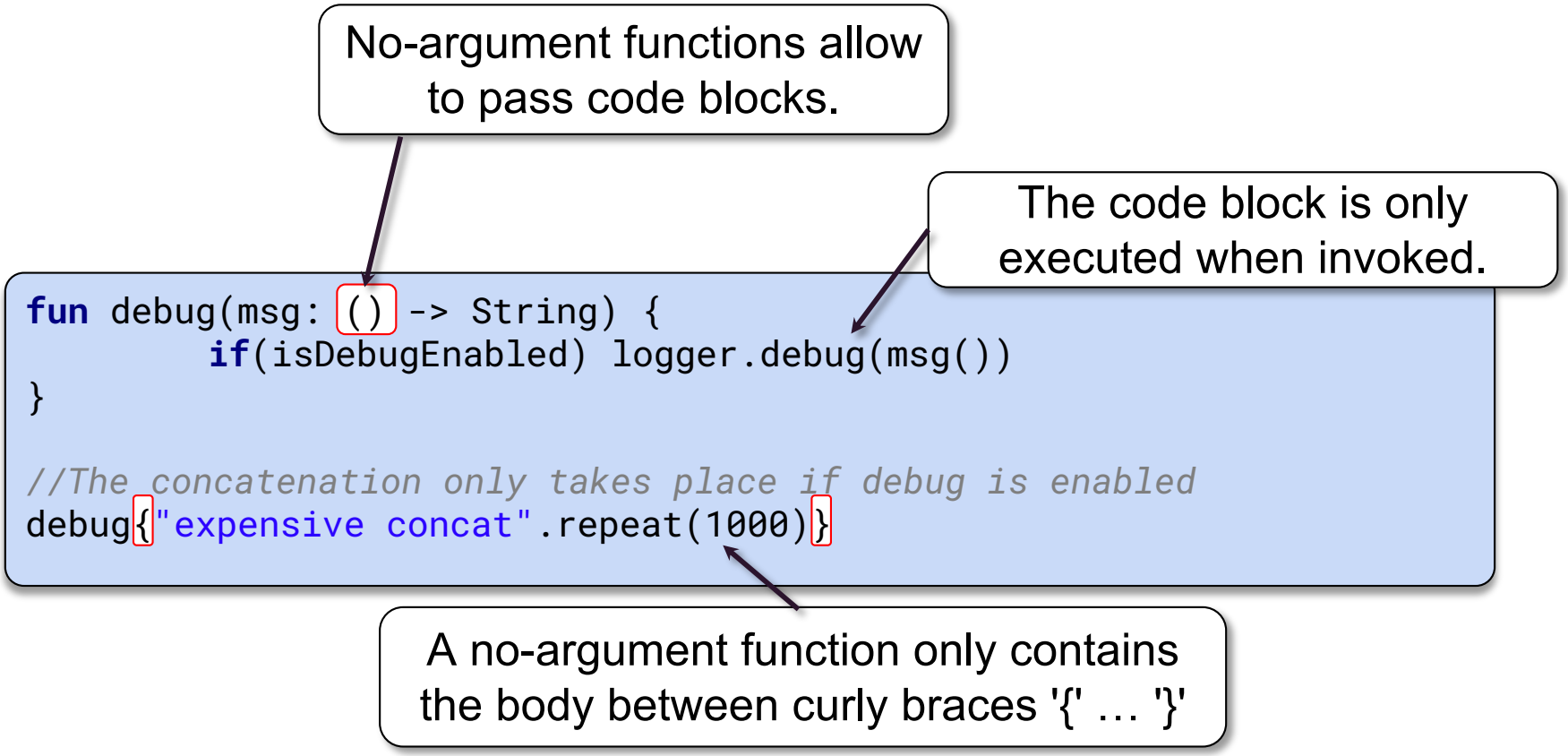
Kotlin methods are interchangeable with functions. Therefore, *methods* == *functions*

```
val squareFun:(Int) -> Int = fun (i:Int):Int = i * i

val squareFunCrisp = fun (i:Int) = i * i

squareFun(2) == squareFunCrisp(2)
> true
```

Method References



No-argument Functions

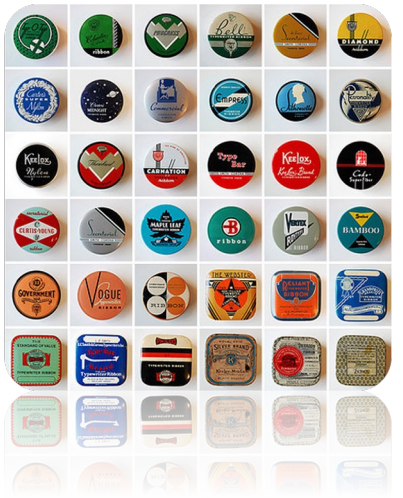


Exercise: Implement a Functional Construct

- Open: FunctionsExercise01/02 and FunctionsExerciseTest
- **Solve FunctionsExercise01**
  - Take a look at the predefined methods `reverseText()` and `upperCaseText()`. Between the two methods lots of code is duplicated, which we want to remove by means of a *higher order function* called: `doWithText{...}`
  - Implement the `doWithText{...}` method that takes care of the `File` resource handling and offers a *function argument* that allows to deal with the content of the `File` (a `String`) directly.
  - Make sure the test succeeds.
- **Solve FunctionsExercise02**
  - Create a method `measure` that accepts a *code block*, executes it and prints the execution time.
    - E.g. “The execution took <elapsed> ms”.
  - Use the `logPerf` method provided to log the execution time.
  - To measure the elapsed time make use of `System.currentTimeMillis()`
  - Provide a suitable implementation in order to make the unittest work.

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring



Collections in Kotlin

Most important collections in Kotlin:

1. List: an index-based, ordered collection of elements.

   (“Albert Heijn”, “Delhaize”, “Delhaize”, “AH ToGo”)

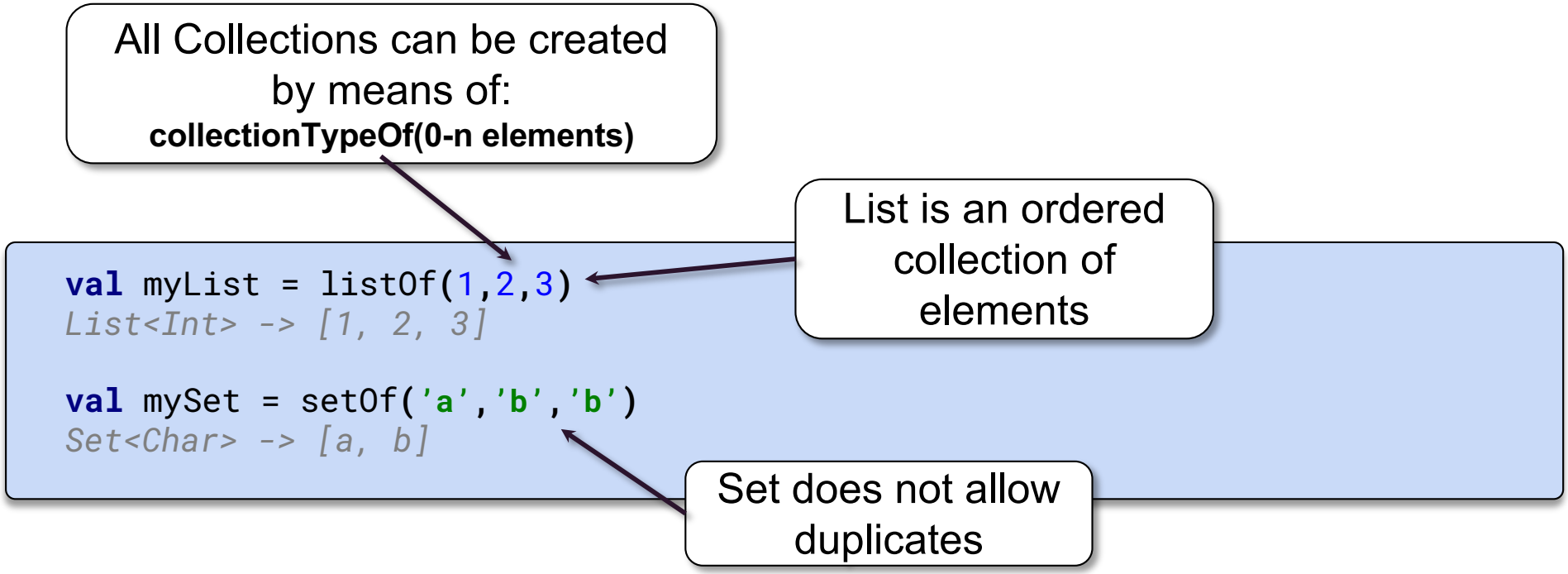
2. Set: an unordered collection without duplicates.

   (“Albert Heijn”, “Delhaize”, “AH ToGo”)

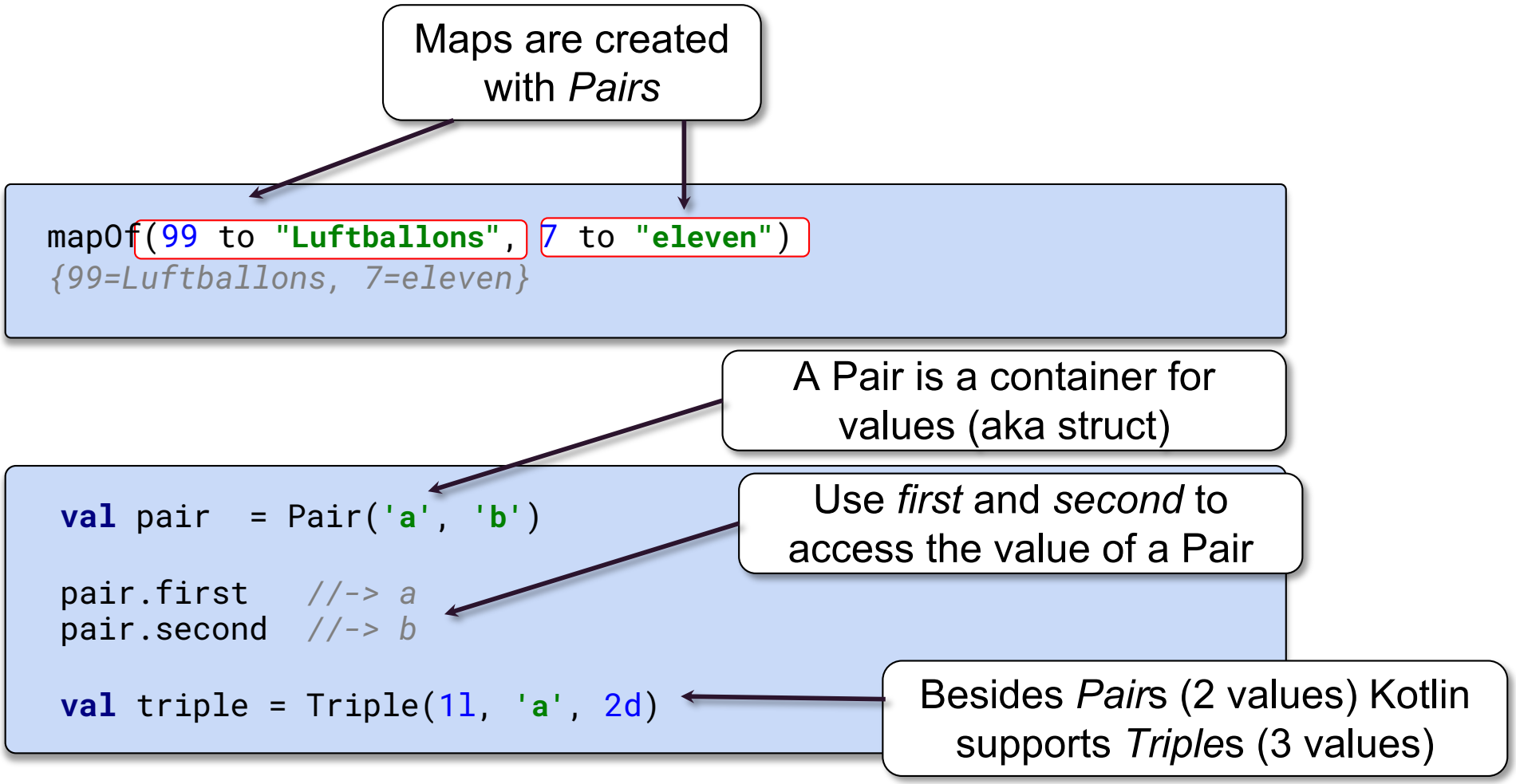
3. Map: collection of key-value pairs.

   (“Albert Heijn” → “NL”, “Delhaize” → “BE”)

Collections in Kotlin



Creating Maps & Tuples



Variable Arguments and Spread Operator

How is it possible that a method accepts *one* or *more* arguments?

```
linkedListOf("a")
linkedListOf("a", "b", "c")
```

… by means of the **vararg** modifier

```
fun <T> linkedListOf(vararg items:T):List<T> = LinkedList(items.toList())
```

The **vararg** parameter is of type `Array`

```
val items = arrayOf("a", "b", "c")
val list1:List<String> = linkedListOf(* items)
val list2:List<String> = linkedListOf(* items, "d")
```

If we already have an array but only want to pass its *contents* to a **vararg** method the *spread operator* * can be used

General Collection Methods

```
[0] or get(0) > element at index 0

first()       > first element
last()        > last element
drop(n)       > all except first n elements
take(n)       > first n elements

to<Coll>      > converts to specified collection,
                    toSet(), toList(), toMap(), toTypedArray()
etc.

size          > element count
isEmpty       > checks whether collection is empty
contains      > checks whether element exists

union         > combine two collections

//numeric collections only:
sum()         > sum of all elements (nullable type)
min()         > min element (nullable type)
max()         > max element (nullable type)
average()     > average of all elements (nullable type)
```

Useful Methods for Lists

```
+                -> append an element:
    listOf(1,2) + 3 ==  listOf(1,2,3)

-           -> remove an element:
    listOf(2,1) - 1 ==  listOf(2)

reversed     -> reverse the order of the elements:
        listOf(1,2,3).reversed() ==listOf(3,2,1)

subtract     -> compute the difference between two sequences:
        listOf(1,2) subtract listOf(2,3) ==listOf(1)

intersect    -> compute the intersection of two sequences
        listOf(1,2) intersect listOf(2,3) ==listOf(2)

zip          -> zip two sequences
        listOf(1) zip listOf('a') ==listOf(1 to 'a')

windowed     -> slice collection in n sub-collections
        listOf(1,2,3).windowed(2) ==listOf(listOf(1,2),listOf(2,3))

joinToString -> transform a collection into a String
        listOf(1,2,3).joinToString("-") =="1-2-3"
```

Useful Methods for Maps

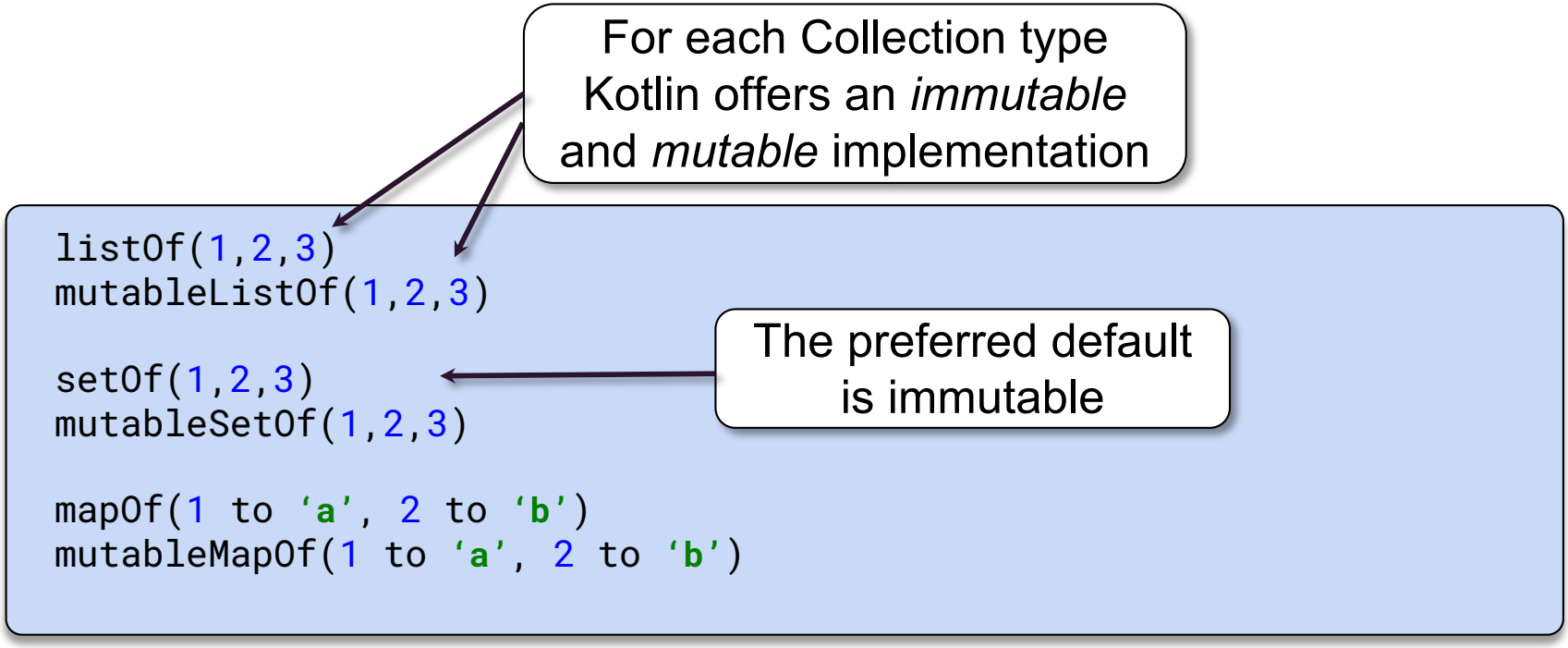
```
val map = mapOf(8 to "ung")
map.getValue(8)             > "ung"
map.getValue(9)             > java.util.NoSuchElementException
map.getOrDefault(9, "?")    > "?"
map.get(9)                  > null
map[9]                      > null

val map2 = mapOf(8 to "ung").withDefault{"?"}
map2.getValue(9)            > "?"
```

Useful Methods for Ranges

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

// 1
// 2
// 3
```

```
1..5                        > IntRange(1, 2, 3, 4, 5)
1 until 5                   > IntRange(1, 2, 3, 4)
'a'..'e'                    > CharRange(a, b, c, d, e)

1..10 step 2                > IntRange(1, 3, 5, 7, 9)
10 downTo 3 step 2          > IntRange(10, 8, 6, 4)
```

Immutable (read-only) vs Mutable Collections



Immutable (read-only) vs Mutable Collections

```
val mutable = mutableListOf(1,2)
mutable.add(3)
mutable: ... [1,2,3]
```

Adding a value to a *mutable* collection changes the contents of the collection

```
val immutable = listOf(1,2)
val i2 = immutable + 3
i2: ... [1,2,3]
immutable: ...[1,2]
```

Adding a value to an *immutable* collection always result in a *new* collection

Initial collection still has only 2 elements

Higher Order Functions for Collections - similar to SQL

Select Dutch students sorted by name and display name and grade

Name	Country	Grade
Albert	NL	9
Bernard	NL	6
Charley	BE	10
Dorine	NL	8

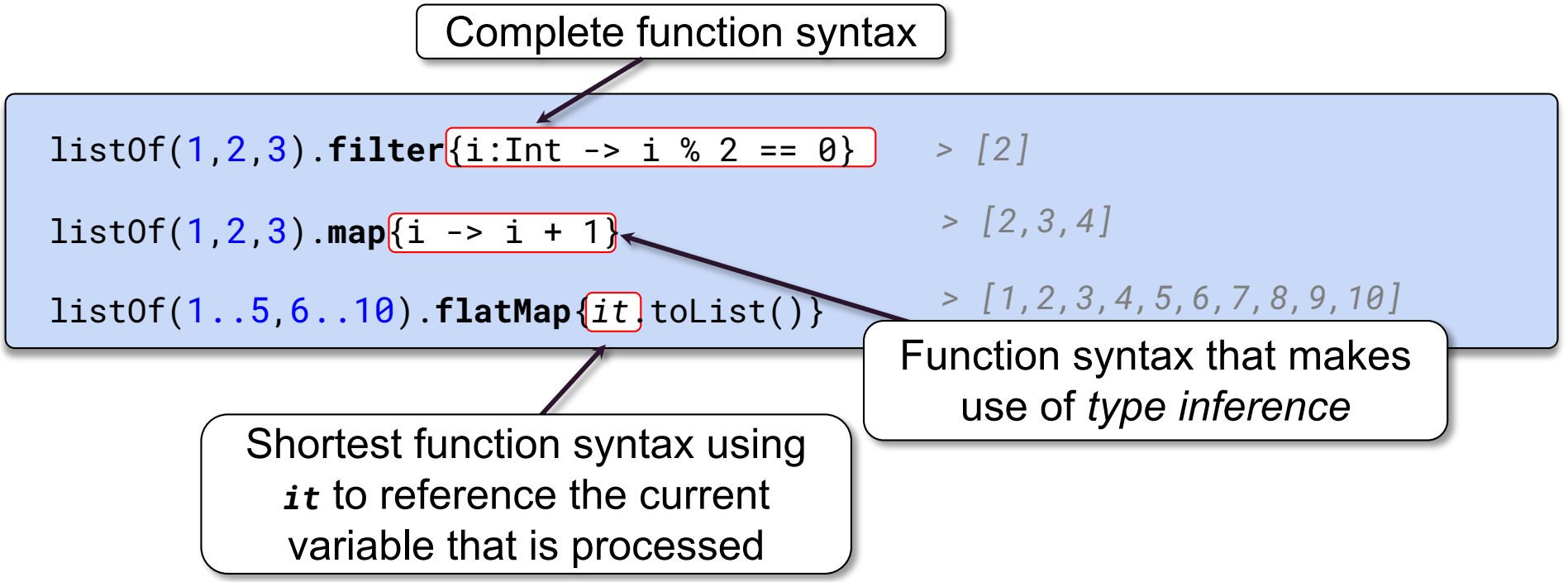
```sql
SELECT s.name || ': ' || s.grade
    FROM STUDENTS s
    WHERE s.country = 'NL'
    ORDER BY s.name;
```

```
val students = listOf(
    Student("Albert", NL, 9),
    Student("Bernard", NL, 6),
    Student("Charley", BE, 10),
    Student("Dorine", NL, 8)
)

val sortedNLGrades = students
    .filter{ it.country == NL }
    .sortedBy{ it.name }
    .map{ "${it.name}: ${it.grade}" }
```

- Re-use of generic control structures
- Less error prone than for loops
- Expressive and readable
- Chainable

Methods that accept functions as parameters are called **Higher Order Functions**

Higher Order Methods for Collections



Higher Order Methods for Collections

```
listOf(1,2,3).filter{i:Int -> i % 2 == 0}      > [2]
listOf(1,2,3).map{i -> i + 1}                  > [2,3,4]
listOf(1..5,5..10).flatMap{it.toList()}        > [1,2,3,4,5,6,7,8,9,10]
```

Using a method as function

```
fun evenOrOdd(item:Int) :Int = item % 2
listOf(1,2,3).groupBy(::evenOrOdd)             >{1=[1,3], 0=[2]}
listOf(1,2,3).forEach(System.out::print)       > 123
listOf(1,2,3).reduce{acc, i -> acc + i}        > 6
```

Higher Order Methods for Collections

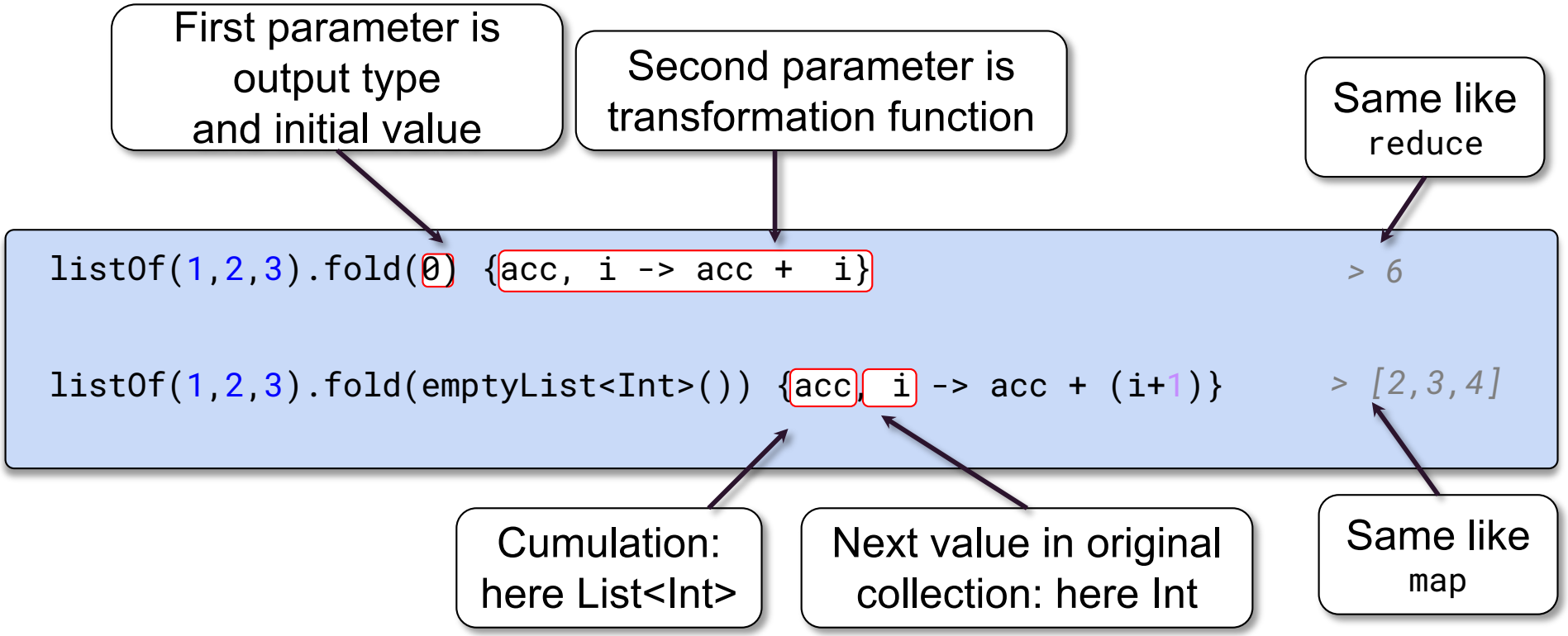
```
listOf("aaa","bb","c").all{it.length > 1}      > false

listOf("aaa","bb","c").maxByOrNull{it.length} > "aaa"

                                               > 6

listOf("aaa","bb","c").sumBy{it.length}
```

Higher Order Methods for Collections

```
listOf("aaa","bb","c").all{it.length >= 1}          > true

listOf("aaa","bb","c").maxByOrNull{it.length}       > "aaa"
                                                    > 6
listOf("aaa","bb","c").sumBy{it.length}
```

```
listOf(1,2,3).takeWhile{it < 3}                     > [1,2]

listOf(1,2,3).dropWhile{it < 3}                     > [3]

listOf(1,2,3).last{it < 3}                          > 2
```

Fold: one to rule them all



Bottom line:
with fold (almost) all higher order functions can be implemented

Group exercise: Google Code Jam Contest

Solve the Problem
The aim of this task is to translate a language into a new language called rese. To translate we take any message and replace each English letter with another English letter. This mapping is one-to-one and onto, which means that the same input letter always gets replaced with the same output letter, and different input letters always get replaced with different output letters. A letter may be replaced by itself. Spaces are left as-is.

For example (and here is a hint!), the translation algorithm includes the following three mappings: 'a' -> 'y', 'o' -> 'e', and 'z' -> 'q'. This means that "**a zoo**" will become "**y qee**".

Sample Input/Output

Input: Case #1: `ejp mysljylc kd kxveddknmc re jsicpdrysi`
Case #2: `rbcpc ypc rtcsra dkh wyfrepkym veddknkmkrkcd`
Case #3: `de kr kd eoya kw aej tysr re ujdr lkgc jv`

Output: Case #1: `our language is impossible to understand`
Case #2: `there are twenty six factorial possibilities`
Case #3: `so it is okay if you want to just give up`

Taken from: http://code.google.com/codejam/contest/1460..8/dashboard

Exercise: Use Functional Collections

- Open: CollectionExercise01 and CollectionExerciseTest
- **Implement: groupAdultsPerAgeGroup**

  Take a look at the **SQL example** in: **`src/main/resources/challenge07.sql`**. Using the query that calculates for each adult it's age group (e.g. Duke is 32, so his age group is 30) as inspiration, rewrite the algorithm in Kotlin using functional collections. Here are some hints:

```
* 1. filter out all adults of the list of persons
* 2. sort the list by name
* 3. group each person by their age group, e.g. 30 -> List<duke, jeniffer>
*    The age group you get as follows: age / 10 * 10
```

- Open: CollectionExercise02 and CollectionExerciseTest
- **Implement: calcLengthLongestWord**
  - Calculate the length of the longest word in a list of sentences

  Hint: To keep it simple it's ok to use `String.split(" ")` to extract all words of a sentence.

- **Bonus**: Open: CollectionExercise03 and CollectionExerciseTest
- **Implement: filterWithFold** and **groupByWithFold**
  - Provide an alternative implementation for the higher order function `filter` using `fold`
  - Provide an alternative implementation for the higher order function `groupBy` using `fold`

Sequences

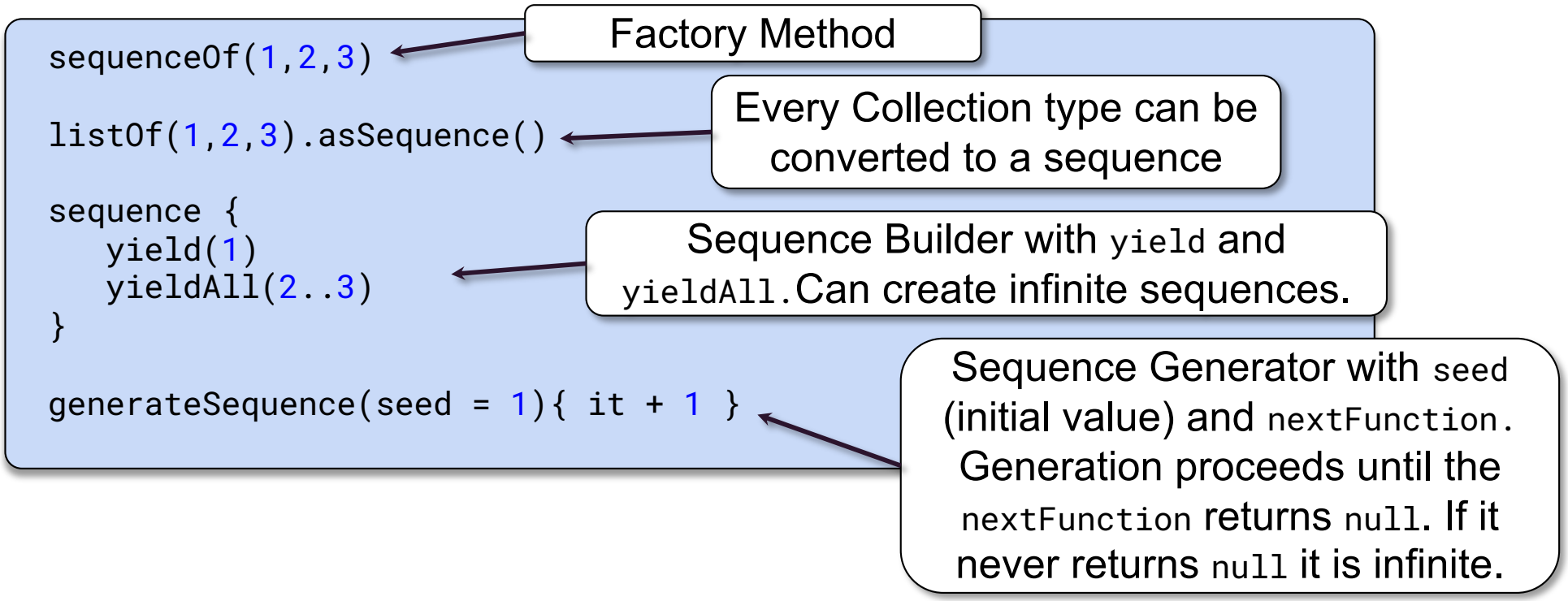
Sequences evaluate data *lazily*, unlike Collections who do that *eagerly.*

Sequences have *intermediate operations* (`map`, `filter`, etc.), which are not evaluated right away but stored. They always return another `Sequence`.

```
val seq1:Sequence<Int>    = sequenceOf(1,2,3)
val seq2:Sequence<String> = seq1.map{it.toString()}
val res:String            = seq2.last()
```

Terminal operations (`toList()`, `last()`, `reduce()`, `sum()`, `count()` etc.) will trigger all intermediate operations *on each element* in *a row* and return a concrete result, thus not another `Sequence`.

Creating Sequences



When to use Sequences

Collections

```
(1..5_000_000).toList()
```

1	3	5	7	9	...
"1"	"3"	"5"	"7"	"9"	...
"1"					

```
.filter{it % 2 != 0}
.map{it.toString()}
.first()
```

1
"1"
"1"

Collections create for every transformation a new Collection

Transformation methods are *inlined*, so no new objects are created on traversal

Sequences

```
(1..5_000_000).asSequence()
```

Sequences rely on the Iterator of the original collection, so transformations will not result in new intermediate Collections

Transformation methods are not inlined, so new Function objects are created for each operation

Bottom line: - Use Collections for small datasets with ~2-4 operands.
- Use Sequences for large datasets.

Exercise: Sequences

- Open: CollectionExercise04 and CollectionExerciseTest
- **Implement: createListOfSentencesAndCalcTheSumOfAllWords**
  - Generate a list with as many sentences as defined in the input-parameter `sentencesCount` using the provided `createSentence()`utility method.
  - Calculate the sum of all words of the generated sentences.

  Hint: To keep it simple it's ok to use `String.split` to extract all words of a sentence.
- **Bonus**: Open: CollectionExercise05 and CollectionExerciseTest
- **Solve** Euler Problem No 2

  Each new term in the Fibonacci sequence is generated by adding the previous two terms.

  By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

  By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.
- **A) Implement: fibonacci sequence**
  - Implement the `fibonacci()` method by using a `Sequence<T>` that generates the fibonacci numeric sequence as mentioned above: (1,2,3,5,8 etc.)

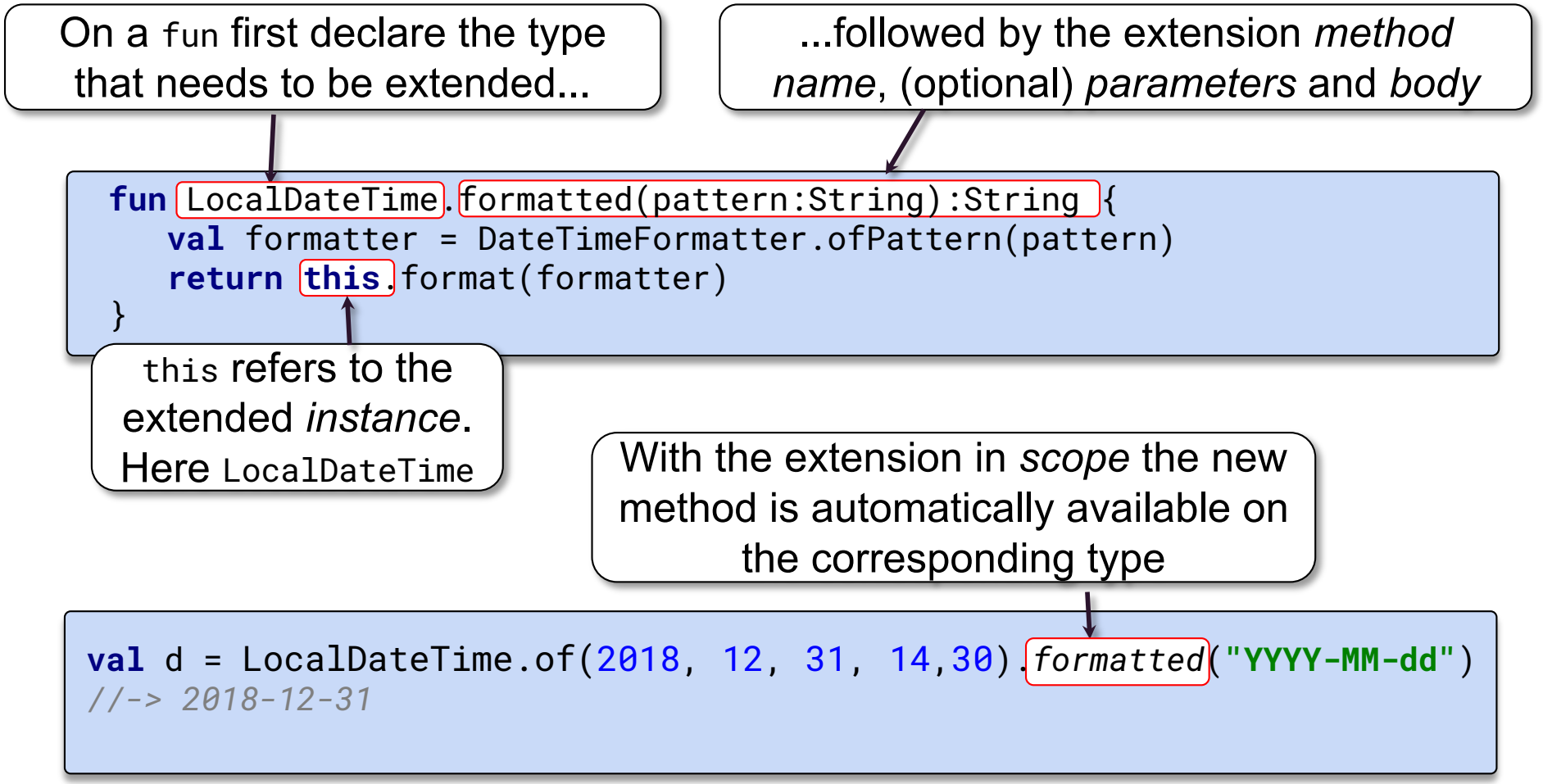
  Hint: the mathematical sequence looks like 0,1,1,2,3,5 etc., so make sure you drop the first 2 elements
- **B) Implement: eulerProblem2**
  - Finally solve Euler Problem No 2: Implement the `eulerProblem2(max:Int)` method that takes a `max` parameter which represents the ceiling to which the fibonacci sequence needs to be generated. Make use of the previously implemented `fibonacci` method.

Agenda

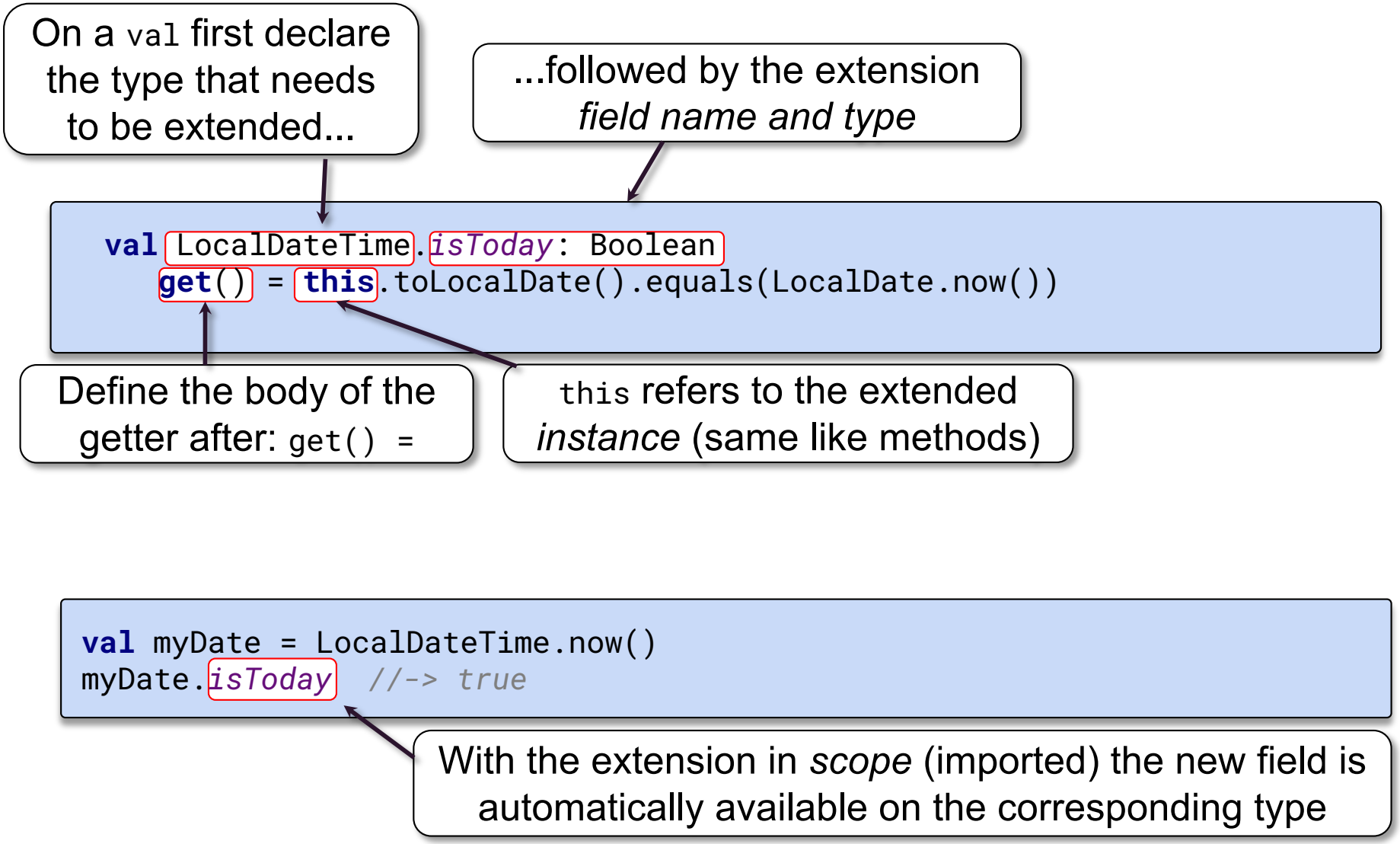
- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

Extensions Methods

Kotlin has a powerful mechanism to add *new methods* or *fields* to *existing* classes. These added methods or fields are called 'Extensions'.



Extensions Properties



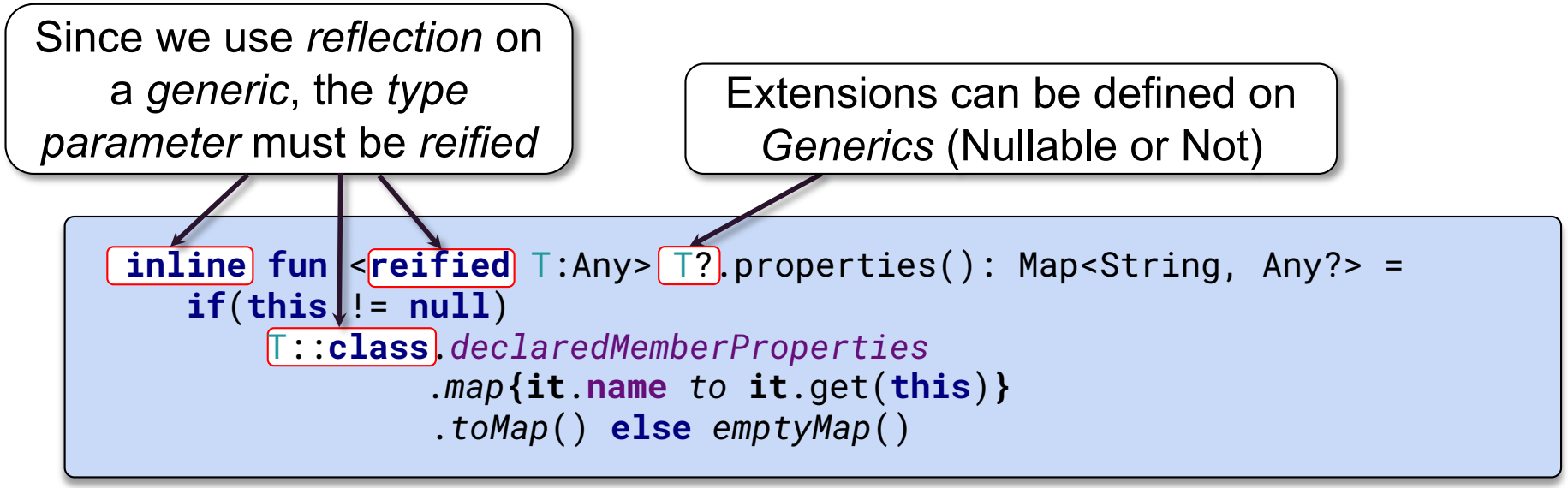
Extension on Nullable Types

Extensions can also be defined on *Nullable Types*

```
fun String?.toLocalDateTime():LocalDateTime? =
   if(this == null) null else LocalDateTime.parse(this)
```

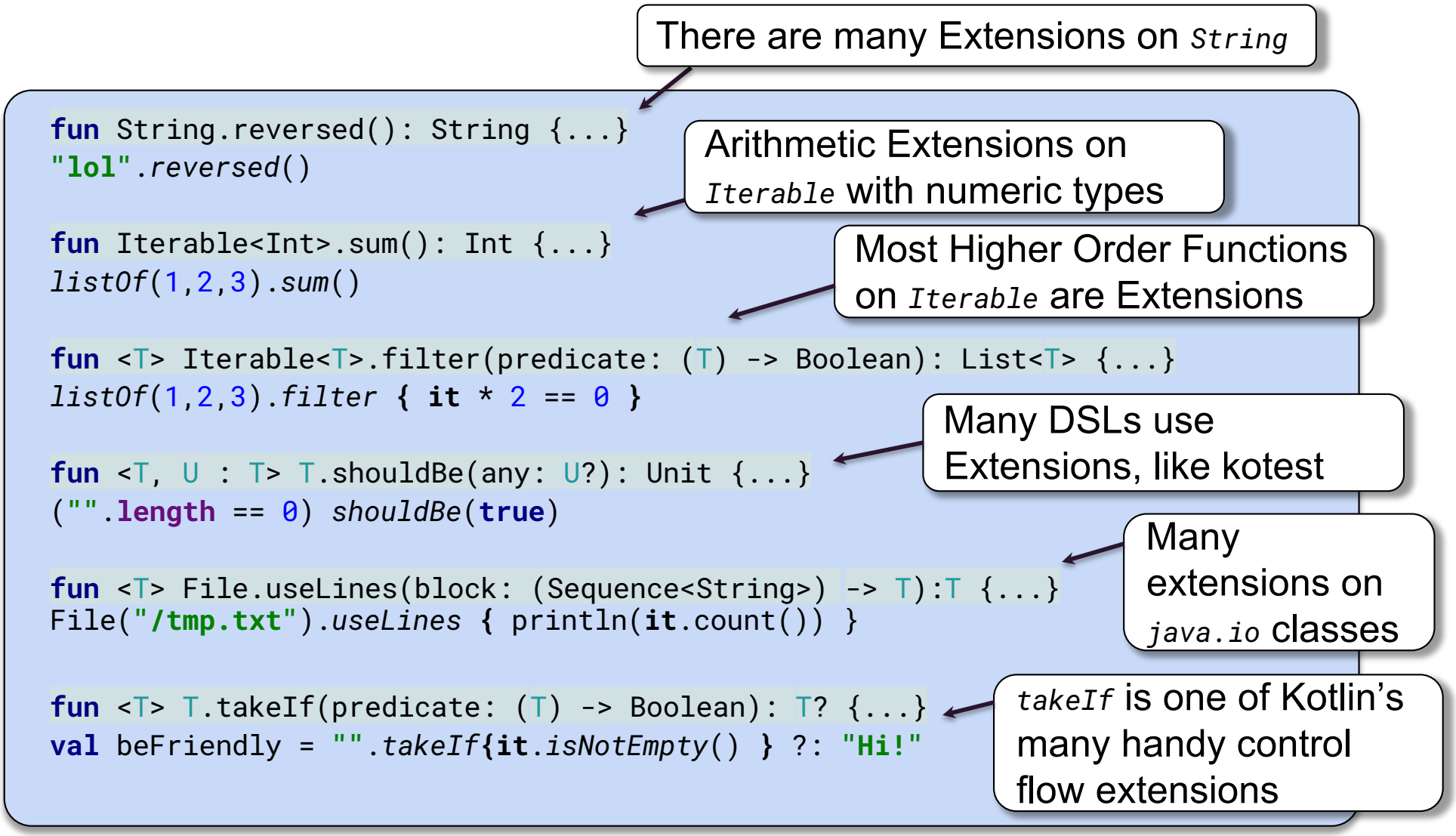
```
var dateStr:String? = null
dateStr.toLocalDateTime()      //-> null
dateStr = "2018-12-31T14:30"
dateStr.toLocalDateTime()      //-> LocalDateTime(2018-12-31T14:30)
```

Extension on Generic Types




```
var person:Person? = null
person.properties() //-> {}

person = Person(name = "Jack", age = 42)
person.properties() //-> {name=Jack, age=42}
```

Extension in APIs



Exercise: Extension Methods and Fields

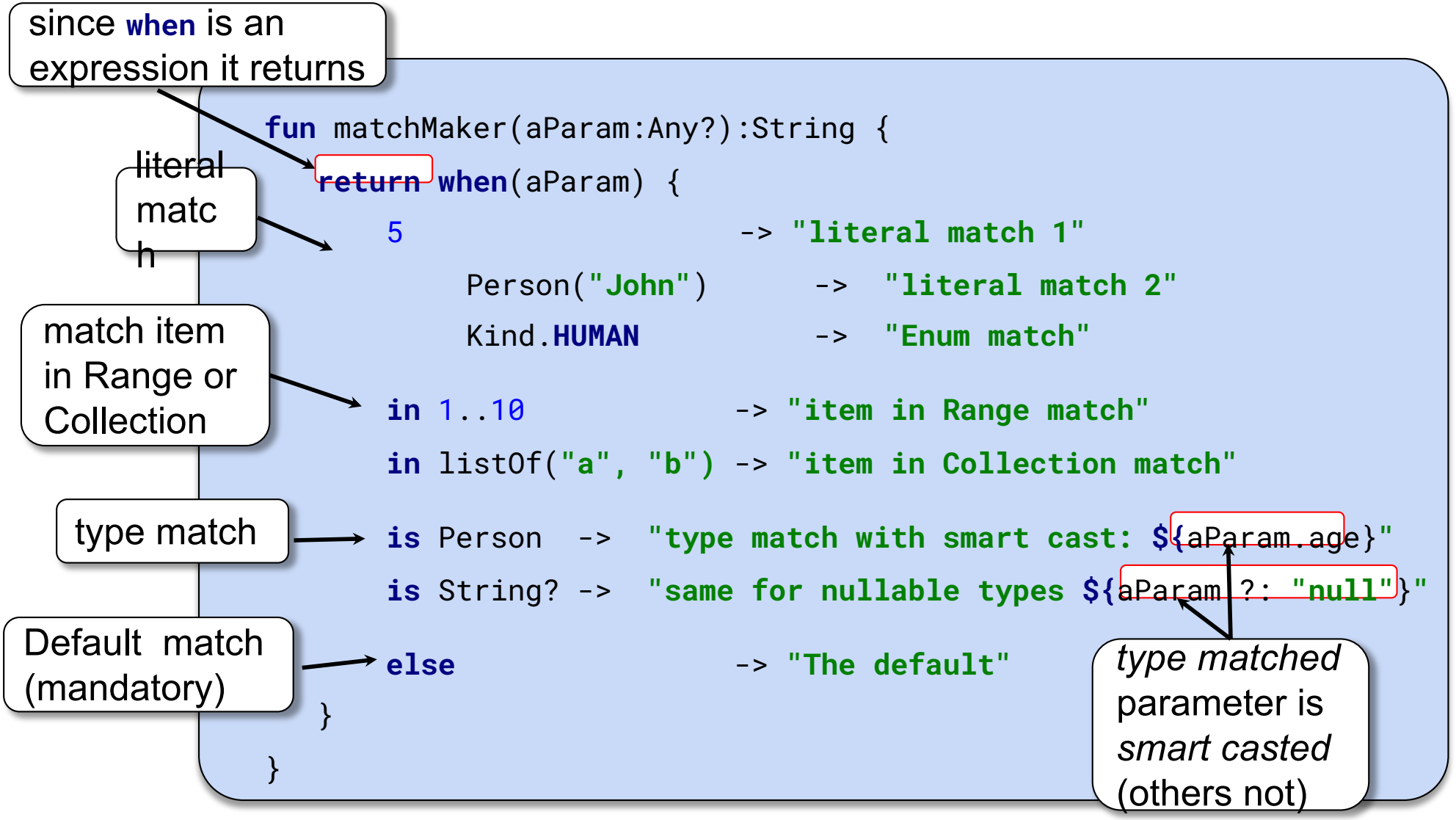
- Open: ExtensionExercise1-4.kt and ExtensionExerciseTest
- **Exercise 1**
  - Define an extension *method* `square()` on `Int` that calculates the square
- **Exercise 2**
  - Define an extension *property* `tail` on `String` that returns the remaining part of the `String` after the first character or an empty `String` if no character(s) is/are available.
  - Tip: `String`'s `drop(...)` method might be of good use...
- **Exercise 3**
  - Define an extension method `sameLength(other:String?):Boolean` on nullable String's (`String?`) to check that they have the same length regardless whether they are null or not.
- **Bonus Exercise 4**
  - Define an extension *method* `filterByType()` on `Iterable<Any>`, which filters and returns all instances of the given type as a `List<T>`.

Make sure all tests succeed.

Agenda

- Introduction
- Object Orientation Basics
- Testing
- Null Safety
- Generics
- Functional Programming
- Collections
- Extensions
- Control-Structures and Destructuring

When expression



when expression as better if-else

Can readability of this long if-else chain be improved?

```
fun matchMaker(any: Any?): String {
    return if (any is String && any.contains("cool")) {
        "cool String"
    } else if (any in listOf(1, 2, 3)) {
        "1, 2 or 3"
    } else if (any == null) {
        "null"
    } else "default"
}
```

Yes! with a `when` expression without argument

```
fun matchMaker(any: Any?): String {
    return when {
        any is String && any.contains("cool") -> "cool String"
        any in listOf(1, 2, 3)                -> "1, 2 or 3"
        any == null                           -> "null"
        else                                  -> "default"
    }
}
```

Each branch condition is a boolean expression like in the if-else

Destructuring

Kotlin supports *destructuring*, which is a means to de-structure an object into multiple variables. Syntax: `val (variable1, variable2, variableN) = objectToDestructure`

```
val pair = Pair("Amsterdam", 23)
pair.first  //-> what's the semantic of first?
pair.second //-> what's the semantic of second?
```

```
val (city, temperature) = pair
//-> ah, now it's clear!
```

The variables names can be chosen freely

```
val cityTemps = mapOf("Amsterdam" to 23, "Rome" to 27)
cityTemps.map { kv -> "${kv.key};${kv.value}" } //-> key? value?
```

Destructuring also works in function arguments allowing for better semantic meaning of the fields of data holder classes, like *Pair*, *Map.Entry* etc.

```
cityTemps.map { (city, temperature) -> "$city;$temperature" }
```

Destructuring under the hood

For `data class`-es `component<var-id>` methods are generated for all primary constructor arguments

```
class Person(val name: String , val age: Int = 0) {
   operator fun component1() = name
   operator fun component2() = age
   operator fun component3() = "Human"
}
```

For every *destructurable* variable a `component<var-id>` `operator` method needs to be available that returns the intended value for position N

```
val jack = Person("Jack", 4)
val (jacksName, jacksAge, kind) = jack
val (jacksName, jacksAge) = jack
val (jacksName, _, kind) = jack
```

There is no need to declare a variable for *every* component. Here at least 1, max 3

The _ is used to skip the variable at the given position

Exercise: Control Flow

- Open: ControlFlowExercise01 and ControlFlowExerciseTest
- **Exercise 1: when expression with argument**

  Implement the `matchOnInputType(param:Any?):String` method using a *when expression with argument* `when(myArg) { … }` covering all the branch conditions required to make the unittest succeeds.

  One case might require nesting. Can you figure out how to use a *when expression without arguments* `when{ … }` to cover this case?

Copyright © 2023 Xebia BV All rights reserved

Unless otherwise agreed, training materials may only be used for educational and reference purposes by individual named participants in a training course offered by Xebia BV.
Unauthorized reproduction, redistribution, or use of this material is prohibited.