Functional programming with Scala

Classes

Djaouida ZAOUCHE, Romain DUJOL

CY TECH – ING2

2020 – 2021

Environment of a Scala application

Environment of a Scala application

sbt

sbt (*Simple Build Tool*)

Open-source build tool for Java and Scala projects
(Some Java equivalents : Apache Maven, Ant)

Features

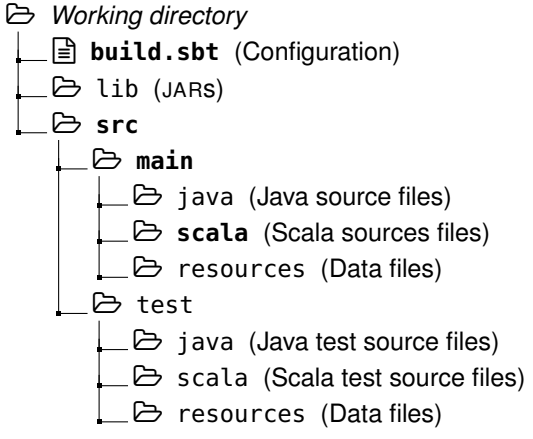
- Minimal configuration for simple projects
- Continuous compilation, testing, deployment
- Incremental compilation and testing
- Tasks written in Scala DSL
- Dependency support via Apache Ivy
- Interactive mode (console) is available

(Source : Wikipedia)

sbt (*Simple Build Tool*)

Create a Scala sbt project

1. Create following directories *(at least the ones in **bold**)* :



2. Create file `build.sbt`.

sbt (*Simple Build Tool*)

Example for `build.sbt` file

```
organization := "fr.cyu"
name         := "HelloWorld"
version      := "1.0-SNAPSHOT"
scalaVersion := "2.13.3"

// Add one dependency for tests (with '% "test"')
libraryDependencies += "junit" % "junit" % "4.8" % "test"

// Add various dependencies
libraryDependencies ++= Seq("org.scala-lang" % "scala-xml" % "2.11.0-M4",
                            "joda-time"      % "joda-time" % "2.9.5")
```

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive
- `run` *arguments* executes main class with *arguments* as parameters

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive
- `run` *arguments* executes main class with *arguments* as parameters
- `help` prints the list of available subcommands

sbt (*Simple Build Tool*)

Subcommands

Any subcommand may be invoked :

- as a parameter when calling `sbt` from terminal ;
- directy in sbt console.

Some useful subcommands

- `clean` deletes all files generated by build in `target` directory
- `compile` compiles all source files from `main` directory
- `test` compiles all source files from `test` directory and launch tests
- `package` builds a JAR archive
- `run` *arguments* executes main class with *arguments* as parameters
- `help` prints the list of available subcommands
- `help` *command* prints the help of *command*

Environment of a Scala application

Classes and objects

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

- Values (or variables)

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

- Values (or variables)
- Functions (a.k.a. *methods*) whose instance is implicitly is a parameter with keyword `this`

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

- Values (or variables)
- Functions (a.k.a. *methods*) whose instance is implicitly is a parameter with keyword **`this`**

Default declaration syntax

`class` *ClassName* [*ParameterTypes**] (*typedParameter**)
{
classBody
}

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

- Values (or variables)
- Functions (a.k.a. *methods*) whose instance is implicitly is a parameter with keyword **`this`**

Default declaration syntax

`class` *ClassName* [*ParameterTypes**] (*typedParameter**)
{
  classBody
}

- Usually in *ClassName*`.scala` (except for sealed classes)

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

- Values (or variables)
- Functions (a.k.a. *methods*) whose instance is implicitly is a parameter with keyword **`this`**

Default declaration syntax

```
class ClassName[ParameterTypes*](typedParameter*)
  {
    classBody
  }
```

- Usually in *ClassName*`.scala` (except for sealed classes)
- *ClassName* must start with an **uppercase** letter.

Class/Type

Definition (similar to Java)

A class described all the members defined for any instance :

- Values (or variables)
- Functions (a.k.a. *methods*) whose instance is implicitly is a parameter with keyword **`this`**

Default declaration syntax

```
class ClassName[ParameterTypes*](typedParameter*)
  {
    classBody
  }
```

- Usually in *ClassName*`.scala` (except for sealed classes)
- *ClassName* must start with an **uppercase** letter.
- *classBody* contains **`val`**/**`var`** and **`def`** declarations.

Class/Type

Example

Class/Type

Example

```
scala> class A
defined class A
```

Class/Type

Example

```
scala> class A
defined class A

scala> class A1 extends A
defined class A1
```

Class/Type

Example

```
scala> class A
defined class A

scala> class A1 extends A
defined class A1

scala> val a : A = new A()
a: A = A@2e8b24a1

scala> val a1 : A1 = new A1()
a1: A1 = A1@3f0c6b3c
```

Class/Type

Example

```
scala> class A
defined class A

scala> class A1 extends A
defined class A1

scala> val a : A = new A()
a: A = A@2e8b24a1

scala> val a1 : A1 = new A1()
a1: A1 = A1@3f0c6b3c

scala> val test : A = a1
test: A = A1@3f0c6b3c
```

Class/Type

Example

```
scala> class A
defined class A

scala> class A1 extends A
defined class A1

scala> val a : A = new A()
a: A = A@2e8b24a1

scala> val a1 : A1 = new A1()
a1: A1 = A1@3f0c6b3c

scala> val test : A = a1
test: A = A1@3f0c6b3c

scala> val test : A1 = a
<console>:13: error: type mismatch;
 found   : A
 required: A1
       val test : A1 = a
                       ^
```

Class

Instantation : `new`

- Same syntax as Java

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

- `final class` : same as Java (cannot have subtypees)

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

- `final class` : same as Java (cannot have subtypees)
- `sealed class` : all subtypees must be in the same *compilation unit* (typically a file)

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

- `final class` : same as Java (cannot have subtypees)
- `sealed class` : all subtypees must be in the same *compilation unit* (typically a file)
- `case class` : class designed for pattern matching

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

- `final class` : same as Java (cannot have subtypees)
- `sealed class` : all subtypees must be in the same *compilation unit* (typically a file)
- `case class` : class designed for pattern matching
- `abstract class` : same as Java (i.e. non-instantiable class)

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

- `final class` : same as Java (cannot have subtypees)
- `sealed class` : all subtypees must be in the same *compilation unit* (typically a file)
- `case class` : class designed for pattern matching
- `abstract class` : same as Java (i.e. non-instantiable class)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by class

Class

Instantation : `new`

- Same syntax as Java

Special declarations (instead of just `class`)

- `final class` : same as Java (cannot have subtypees)
- `sealed class` : all subtypees must be in the same *compilation unit* (typically a file)
- `case class` : class designed for pattern matching
- `abstract class` : same as Java (i.e. non-instantiable class)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by class

Like Java, a class may extend at most one class.

Definition

Object

Definition

- An object is a **class with one immutable instance** so :

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;
  - an object **cannot be parameter-dependant**.

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;
  - an object **cannot be parameter-dependant**.
- This instance is treated as a lazy value.

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;
  - an object **cannot be parameter-dependant**.
- This instance is treated as a lazy value.

Declaration syntax

```
object ObjectName
  {
    objectBody
  }
```

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;
  - an object **cannot be parameter-dependant**.
- This instance is treated as a lazy value.

Declaration syntax

```
object ObjectName
  {
    objectBody
  }
```

- Usually in *ObjectName*`.scala`

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;
  - an object **cannot be parameter-dependant**.
- This instance is treated as a lazy value.

Declaration syntax

```
object ObjectName
  {
    objectBody
  }
```

- Usually in *ObjectName*`.scala`
- *ObjectName* must start with an **uppercase** letter.

Object

Definition

- An object is a **class with one immutable instance** so :
  - an object **cannot be type-dependant** ;
  - an object **cannot be parameter-dependant**.
- This instance is treated as a lazy value.

Declaration syntax

```
object ObjectName
  {
    objectBody
  }
```

- Usually in *ObjectName*`.scala`
- *ObjectName* must start with an **uppercase** letter.
- *objectBody* contains **`val`**/**`var`** and **`def`** declarations.

Object

Special declaration

- `case object` : `case class` and `object` (e.g. `None`)

Object

Special declaration

- **`case object`** : **`case class`** and **`object`** (e.g. **`None`**)

Extensions

- **`extends`** : first superclass (or implemented trait)
- **`with`** : for each other trait implemented by object

Object

Special declaration

- `case object` : `case class` and `object` (e.g. `None`)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by object

Companion object

Object

Special declaration

- `case object` : `case class` and `object` (e.g. `None`)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by object

Companion object

Object with same name as a class (usually in same file)

Object

Special declaration

- `case object` : `case class` and `object` (e.g. `None`)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by object

Companion object

Object with same name as a class (usually in same file)

- Class can access private members of its companion object.

Object

Special declaration

- `case object` : `case class` and `object` (e.g. `None`)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by object

Companion object

Object with same name as a class (usually in same file)

- Class can access private members of its companion object.
- Members of companion object do not depend on any class instance

Object

Special declaration

- `case object` : `case class` and `object` (e.g. `None`)

Extensions

- `extends` : first superclass (or implemented trait)
- `with` : for each other trait implemented by object

Companion object

Object with same name as a class (usually in same file)

- Class can access private members of its companion object.
- Members of companion object do not depend on any class instance (think Java's `static`).

Environment of a Scala application

Main class

Main class. . . rather main object

Same purpose as Java. . .

- Entry point for program with `main` method
- Arguments from call in console captured in a string array

Main class. . . rather main object

Same purpose as Java. . .

- Entry point for program with `main` method
- Arguments from call in console captured in a string array

. . . but some differences in defining it

- One instance $\Longrightarrow$ **`object`** declaration (instead of **`class`**)
- **`App`** trait defines `main` method :
  - no need to declare method if object extends **`App`** ;
  - **the whole object body becomes the `main` method body**.

Main class. . . rather main object

Hello World

```
object HelloWorld
  {
    def main(args : Array[String])
      {
        println("Hello World !")
      }
  }
```

Main class. . . rather main object

Hello World

```
object HelloWorld
  {
    def main(args : Array[String])
      {
        println("Hello World !")
      }
  }
```

Hello World (extending **App**)

```
object HelloWorld extends App
  {
    println("Hello, world!")
  }
```

(Not recommended in Scala 3 for performance reasons)

Classes

Classes

Members

Members declarations

As expected. . .

Members declarations

As expected. . .

- Values : with `val` (or `lazy val`)

Members declarations

As expected...

- Values : with `val` (or `lazy val`)
- Variables : with `var`

Members declarations

As expected...

- Values : with `val` (or `lazy val`)
- Variables : with `var`
- Methods : with `def`

Members declarations

As expected...

- Values : with `val` (or `lazy val`)
- Variables : with `var`
- Methods : with `def`

By default, visibility is **public**.

Members declarations

As expected...

- Values : with `val` (or `lazy val`)
- Variables : with `var`
- Methods : with `def`

By default, visibility is **public**.
Preceding declaration with `private` makes it private.

Members declarations

As expected...

- Values : with **`val`** (or **`lazy val`**)
- Variables : with **`var`**
- Methods : with **`def`**

By default, visibility is **public**.
Preceding declaration with **`private`** makes it private.

Example

Members declarations

As expected...

- Values : with **`val`** (or **`lazy val`**)
- Variables : with **`var`**
- Methods : with **`def`**

By default, visibility is **public**.
Preceding declaration with **`private`** makes it private.

Example

```
scala> class A {
           val x : Int = 2
           private val ans : Int = 42
       }
defined class A
```

Members declarations

As expected...

- Values : with **`val`** (or **`lazy val`**)
- Variables : with **`var`**
- Methods : with **`def`**

By default, visibility is **public**.
Preceding declaration with **`private`** makes it private.

Example

```
scala> class A {
           val x : Int = 2
           private val ans : Int = 42
         }
defined class A

scala> (new A()).x
res0: Int = 2
```

Members declarations

As expected...

- Values : with **`val`** (or **`lazy val`**)
- Variables : with **`var`**
- Methods : with **`def`**

By default, visibility is **public**.
Preceding declaration with **`private`** makes it private.

Example

```
scala> class A {
           val x : Int = 2
           private val ans : Int = 42
         }
defined class A

scala> (new A()).x
res0: Int = 2

scala> (new A()).ans
<console>:13: error: value ans in class A cannot be accessed in A
       (new A()).ans
                 ^
```

Method call

Standard (Java-like) notation

Method call

Standard (Java-like) notation

- *instance* . *methodName* (*parameters*)

Method call

Standard (Java-like) notation

- *instance*.*methodName*(*parameters*)
- Parameters may be called in another order if **named**.

Method call

Standard (Java-like) notation

- *instance*.*methodName*(*parameters*)
- Parameters may be called in another order if **named**.

Example

Method call

Standard (Java-like) notation

- *instance*.*methodName*(*parameters*)
- Parameters may be called in another order if **named**.

Example

```
scala> class A {
           def f(x : Int, y : Int) = x + y
       }
defined class A
```

Method call

Standard (Java-like) notation

- *instance*.*methodName*(*parameters*)
- Parameters may be called in another order if **named**.

Example

```
scala> class A {
           def f(x : Int, y : Int) = x + y
         }
defined class A

scala> (new A()).f(2, 3)
res0: Int = 5
```

Method call

Standard (Java-like) notation

- *instance*.*methodName*(*parameters*)
- Parameters may be called in another order if **named**.

Example

```
scala> class A {
           def f(x : Int, y : Int) = x + y
         }
defined class A

scala> (new A()).f(2, 3)
res0: Int = 5

scala> (new A()).f(y = 3, x = 2)
res1: Int = 5
```

Method call

Infix notation

Method call

Infix notation

- **Only valid for a method with one parameter**

Method call

Infix notation

- **Only valid for a method with one parameter**
- *instance*␣*methodName*␣*parameter*
  ≡ *instance* . *methodName* (*parameter*)

Method call

Infix notation

- **Only valid for a method with one parameter**
- *instance* ␣ *methodName* ␣ *parameter*
  $\equiv$ *instance* . *methodName* (*parameter*)

Comes in handy to define DSLs (e.g. sbt)

Method call

Infix notation

- **Only valid for a method with one parameter**
- *instance* ␣ *methodName* ␣ *parameter*
  $\equiv$ *instance* . *methodName* (*parameter*)

Comes in handy to define DSLs (e.g. sbt)

Example

Method call

Infix notation

- **Only valid for a method with one parameter**
- *instance* ␣ *methodName* ␣ *parameter*
  ≡ *instance* . *methodName* (*parameter*)

Comes in handy to define DSLs (e.g. sbt)

Example

```
scala> class A(x : Int) {
           def + (y : Int) = x + y
         }
defined class A
```

Method call

Infix notation

- **Only valid for a method with one parameter**
- *instance* ␣ *methodName* ␣ *parameter*
  ≡ *instance* . *methodName* (*parameter*)

Comes in handy to define DSLs (e.g. sbt)

Example

```
scala> class A(x : Int) {
           def + (y : Int) = x + y
         }
defined class A

scala> (new A(2)).+(3)
res0: Int = 5
```

Method call

Infix notation

- **Only valid for a method with one parameter**
- *instance* ␣ *methodName* ␣ *parameter*
  ≡ *instance* . *methodName* (*parameter*)

Comes in handy to define DSLs (e.g. sbt)

Example

```
scala> class A(x : Int) {
           def + (y : Int) = x + y
         }
defined class A

scala> (new A(2)).+(3)
res0: Int = 5

scala> (new A(2)) + 3
res1: Int = 5
```

Classes

Constructors

Primary constructor

Primary constructor is implicitly defined in class declaration

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Since class parameters are actually method parameters, *same mechanisms apply* (in particular, default parameters).

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Since class parameters are actually method parameters, *same mechanisms apply* (in particular, default parameters).

WARNING !
By default, class parameters are **private immutable** members.
Adding some modifier before parameter alter this behaviour :

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Since class parameters are actually method parameters, *same mechanisms apply* (in particular, default parameters).

WARNING !
By default, class parameters are **private immutable** members.
Adding some modifier before parameter alter this behaviour :

- `val` : parameter is a **public immutable** member ;

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Since class parameters are actually method parameters, *same mechanisms apply* (in particular, default parameters).

WARNING !

By default, class parameters are **private immutable** members.
Adding some modifier before parameter alter this behaviour :

- `val` : parameter is a **public immutable** member ;
- `var` : parameter is a **public mutable** member ;

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Since class parameters are actually method parameters, *same mechanisms apply* (in particular, default parameters).

WARNING !

By default, class parameters are **private immutable** members.
Adding some modifier before parameter alter this behaviour :

- `val` : parameter is a **public immutable** member ;
- `var` : parameter is a **public mutable** member ;
- `private val` : parameter is a **private immutable** member ;

Primary constructor

Primary constructor is implicitly defined in class declaration

- Parameters : *the list of parameters of the class*
- Body : *the whole body of the class*

Since class parameters are actually method parameters, *same mechanisms apply* (in particular, default parameters).

WARNING !

By default, class parameters are **private immutable** members.
Adding some modifier before parameter alter this behaviour :

- `val` : parameter is a **public immutable** member ;
- `var` : parameter is a **public mutable** member ;
- `private val` : parameter is a **private immutable** member ;
- `private var` : parameter is a **private mutable** member.

Primary constructor

By default, class parameter is **private**

Primary constructor

By default, class parameter is **private**

```
scala> class A(x : Int) {
         def test = println(this.x)
       }
defined class A
```

Primary constructor

By default, class parameter is **private**

```
scala> class A(x : Int) {
         def test = println(this.x)
       }
defined class A

scala> (new A(2)).test
2
```

Primary constructor

By default, class parameter is **private**

```
scala> class A(x : Int) {
         def test = println(this.x)
       }
defined class A

scala> (new A(2)).test
2

scala> (new A(2)).x
<console>:13: error: value x is not a member of A
       (new A(2)).x
                  ^
```

Primary constructor

By default, class parameter is **private**

```
scala> class A(x : Int) {
         def test = println(this.x)
       }
defined class A

scala> (new A(2)).test
2

scala> (new A(2)).x
<console>:13: error: value x is not a member of A
       (new A(2)).x
                  ^
```

By default, class parameter is **immutable**

Primary constructor

By default, class parameter is **private**

```
scala> class A(x : Int) {
         def test = println(this.x)
       }
defined class A

scala> (new A(2)).test
2

scala> (new A(2)).x
<console>:13: error: value x is not a member of A
       (new A(2)).x
                  ^
```

By default, class parameter is **immutable**

```
scala> class A(x : Int) {
         def test = { x = -1 ; println(this.x) }
       }
<console>:12: error: reassignment to val
         def test = { x = -1 ; println(this.x) }
                      ^
```

Primary constructor

Example with **val**

```
scala> class A(val x : Int)
defined class A
```

Primary constructor

Example with **val**

```
scala> class A(val x : Int)
defined class A

scala> (new A(2)).x
res0: Int = 2
```

Primary constructor

Example with **val**

```
scala> class A(val x : Int)
defined class A

scala> (new A(2)).x
res0: Int = 2
```

Example with **var**

```
scala> class A(var x : Int)
defined class A
```

Primary constructor

Example with **val**

```
scala> class A(val x : Int)
defined class A

scala> (new A(2)).x
res0: Int = 2
```

Example with **var**

```
scala> class A(var x : Int)
defined class A

scala> val a = new A(2) ; a.x = 3
a: A = A@74cd798f
a.x: Int = 3
```

Auxiliary constructors

Same principle as Java

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **`this`**

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **`this`**
- Signature must be different from other constructors (primary or auxiliary).

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **`this`**
- Signature must be different from other constructors (primary or auxiliary).

Any internal call to a constructor is done with keyword **`this`**.

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **`this`**
- Signature must be different from other constructors (primary or auxiliary).

Any internal call to a constructor is done with keyword **`this`**.

Example

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **this**
- Signature must be different from other constructors (primary or auxiliary).

Any internal call to a constructor is done with keyword **this**.

Example

```
scala> class A(x : Int, y : Int) {
            def this(x : Int) = this(x, 0)
          }
defined class A
```

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **this**
- Signature must be different from other constructors (primary or auxiliary).

Any internal call to a constructor is done with keyword **this**.

Example

```
scala> class A(x : Int, y : Int) {
           def this(x : Int) = this(x, 0)
         }
defined class A

scala> new A(1, 2)
res0: A = A@5cbe95b1
```

Auxiliary constructors

Same principle as Java

- Auxiliary constructor : method whose name is **this**
- Signature must be different from other constructors (primary or auxiliary).

Any internal call to a constructor is done with keyword **this**.

Example

```
scala> class A(x : Int, y : Int) {
           def this(x : Int) = this(x, 0)
         }
defined class A

scala> new A(1, 2)
res0: A = A@5cbe95b1

scala> new A(1)
res1: A = A@65d7eea4
```

Classes

Case class

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference
- `toString` method is updated accordingly

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference
- `toString` method is updated accordingly

Companion object is implicitly created with :

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference
- `toString` method is updated accordingly

Companion object is implicitly created with :

- method `apply` with same parameters as class

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference
- `toString` method is updated accordingly

Companion object is implicitly created with :

- method `apply` with same parameters as class
  $\Longrightarrow$ keyword `new` is no longer required :
  CompanionObject(*parameters*)
  $\equiv$ *CompanionObject*.`apply`(*parameters*)

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference
- `toString` method is updated accordingly

Companion object is implicitly created with :

- method `apply` with same parameters as class
  $\Longrightarrow$ keyword `new` is no longer required :
  CompanionObject`(`*parameters*`)`
  $\equiv$ *CompanionObject*`.apply(`*parameters*`)`
- method `unapply` that enables constructor in any `case` clause (hence the name of this type of class)

Case class $\equiv$ class $+ \ldots$

`val` modifier implicitly applied to any class parameter

- **Any parameter is actually a public immutable member.**
- == compares content, not reference
- `toString` method is updated accordingly

Companion object is implicitly created with :

- method `apply` with same parameters as class
  $\Longrightarrow$ keyword `new` is no longer required :
  CompanionObject(*parameters*)
  $\equiv$ *CompanionObject*.`apply`(*parameters*)
- method `unapply` that enables constructor in any `case` clause (hence the name of this type of class)

Hint : Explicit object declaration implicity includes all properties

Case class

Example

Case class

Example

```
scala> case class Person(last : String, first : String)
defined class Person
```

Case class

Example

```
scala> case class Person(last : String, first : String)
defined class Person

scala> val rdl = Person("Dujol", "Romain")
rdl: Person = Person(Dujol,Romain)
```

Case class

Example

```
scala> case class Person(last : String, first : String)
defined class Person

scala> val rdl = Person("Dujol", "Romain")
rdl: Person = Person(Dujol,Romain)

scala> rdl == Person("Dujol", "Romain")
res0: Boolean = true
```

Case class

Example

```
scala> case class Person(last : String, first : String)
defined class Person

scala> val rdl = Person("Dujol", "Romain")
rdl: Person = Person(Dujol,Romain)

scala> rdl == Person("Dujol", "Romain")
res0: Boolean = true

scala> rdl.first + " " + rdl.last
res1: String = Romain Dujol
```

Case class

Example

```
scala> case class Person(last : String, first : String)
defined class Person

scala> val rdl = Person("Dujol", "Romain")
rdl: Person = Person(Dujol,Romain)

scala> rdl == Person("Dujol", "Romain")
res0: Boolean = true

scala> rdl.first + " " + rdl.last
res1: String = Romain Dujol

scala> Person("Romain", "Dujol") match {
           case Person("Dujol", "Romain") => "bibi"
           case Person("Dujol",  _      ) => "famille"
           case Person(l, f) if l == f    => "Ca existe ?"
           case _                         => "autre"
         }
res2: String = autre
```

Classes

Sealed class

Sealed class

Standard use case

- Class is abstract.
- Any subtype is a case class.

Sealed class

Standard use case

- Class is abstract.
- Any subtype is a case class.

Exhaustivity in a pattern matching may be checked

- Indeed, any instance class must be one of the subtypees.

Sealed class and case classes

Example : integer-valued option

Sealed class and case classes

Example : integer-valued option

```
scala> sealed abstract class IntOption
defined class IntOption
```

Sealed class and case classes

Example : integer-valued option

```
scala> sealed abstract class IntOption
defined class IntOption

scala> case class IntSome(x : Int) extends IntOption
defined class IntSome
```

Sealed class and case classes

Example : integer-valued option

```
scala> sealed abstract class IntOption
defined class IntOption

scala> case class IntSome(x : Int) extends IntOption
defined class IntSome

scala> case object IntNone extends IntOption
defined class IntNone
```

Sealed class and case classes

Example : integer-valued option

```
scala> sealed abstract class IntOption
defined class IntOption

scala> case class IntSome(x : Int) extends IntOption
defined class IntSome

scala> case object IntNone extends IntOption
defined class IntNone

scala> val test : IntOption = IntNone
res0: IntOption = IntNone

scala> test match { case IntNone => 0 }
<console>:14: warning: match may not be exhaustive.
It would fail on the following input: IntSome(_)
       test match { case IntNone => 0 }
       ^
res1: Int = 0
```

*(Warning does not appear if class **`IntOption`** is not sealed.)*

Traits

Traits

A trait is like a Java interface. . .

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`
- No constructor

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`
- No constructor
- Any class/trait may implement any number of traits.

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`
- No constructor
- Any class/trait may implement any number of traits.

... but

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`
- No constructor
- Any class/trait may implement any number of traits.

... but

- Trait may contain values/variables.

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`
- No constructor
- Any class/trait may implement any number of traits.

... but

- Trait may contain values/variables.
- Trait body may contain partially implemented bits.

Traits

A trait is like a Java interface...

- No parameters *(parameters will be available in Scala 3)*
- Abstract, i.e. no direct instantiation with `new`
- No constructor
- Any class/trait may implement any number of traits.

... but

- Trait may contain values/variables.
- Trait body may contain partially implemented bits.
- Trait may be implemented by a sole instance (not the whole class).

Traits

Default declaration syntax

```
trait TraitName[ParameterTypes*]
  {
    traitBody
  }
```

Traits

Default declaration syntax

```
trait TraitName[ParameterTypes*]
  {
    traitBody
  }
```

- Usually in *TraitName*`.scala`

Traits

Default declaration syntax

```
trait TraitName[ParameterTypes*]
  {
    traitBody
  }
```

- Usually in *TraitName*`.scala`
- *TraitName* must start with an **uppercase** letter.

Traits

Default declaration syntax

```
trait TraitName[ParameterTypes*]
  {
    traitBody
  }
```

- Usually in *TraitName*`.scala`
- *TraitName* must start with an **uppercase** letter.
- *traitBody* contains **`val`**/**`var`** and **`def`** *signatures* (at least)

Traits

Comparable.scala : (Java style)

```
trait Comparable[T]
  {
    def == (that : T)
    def != (that : T)
    def <  (that : T)
    def <= (that : T)
    def >  (that : T)
    def >= (that : T)
  }
```

- Any implementing class must define all methods.

Traits : partial implementation

Comparable.scala : (Scala style)

```
trait Comparable[T]
  {
    def == (that : T)
    def != (that : T) = !(this == that)
    def <  (that : T)
    def <= (that : T) =  (this <  that) || (this == that)
    def >  (that : T) = !(this <= that)
    def >= (that : T) = !(this <  that)
  }
```

- Any implementing class only needs to define == and <.

Trait : values/variables

Example

Trait : values/variables

Example

```
scala> trait Person {
           val  last : String
           val first : String
         }
defined trait Person
```

Trait : values/variables

Example

```
scala> trait Person {
           val  last : String
           val first : String
         }
defined trait Person

scala> case class Employee(      last : String,
                                first : String,
                           department : String) extends Person
defined class Employee
```

Trait : values/variables

Example

```
scala> trait Person {
           val  last : String
           val first : String
         }
defined trait Person

scala> case class Employee(      last : String,
                                first : String,
                           department : String) extends Person
defined class Employee

scala> val rdl : Person = Employee("Dujol", "Romain", "Mathematics")
rdl: Person = Employee(Dujol,Romain,Mathematics)
```

Trait : values/variables

Example

```
scala> trait Person {
           val  last : String
           val first : String
         }
defined trait Person

scala> case class Employee(      last : String,
                                first : String,
                           department : String) extends Person
defined class Employee

scala> val rdl : Person = Employee("Dujol", "Romain", "Mathematics")
rdl: Person = Employee(Dujol,Romain,Mathematics)

scala> rdl.last
res0: String = Dujol
```

Trait : implementation by an instance

Example

Trait : implementation by an instance

Example

```
scala> trait Employee {
           val department : String
         }
defined trait Employee
```

Trait : implementation by an instance

Example

```
scala> trait Employee {
           val department : String
         }
defined trait Employee

scala> case class Person(last : String, first : String)
defined class Person
```

Trait : implementation by an instance

Example

```
scala> trait Employee {
           val department : String
         }
defined trait Employee

scala> case class Person(last : String, first : String)
defined class Person

scala> val rdl = Person("Dujol", "Romain") with Employee {
                     override val department = "Mathematics"
                   }
rdl: Person with Employee = Person(Dujol,Romain)
```

Trait : implementation by an instance

Example

```
scala> trait Employee {
           val department : String
         }
defined trait Employee

scala> case class Person(last : String, first : String)
defined class Person

scala> val rdl = Person("Dujol", "Romain") with Employee {
                     override val department = "Mathematics"
                   }
rdl: Person with Employee = Person(Dujol,Romain)

scala> rdl.department
res0: String = Mathematics
```

Generics

Generics

Bounding

Generic classes/traits

As for methods, type may be infered when used. . .

Generic classes/traits

As for methods, type may be infered when used. . .

- . . . but it may not be always recommended.

Generic classes/traits

As for methods, type may be infered when used. . .

- . . . but it may not be always recommended.

Bounding types parameters

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1
- **Lower bound** : `class A[T >: T0]`
  T0 must extend (or implement) T

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1
- **Lower bound** : `class A[T >: T0]`
  T0 must extend (or implement) T

Both on the same is possible : `class A[T <: T1 >: T0]`

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1
- **Lower bound** : `class A[T >: T0]`
  T0 must extend (or implement) T

Both on the same is possible : `class A[T <: T1 >: T0]`

Why ?

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1
- **Lower bound** : `class A[T >: T0]`
  T0 must extend (or implement) T

Both on the same is possible : `class A[T <: T1 >: T0]`

Why ?

- Upper bound may be used to ensure that any element of type T has some properties (namely, the ones defined by T1).

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1
- **Lower bound** : `class A[T >: T0]`
  T0 must extend (or implement) T

Both on the same is possible : `class A[T <: T1 >: T0]`

Why ?

- Upper bound may be used to ensure that any element of type T has some properties (namely, the ones defined by T1).
- Lower bound may be used for contravariant type parameters (e.g. actual definition of methods on collections).

Generic classes/traits

As for methods, type may be infered when used...

- ... but it may not be always recommended.

Bounding types parameters

- **Upper bound** : `class A[T <: T1]`
  T must extend (or implement) T1
- **Lower bound** : `class A[T >: T0]`
  T0 must extend (or implement) T

Both on the same is possible : `class A[T <: T1 >: T0]`

Why ?

- Upper bound may be used to ensure that any element of type T has some properties (namely, the ones defined by T1).
- Lower bound may be used for contravariant type parameters (e.g. actual definition of methods on collections).

Also works with type parameters in methods.

Generics

Variance

Let's start with a basic example

Example

Let's start with a basic example

Example

```
scala> class A[T]
defined class A
```

Let's start with a basic example

Example

```
scala> class A[T]
defined class A

scala> val a = new A[Int]()
a: A[Int] = A@12448de1
```

Let's start with a basic example

Example

```
scala> class A[T]
defined class A

scala> val a = new A[Int]()
a: A[Int] = A@12448de1

scala> val a0 : A[Any] = a
<console>:13: error: type mismatch;
 found   : A[Int]
 required: A[Any]
Note: Int <: Any, but class A is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       val a0 : A[Any] = a
                         ^
```

Any **`Int`** is an **`Any`** so that should work. . . right ?

Let's start with a basic example

Example

```
scala> class A[T]
defined class A

scala> val a = new A[Int]()
a: A[Int] = A@12448de1

scala> val a0 : A[Any] = a
<console>:13: error: type mismatch;
 found   : A[Int]
 required: A[Any]
Note: Int <: Any, but class A is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       val a0 : A[Any] = a
                         ^
```

Any **`Int`** is an **`Any`** so that should work. . . right ?

- Scala must be told **explicitly** that it should behave that way.

Let's start with a basic example

Example

```
scala> class A[T]
defined class A

scala> val a = new A[Int]()
a: A[Int] = A@12448de1

scala> val a0 : A[Any] = a
<console>:13: error: type mismatch;
 found   : A[Int]
 required: A[Any]
Note: Int <: Any, but class A is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       val a0 : A[Any] = a
                         ^
```

Any **Int** is an **Any** so that should work. . . right ?

- Scala must be told **explicitly** that it should behave that way.
- Error message give a hint : put +T instead of T.

Covariance

By default, any type parameter is **invariant**.

Covariance

By default, any type parameter is **invariant**.

- Adding + before makes it **covariant** : `class A[+T]`.

Covariance

By default, any type parameter is **invariant**.

- Adding + before makes it **covariant** : `class A[+T]`.
- Then, if T is a subtype of U, then `A[T]` is a subtype of `A[U]`.

Covariance

By default, any type parameter is **invariant**.

- Adding + before makes it **covariant** : `class A[+T]`.
- Then, if T is a subtype of U, then `A[T]` is a subtype of `A[U]`.

Example (corrected)

Covariance

By default, any type parameter is **invariant**.

- Adding + before makes it **covariant** : `class A[+T]`.
- Then, if T is a subtype of U, then `A[T]` is a subtype of `A[U]`.

Example (corrected)

```
scala> class A[+T]
defined class A
```

Covariance

By default, any type parameter is **invariant**.

- Adding + before makes it **covariant** : `class A[+T]`.
- Then, if T is a subtype of U, then `A[T]` is a subtype of `A[U]`.

Example (corrected)

```
scala> class A[+T]
defined class A

scala> val a = new A[Int]()
a: A[Int] = A@134abd78
```

Covariance

By default, any type parameter is **invariant**.

- Adding + before makes it **covariant** : `class A[+T]`.
- Then, if T is a subtype of U, then `A[T]` is a subtype of `A[U]`.

Example (corrected)

```
scala> class A[+T]
defined class A

scala> val a = new A[Int]()
a: A[Int] = A@134abd78

scala> val a0 : A[Any] = a
a0: A[Any] = A@134abd78
```

Covariance : class **Option**

Here how is (roughly) defined **Option**

```
sealed abstract class Option[+T]

case class  Some[+T](x : T) extends Option[T]
case object None            extends Option[Nothing]
```

Covariance : class **Option**

Here how is (roughly) defined **Option**

```
sealed abstract class Option[+T]

case class  Some[+T](x : T) extends Option[T]
case object None            extends Option[Nothing]
```

Why such a declaration for **None** works ?

Covariance : class **Option**

Here how is (roughly) defined **Option**

```
sealed abstract class Option[+T]

case class  Some[+T](x : T) extends Option[T]
case object None            extends Option[Nothing]
```

Why such a declaration for **None** works ?

1. *Reminder* : **Nothing** is a subtype of any type T.

Covariance : class **Option**

Here how is (roughly) defined **Option**

```
sealed abstract class Option[+T]

case class  Some[+T](x : T) extends Option[T]
case object None            extends Option[Nothing]
```

Why such a declaration for **None** works ?

1. *Reminder* : **Nothing** is a subtype of any type T.
2. So **Option**[**Nothing**] is a subtype of any type **Option**[**T**].

Covariance : class **Option**

Here how is (roughly) defined **Option**

```
sealed abstract class Option[+T]

case class  Some[+T](x : T) extends Option[T]
case object None            extends Option[Nothing]
```

Why such a declaration for **None** works ?

1. *Reminder* : **Nothing** is a subtype of any type T.
2. So **Option**[**Nothing**] is a subtype of any type **Option**[**T**].
3. By transitivity, **None** is a subtype of any type **Option**[**T**].

Contravariance

Contravariant type parameter

Contravariance

Contravariant type parameter

- Adding - before makes it **contravariant** : `class A[-T]`.

Contravariance

Contravariant type parameter

- Adding `-` before makes it **contravariant** : `class A[-T]`.
- Then, if T is a subtype of U, then `A[U]` is a subtype of `A[U]`.

Contravariance

Contravariant type parameter

- Adding `-` before makes it **contravariant** : `class A[-T]`.
- Then, if `T` is a subtype of `U`, then `A[U]` is a subtype of `A[U]`.

So mechanism is *reversed*, compared to covariance.

Contravariance

Contravariant type parameter

- Adding `-` before makes it **contravariant** : `class A[-T]`.
- Then, if T is a subtype of U, then `A[U]` is a subtype of `A[U]`.

So mechanism is *reversed*, compared to covariance.

Why bother with contravariance ?

- Let us consider the type `T => R`, also denoted `Function1[T, R]`.
- Then the right variance is `Function1[-T, +R]`, i.e. :

Contravariance

Contravariant type parameter

- Adding `-` before makes it **contravariant** : `class A[-T]`.
- Then, if T is a subtype of U, then `A[U]` is a subtype of `A[U]`.

So mechanism is *reversed*, compared to covariance.

Why bother with contravariance ?

- Let us consider the type `T => R`, also denoted `Function1[T, R]`.
- Then the right variance is `Function1[-T, +R]`, i.e. :
  - if T1 is a subtype of T2 ;
  - if R1 is a subtype of R2 ;

  `Function1[T2, R1]` is a subtype of `Function1[T1, R2]`.

LISKOV substitution principle

Actual formulation (Barbara LISKOV, 1994)

Let P be a provable property on objects of type U.
Then P should be provable on objects of any subtype of U.

LISKOV substitution principle

Actual formulation (Barbara LISKOV, 1994)

Let *P* be a provable property on objects of type U.
Then *P* should be provable on objects of any subtype of U.

Reformulation (Martin ODERSKY, Coursera online course)

If T is a subtype of U, then :

LISKOV substitution principle

Actual formulation (Barbara LISKOV, 1994)

Let *P* be a provable property on objects of type U.
Then *P* should be provable on objects of any subtype of U.

Reformulation (Martin ODERSKY, Coursera online course)

If T is a subtype of U, then :

- everything one can do with a value of type U,

LISKOV substitution principle

Actual formulation (Barbara LISKOV, 1994)

Let *P* be a provable property on objects of type U.
Then *P* should be provable on objects of any subtype of U.

Reformulation (Martin ODERSKY, Coursera online course)

If T is a subtype of U, then :

- everything one can do with a value of type U,
- one should also be able to so do with a value of type T.

LISKOV substitution principle

Actual formulation (Barbara LISKOV, 1994)

Let *P* be a provable property on objects of type U.
Then *P* should be provable on objects of any subtype of U.

Reformulation (Martin ODERSKY, Coursera online course)

If T is a subtype of U, then :

- everything one can do with a value of type U,
- one should also be able to so do with a value of type T.

Application to following property

For any element of `Function1[T, R]`, one can :

- take an input of type T ;
- get a result of type R.

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 :

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;

LISKOV substitution principle

Let's check !

- For any element of **`Function1`**`[`**`T1`**`,` **`R2`**`]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of **`Function1`**`[`**`T2`**`,` **`R1`**`]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 :

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 : *indeed, result has type R1*.

LISKOV substitution principle

Let's check !

- For any element of **`Function1[T1, R2]`**, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of **`Function1[T2, R1]`**, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 : *indeed, result has type R1*.

So **`Function1[T2, R1]`** is a subtype of **`Function1[T1, R2]`**.

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 : *indeed, result has type R1*.

So `Function1[T2, R1]` is a subtype of `Function1[T1, R2]`.

Consequences

When defining methods in a class with type parameters :

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 : *indeed, result has type R1*.

So `Function1[T2, R1]` is a subtype of `Function1[T1, R2]`.

Consequences

When defining methods in a class with type parameters :

- covariant type parameters can only appear as **result types** ;

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 : *indeed, result has type R1*.

So `Function1[T2, R1]` is a subtype of `Function1[T1, R2]`.

Consequences

When defining methods in a class with type parameters :

- covariant type parameters can only appear as **result types** ;
- contravariant type parameters can only appear as **types of input parameters** ;

LISKOV substitution principle

Let's check !

- For any element of `Function1[T1, R2]`, one can :
  - take an input of type T1 ;
  - get a result of type R2.
- For any element of `Function1[T2, R1]`, one can :
  - take an input of type T1 : *indeed, input will have type T2* ;
  - get a result of type R2 : *indeed, result has type R1*.

So `Function1[T2, R1]` is a subtype of `Function1[T1, R2]`.

Consequences

When defining methods in a class with type parameters :

- covariant type parameters can only appear as **result types** ;
- contravariant type parameters can only appear as **types of input parameters** ;
- invariant type parameters may appear anywhere.