Advanced Programming II

Lesson 1: Introduction to Scala

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Contents

- First steps in Scala.
- Classes and objects.
  - Packages, import and export.
- Values, variables and control structures.
- Lists, tuples, sets and maps.
- Inheritance.
- Scala's hierarchy.
- Assertions.

First steps in Scala

- **Scala**: scalable language.
  - It is a programming language created in 2003 by Martin Ordesky.
  - It mixes object-oriented and functional programming concepts in a statically typed language.
  - It is designed to grow according to the demands of users. It can be used to write anything, from small scripts to large systems.

Programming paradigms

- Basic programming paradigms are based on different computational models and determine the most basic constructions of a program:
  - **Imperative paradigm**: programmer writes how a calculus has to be performed. Describes programming in terms of a state and statements that change such a state.
  - **Declarative paradigm**: programmer writes what has to be calculated. As opposed to the imperative, is based on describing the problem or solution with no explicit algorithm.

Main features of Scala

- **Scala** is...
  - **Object-oriented (imperative).**
    - Each value is an object, and each operation is a call to a method.
    - Allows mutable data and side effects on calls.
  - **Functional (declarative).**
    - Every function is a value, every value is an object, so a function is also an object.
    - It allows immutable data and referentially transparent methods.
  - **Statically typed.**
    - Each variable is type-bound at compile time. Since its declaration, a variable is associated with only one value of that type.
    - Scala is a language with a very advanced static type system. Allows:
      - Parameterization of types with *generics*.
      - Combination of types using *intersections*.

Main features of Scala

- Scala is...
  - Compatible with Java.
    - Compiles into bytecodes and it is executed by the JVM (Java Virtual Machine).
    - Reuses Java types and classes, calls internally to Java methods, etc.
  - Concise.
    - It has a wide syntax that allows writing programs with shorter code.
    - It is complex, as the same sentence can be written using different syntaxes.
  - Not backward compatible.
    - Some newer versions of Scala are not compatible with older ones.
  - REPL enabled.
    - Provides a console to test short sentences and programs.

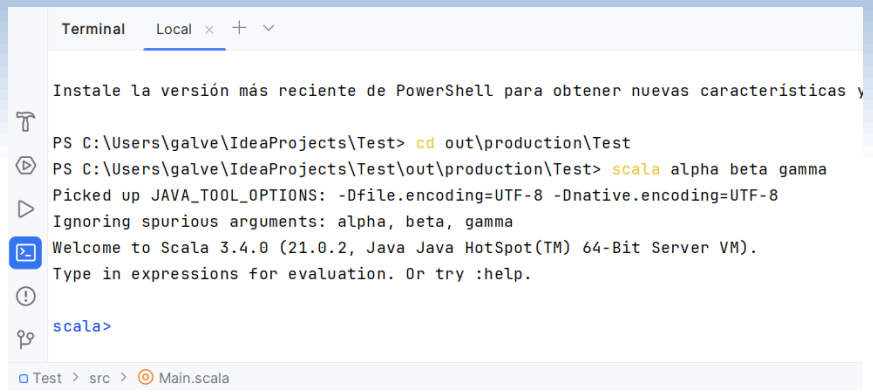
First program in Scala

```
object Main {
  def main(args: Array[String]): Unit = {
    println("Hello friend!")
  }
}
```

- Declares a class and an **object** of it.
- **def** is used to declare a function.
- Types are **after** the variable's name.
- **Unit** refers to a piece of code with no returning value.
- Some objects are used by default, as **System.out**.

First program in Scala

- Like in Java:
  - Use **scalac** to compile.
  - Use **scala** to execute.
- Same result with:



Inherits the method main from App

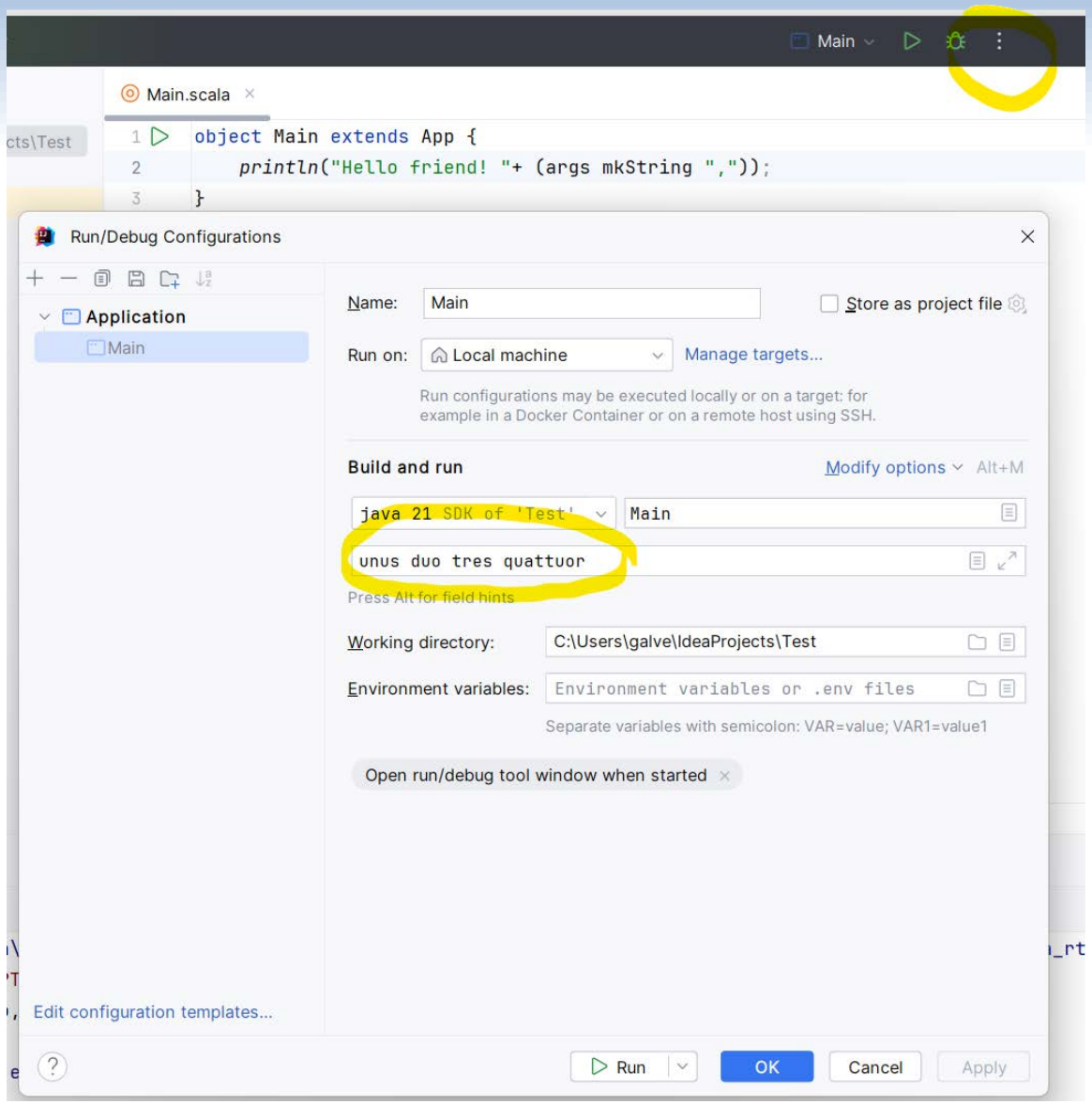
```
object Main extends App {
   println("Hello friend! “+ (args mkString ","));
}
```

`args` is the command line as an array

`mkString` converts a list into a String separating ítems with a char

- App is a **trait** (like a Java's interface)

Configuration in IntelliJ



val and var

- Scala tries to use only immutable values:
  - val
    - Defines an immutable variable whose value can never change.
    - Equivalent to **final** in Java.
  - var
    - Defines a traditional variable.

```
object Main {
  def main(args: Array[String]): Unit = {
    val c = 0;
    val d : Int = 4;
    var x = 1;
    var y : Float = 3.1F;
    var z : Double = 3.14159; // Double by default
    c = 2; // Error
  }
}
```

Easy to use associative arrays

- **Scala** simplifies syntax.
- Types may be inferred (better declare them).

```
object Main {
  def main(args: Array[String]): Unit = {
    val capital = Map(
        "Spain" -> "Madrid",
        "Russia" -> "Moskow",
        "Germany" -> "Berlin"
    );
    val country = "Spain";
    println(capital(country));
  }
}
```

Declaration of functions

- Functions are declared similarly to Pascal.
- The last calculus is returned (return is suggested).

```
object Main {
  def factorial(x: BigInt): BigInt = {
    if (x==0) 1 else x * factorial(x-1);
  }
  def main(args: Array[String]): Unit = {
   println (factorial(13))
  }
}
```

- Semicolon is optative.
- Unit is the class for a block of code.

Classes uses val and private

- **Scala** is concise:
  - **val** is used by default in params (private) and fields.

```
class Person (name: String, dni: Int) {
  def getName() = name;
  def getDni() = dni;
  def appendToName(text: String): Unit = {
    name = name + text;
  }
}

object Main extends App {
  val paco = new Person("Paco", 333);
  var pepe = new Person("Pepe", 233);
  println(paco.name);
  println(paco.getName());
}
```

Data types

All data types in **Scala** are true objects, and there are no primitive types like in Java. Hence, e.g., you may send a message to an int:

Basic type	Range
`Byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`Short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`Int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`Long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`Char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`String`	a sequence of `Char`s
`Float`	32-bit IEEE 754 single-precision float
`Double`	64-bit IEEE 754 double-precision float
`Boolean`	`true` or `false`

Expressions

- **Scala** has the same syntax than Java for most expressions.

Operator	Type	Associativity
[] -> .	Binary	Left to right
! ~ - *	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
&&	Binary	Left to right
||	Binary	Left to right

NOTE: this table does not include all Scala's operators

```
val c = {
  val i1 = 2
  val j1 = 4/i1
  i1 * j1
 }
println(c)
/* Output: 4 */
```

The value of a block is the one of its last expression

```
val c = { val a = 11; a + 42 }
// c: Int = 53
```

a is an immutable variable with a local scope.

Scala 3 syntax. Semicolon

- **Semicolon inference**
  - Precise rules for statement separation are simple.
  - A line ending is treated as a semicolon unless:
    - The line ends in an illegal token, such as a period or an infix operator.
    - The next line begins with a token that cannot start a legal sentence (including an expression).
    - The line ends inside parenthesis or square brackets () [].
- **Scala 3 syntax**
  - Scala 3 allows syntax based on indentation like in Python.
  - Scala 3 enforces some rules on indentation and allows some occurrences of braces {...} to be optional:
    - First, some badly indented programs are flagged with warnings.
    - Second, some occurrences of braces {...} are made optional. Generally, the rule is that adding a pair of optional braces will not change the meaning of a well-indented program.
    - These changes can be turned off with the compiler flag -no-indent.
  - Be careful with such a syntax.

Basic input/output

- Console output (`scala.Console extends scala.io.AnsiColor`):
  - `print(expression)`
  - `println(expression)`
  - `printf(String, expression1, expression2, …)`
- Console input (`scala.io.StdIn`)
  - `def readLine():String` reads a string from the keyboard
  - `def readInt():Int` reads an integer from the keyboard
  - `def readDouble():Double` reads a decimal value from the keyboard
- Reading from a file:

```
import scala.io.Source

object Main extends App {
   for(line <- Source.fromFile("example.txt").getLines())
      println(line)
   // Functional abilities
   Source.fromFile("example.txt").getLines().foreach(println)
}
```

String interpolation

- String interpolation provides a way to use variables inside strings. For instance:
- While it may seem obvious, it's important to note here that string interpolation will not happen in normal string literals.
- Any arbitrary expression can be embedded in **${}**.
- Letter **s** allows interpolation through **$**, whereas letter **f** allows interpolation as in **printf**. **Raw** interpolation cancels the meaning of the escape **** character.

```
val name = "James"
val height = 1.9d
val age = 30
println(s"$name is $age years old") // "James is 30 years old"
println("$name is $age years old")  // "$name is $age years old"
println(f"$name%s is $height%2.2f meters tall")  // "James is 1.90 meters tall"
println("a\nb")
println(raw"a\nb")
```

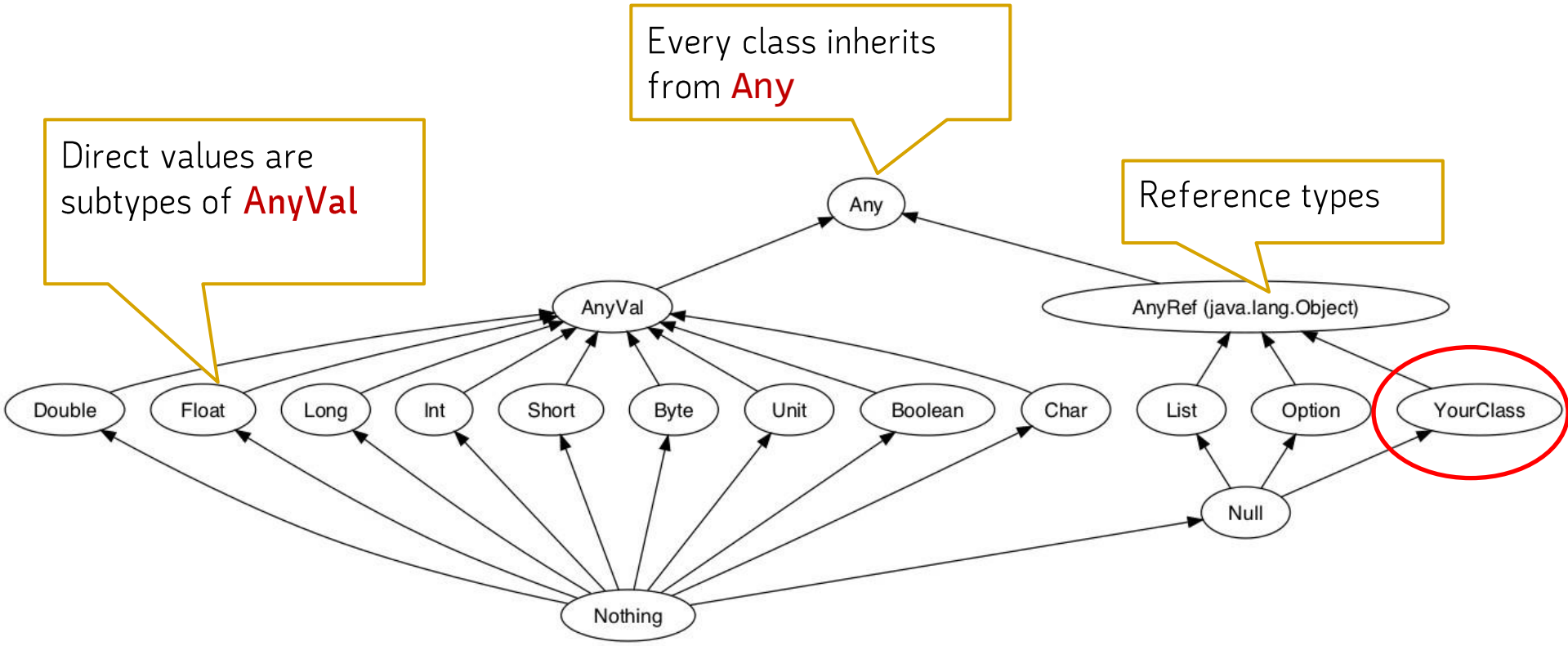
Class AnsiColor

- The class AnsiColor allows rich outputs.
- E.g.:

```
import Console.{GREEN, RED, RESET, YELLOW_B, UNDERLINED}

object PrimeTest:
  def isPrime(): Unit =
    val candidate = io.StdIn.readInt().ensuring(_ > 1)
    val prime = (2 to candidate - 1).forall(candidate % _ != 0)
    if (prime)
      Console.println(s"${RESET}${GREEN}yes${RESET}")
    else
      Console.err.println(s"${RESET}${YELLOW_B}${RED}${UNDERLINED}NO!${RESET}")

  def main(args: Array[String]): Unit = isPrime()
```

Classes and objects

- **Scala** has the next structure of basic classes:



Classes

- **Unit** is a class that defines procedures in contrast to functions.
- Parenthesis may be omitted in classes without params.
- **Many syntaxes available:**

```
class ChecksumAccumulator:
  private var sum = 0;
  def add(b: Byte): Unit =
    sum += b;
  def checksum =
    ~(sum & 0xFF) + 1
```

```
class ChecksumAccumulator() {
    private var sum = 0;
    def add(b: Byte): Unit = {
        sum += b;
    };
    def checksum(): Int = {
        return ~ (sum & 0xFF) + 1;
    }
}
```

```
class ChecksumAccumulator() {
    private var sum = 0
    def add(b: Byte) = sum += b
    def checksum = ~(sum & 0xFF) + 1
}
```

Classes

- A class declaration may have parameters that, in turn, are fields.
- By default, everything is **val**.

1

```
class Tiger(dangerous: Boolean, var age: Int)
```

Dangerous and **age** are private

dangerous and **age** are public
param1 and **param2** are private

```
class Tiger(param1: Boolean, param2: Int) {
   val dangerous = param1
   var age = param2
}
```

2

- Where you say public in Java, in Scala you say nothing: public is the default access level in Scala.
- Class' **parameters** are **private** but if declared as **var** or **val**

```
object Main extends App {
   var t = new Tiger(true, 6)
   println(t.age)
}
```

2

Constructors

- In a Scala program, the constructors other than the primary constructor are known as auxiliary constructors. we are allowed to create any number of auxiliary constructors in our program, but a program contains only one primary constructor.

  Syntax: **def this(...)**

- Details:
  - In a single program, we are allowed to create multiple auxiliary constructors, but they have different signatures or parameter-lists.
  - Every auxiliary constructor must call one of the previously defined constructors.
  - The invoke constructor may be a primary or another auxiliary constructor that comes textually before the calling constructor.
  - The first statement of the auxiliary constructor must contain the constructor call using this.

Singleton pattern

- This pattern creates a single object for a class.
- It uses the keyword **object** instead of **class**.
- When the singleton object shares its name with the class is called companion class, and the object is called class's companion object.

```
object Main extends App {
  checksumAccumulator.calculate("Novosibirsk")
  println(checksumAccumulator.checksum());
}
```

```
// In class checksumAccumulator.scala
// By default, collections are immutable
import scala.collection.mutable;
object checksumAccumulator {
  private var sum = 0;
  private val cache = mutable.Map.empty[String, Int];
  def add(b: Byte): Unit = {
    sum += b;
  };
  def checksum(): Int = {
    return ~ (sum & 0xFF) + 1;
  }
  def calculate(s: String): Int = {
    if (cache.contains(s))
      return cache(s);
    else {
      sum = 0;
      for (c <- s) {
        add((c >> 8).toByte);
        add(c.toByte);
      }
      val cs = checksum();
      cache += (s -> cs);
      return cs;
    }
  }
}
```

update and apply

- Generics are specified in square brackets:

```
val greetings = new Array[String](3)
```

- Parenthesis are used to access items:

```
greetings(0) = "Hello";
```

- When you apply parenthesis surrounding 0, 1 or more expressions to a variable v, Scala transforms into a call to the **apply** function.
- Similarly happens with assignments and **update**.

```
println(greetings(0));   ⟺   println(greetings.apply(0));
```

```
greetings(0) = "Hola";   ⟺   greetings.update(0, "Hola");
```

Companion object

- Scala has no static components in a class.
- However, you may use a companion object to achieve the same results.
- In this case, the file must be named as its class.
- A companion object in Scala is an object that's declared in the same file as a class and has the same name as the class.
- A companion object and its class can access each other's private members (fields and methods).
- This can be used, e.g.
  to avoid usage of keyword **new**.

```
object Main extends App {
  var p = Person("Igor")
  println(p.name);
}
```

No **new** is needed

```
class Person {
  var name = ""
}

object Person {
  def apply(name: String): Person = {
    var p = new Person
    p.name = name
    p
  }
}
```

Factory pattern

Companion object

- **apply** allows creating several constructors.
- Just as adding an apply method in a companion object lets you construct new object instances, adding an **unapply** lets us de-construct object instances.
- We can write **unapply** to return anything. Here's an example that returns the two fields in a tuple:

```
class Person {
  var name = "";
  var age = -1;
}

object Person {
  def apply(name: String): Person = {
    var p = new Person;
    p.name = name;
    p
  }
  def apply(name: String, age: Int): Person = {
    val p = Person(name);
    p.age = age;
    p
  }
  def unapply(p: Person): (String, Int) = (p.name, p.age)
}
```

```
object Main extends App {
    var p = Person("Igor", 33)
    println(Person.unapply(p));
}
```

Constructors

- **apply** allows creating several constructors.
- Constructors can also be created using **this**.

```
class Person(val name:String, var age:Int) {
 // variables need to be initialized to be defined
 var dni:BigInt = 0;
 var dniLetter: Char = '-';
 var salary: Int = 0;
 def this(name:String, age:Int, salary: Int) = {
  this(name, age);
  this.salary = salary;
 }
 def this(name:String, age:Int, salary: Int, dniComplete:String) = {
  this(name, age, salary);
  this.salary = salary;
  this.dniLetter = (dniComplete takeRight 1).head;
  this.dni = (dniComplete dropRight 1).toInt;
 }
}
```

Packages, import and export

- Scala is very rich in package management.
- Scala may manage packages in the same way than Java.
- Scala also allows the use of a **package** like a C# namespace. In this case, you have to put between curly brackets what is affected by the package.
- **_root_** is the root package.

```
package roscosmos {
  package navigation {
    class Navigator { // In package roscosmos.navigation
      // No need to say roscosmos.navigation.StarMap
      val map = new StarMap
    }
    class StarMap // In package roscosmos.navigation
    package launch {
      class Booster // In package roscosmos.navigation.launch
    }
  }
  class Ship { // In package roscosmos
    // No need to say roscosmos.navigation.Navigator
    val nav = new navigation.Navigator
  }
  package fleets {
    class Fleet { // In package roscosmos.fleets
      // No need to say roscosmos.fleets.Fleet
      def addShip = new Ship
    }
  }
}
```

import

- Scala is very rich and flexible when importing classes.
- **import** clause may appear anywhere.
- May refer to packages, classes, singleton and objects.
- Let us to rename and hide some of the imported members
- Let's assume the next declarations:

```
package delights

abstract class Fruit(val name: String, val color: String)

object Fruits {
   object Apple extends Fruit("apple", "red")
   object Orange extends Fruit("orange", "orange")
   object Pear extends Fruit("pear", "yellowish")
   val menu = List(Apple, Orange, Pear)
}
```

import

- Some imports may be:

```
import delights.Fruit
import delights.*
import delights.Fruits.*
import delights.Fruits.{Apple, Orange}
import delights.Fruits.{Pear => Conferencia, Orange} // renames Pear
import delights.Fruits.{Apple => _, Orange} // Excludes Apple
```

- All contents using **`*`**
- Renaming: **`a => alias`** or **`a as alias`**
- Excluding: **`a => _`** or **`a as _`**
- Explicit set using **`{}`**

Default import. export

- Scala adds some imports automatically:

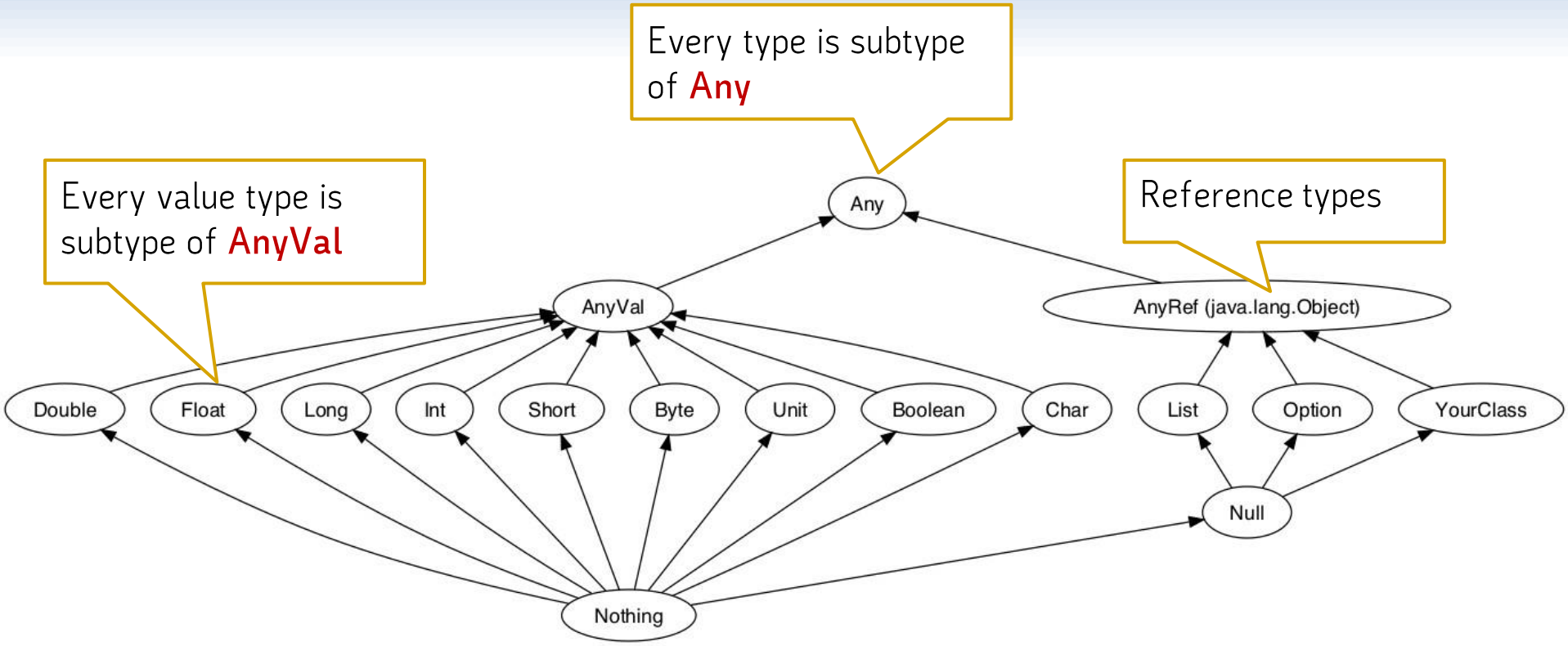
```
import java.Lang.*
import scala.*
import Predef.* // Everything in the Predef object
```

- **Scala encourages composition over inheritance**.
- Keyword **export** may be used to facilitate composition by "forwarding" the required methods.
- E.g., if you want to create a class for positive integers you may create it by composition and "inheriting" only the required functions:

```
class PosInt(value: Int) {
  require(value >= 0)
  // PosInt "inherits" all the functions of the type of "value"
  // operator >> is renamed and operator << is removed
  export value.{>> as shr, << as _, *}
}
```

```
object Main extends App {
  val x = PosInt(23)
  println(x + 5)
  println(x shr 2)
  println(x << 2) // Error
}
```

Values, variables and control structures



Literals

- A literal is a direct value written in the source code.

Type	Example of literal
Int (hexadecimal)	0x5F
Int (decimal)	321
Long (postfix l or L)	35L
Float Double	0.0 1e30f 3.14159f 1.0e-100 .1
Boolean	true false
Char	'a' '\u0041' '\n' '\t'
String	"¡Hello \n world!" "Quotes \" in a string"
String multiline	"""string in three lines """

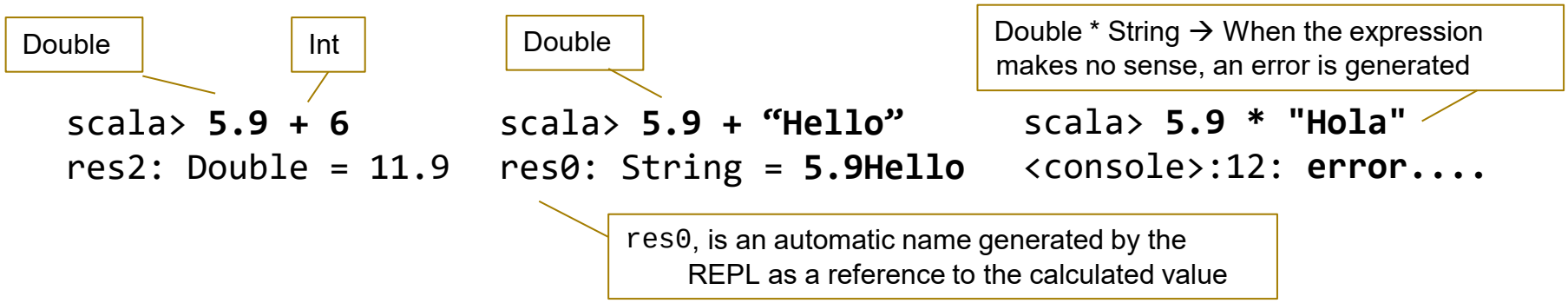
Remarks on val and var

- In Scala there are two types of variables:
  - **val** (values). Once initialized it cannot be reassigned (as final in Java). **Immutable**
  - **var** (variables). Can change along its life. **Mutable**

```
object Main extends App {
   val msg: String = "Hello comrades!"
   var mut : String = "Bye friends!"
   msg = "Alter string" // Error
   mut = "changed!"
}
```

Type inference

- Scala finds out the type for use based on its usage. This is known as **type inference**.



- When an initial value is assigned to a variable, the compiler can detect its type based on the value assigned to it.

```
val n = 1      ⟹  Int
val p = 1.2    ⟹  Double
```

Identifiers

- In Scala an identifier may be:
  - **Alphanumeric**. Start with a letter followed by letters and/or numbers.
    - **Underline char '_'** is considered like a letter.
    - Alphanumeric identifiers can also end with an underline, followed by some operator characters:

      ```
      msg    x2   x2cont_++
      ```

    - Spaces may be part of an identifier when used French accents:

      ```
      `identifier with spaces`
      ```

  - **Symbolic**. Starts with an operator character, optionally followed by more operator characters:

    ```
    +   **   +?%&
    ```


```
object Main extends App {
   val c_++ = 5
   val + : Int = 10
   val `id composed` = 12
   println(c_++)
   println(+)
   println(`id composed`)
}
```

Variables

- In Scala there are **three different scopes** for the variables depending on where they are used.
  - **Fields**
    - These are “variables” that belong to the state of an object.
    - They are always accessible from the methods inside the object and sometimes from outside it depending on the access modifiers with which it has been declared.
    - They can be mutable or immutable (**var** or **val**).
  - **Method parameters**
    - These are variables that are used to pass values to a method.
    - They are only accessible within the method but objects passed within the variable could be accessed from outside (if outside a reference to the object is available).
    - They are always immutable (**val**)
  - **Local variables**
    - These are variables declared within a method.
    - They are only accessible within the method.
    - They can be mutable or immutable (**var** or **val**).

Conditional sentences

- If the condition (logical expression) is true, the actions of the if are executed, if not those of the else block (can be omitted).
- In **Scala 3** you may **omit the parenthesis** in the condition and use the keyword **then**.
- Concatenated conditional statements can be used to evaluate more than one condition

```
if (condition) {
        action1
        ...
        actionN
} else {
        actionElse1
        …
        actionElseN
}
```

```
if condition then
        action1
        ...
        actionN
else
        actionElse1
        …
        actionElseN
```

```
if (condition) {
        action1
        ...
        actionN
} else if (condition2) {
        actionElse1
        …
        actionElseN
}
…
else {
}
```

Conditional expression. Equals

- Tip 1. An **if** sentence may be an expression in Scala.
- Tip 2. **==** in Scala behaves like **equals** in Java.
- The Java's **==** is achieved through **eq** in Scala.

```
object Main extends App {
   val radius = 1.9
   val shape = "rectangle"
   val length = 7
   val height = 5
   val area =  if (shape == "circle")
                  Math.PI * radius*radius
               else
                  length *height
   println(area)
}
```

Iteration sentences: while and for

- Scala uses the **while** keyword with the same syntax and meaning than in Java. (Positional with **do** keyword).
- **for** keyword is different: it is used only as a "**forEach**".
- The for construct (variable <- range) causes the variable to take all the values of the expression to the right of the arrow. The scope of the variable extends to the end of the loop.

The variable i takes all values between 1 and n (both included)

```
for (i <- 1 to n)
    r = r * i
```

The last value for i is n-1

```
for (i <- 1 until n)
    r = r * i
```

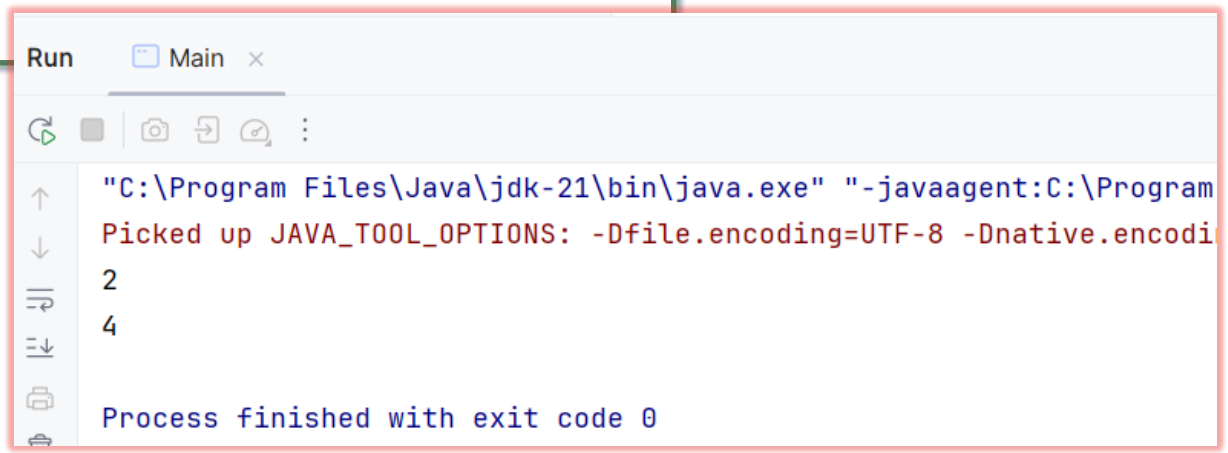
for sentence

- A **for** loop allows filtering items in a collection using one or more if statements.

```
for (var x  <- list if condition1; if condition2) {
        sentences
}
```

```
object Main extends App {
   for (a <- 1 to 10 if a<5 if a % 2 == 0)
      println(a)
}
```

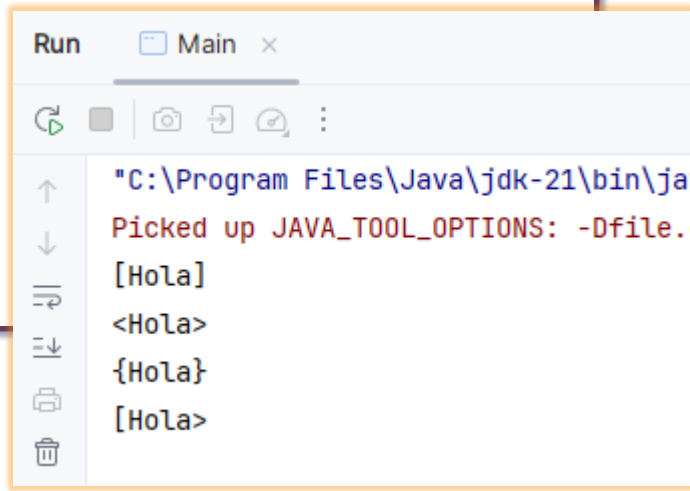
EXAMPLE: displays array even-numbered values greater than 5



Functions

- Functions are preceded by the keyword **def**.
- Its type may be inferred.
- The **return** keyword may be omitted: the last expression calculated is returned.
- Can be used **default parameters**. E.g.:

```
object Main extends App {
   def decorate(text: String, left: String = "[", right: String = "]") = left+text+right

   println(decorate("Hola"))
   println(decorate("Hola", "<", ">"))
   println(decorate(left="{", text="Hola", right="}"))
   println(decorate("Hola", right=">"))
}
```



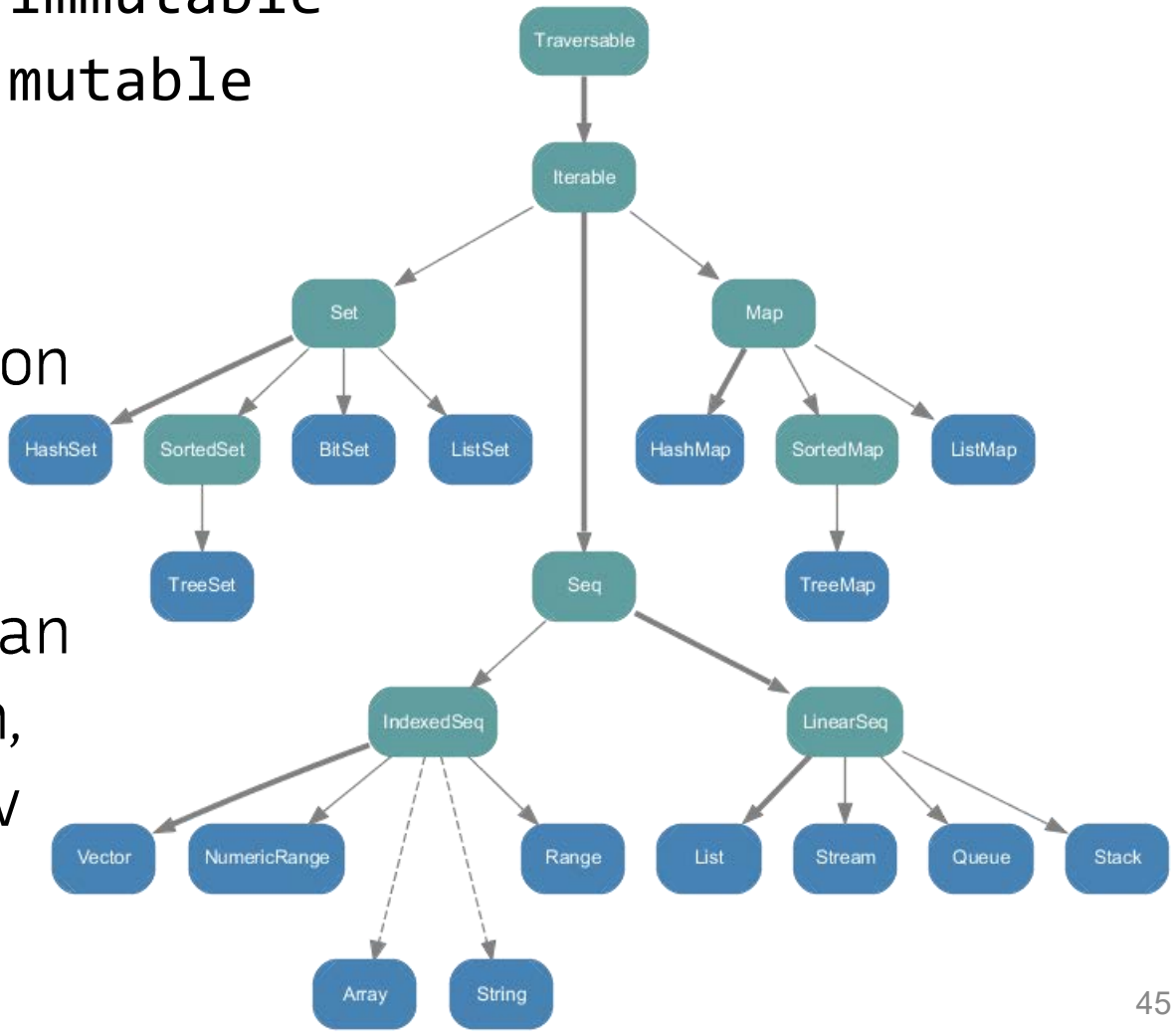
Functions

- * can be used to indicate a variable number of arguments.
- It must be the last argument.

```
object Main extends App {
   def sum(text:String, args:Int*): String = {
      var acc = 0
      for (arg <- args)
         acc += arg
      text + acc
   }

   println(sum("Result ", 1, 2, 3, 5, 6, 7))
}
```

Lists, tuples, sets and maps

- **Scala** has two types of collections (all generics):
  `scala.collection.immutable`
  `scala.collection.mutable`
- An immutable collection does not allow any modification on the list after initialized.
- If you try to modify an immutable collection, you will obtain a new object.



Collections. Lists

- Most collections implements **structural sharing**.
- This means that many operations have either a constant memory footprint or no memory footprint at all.
- **Generic type** is enclosed in square brackets [].

```
val numbersList: List[Int] = List(1, 2, 3 ,4)
val emptyList: List[Int] = List() // Empty List
val x = numbersList.head
val xs = numbersList.tail
```

- **Lists** are **immutable** and widely used in **functional notation**:

```
val numbersList: List[Int] = 1 :: 2 :: 3 :: 4 :: Nil  // List of Integers
val emptyList: List[Int] = Nil  // Empty List
val x :: xs = numbersList
assert(x == 1) // true
assert(xs == List(2, 3, 4)) // true
```

```
List(1,2) ::: List(3,4) // List(1, 2, 3, 4)
List.fill(3)(100) // List(100, 100, 100)
List(1,2,3,4).reverse // List(4, 3, 2, 1)
```

Collections. Array

- An **Array** is a fixed size **mutable** sequence.

Returns a reference to the created array that is stored in greetStrings

Declaration of an array that allows you to store three elements of type String

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello”
greetStrings(1) = ", ”
greetStrings(2) = "world!\n"
for (i <-0 to 2) print(greetStrings(i))
for (e <- greetStrings) print(e)
val greetStrings:Array[String] = new Array(3)
val greetStrings = Array (“Hello”, “, ”, “ world\n”)
```

Accessing an Item

Iterating over the items of the array: Can only be used for methods that need to query the elements. It does not allow to modify them.

Iterating an array using its indexes

Another way to declare the array

Creating and **initializing** an array. The type of the elements is inferred

Collections. Array

- There are two ways to declare **multidimensional** arrays.
  - Using array.ofDim:

```
val matrix = Array.ofDim[Int](2, 3)
matrix(0)(0) = 12
matrix(0)(1) = 23
```

  - Using array of arrays:

```
val matrix = Array(Array(1,2,3),Array(4,5,6))
```

- Iterating over a multidimensional array

```
val matrix = Array(Array(1,2,3),Array(4,5,6))
for (i <- 0 until 2) // excludes
   for (j <- 0 to 2) // includes
      println(matrix(i)(j))
```

Collections. Array

- Initialization using range:

```
val anArray = Array.range(10, 20, 2)
// anArray = Array[Int] = Array(10, 12, 14, 16, 18)
val otherArray = Array.range(10, 20)
// otherArray = Array[Int] = Array(10, 11, 12, 13,  14, 15,  16, 17,  18, 19)
```

- There are many methods in the Array class:

  https://www.scala-lang.org/api/current/scala/Array.html

```
val list = Array.range(1, 6)
val suma = list.sum
val product = list.product
val maximum = list.max
val minimum = list.min
val contains3 = list.contains(3)
```

List and Array buffers

- ListBuffer and ArrayBuffer are mutable collections.
- They are alike List and Array but they allow appending items at the end of the sequences.

- They use the operators **+=** and **+=:** to append (an alias for **append** and **prepend**, respectively).
- When you finish to create a ListBuffer you may convert it into immutable to work faster with it.

```
val buf = new ListBuffer[Int]
buf += 23
buf += 12
5 +=: buf
println(buf)
val list = buf.toList
println(list)
```

Tuples

- A **tuple** is an **immutable** object that stores elements, possibly of different types.
- It's like a C's struct but with a flexible syntax.

Create a tuple that contains an Int and a String. The inferred type for pair is **Tuple2[String, Int]** and depends on the number of elements.

```
val hostPort = ("localhost", 80)
println(hostPort._1)
println(hostPort._2)
```

The first element is accessed

The second element is accessed

```
val hostPort = ("localhost", 80)
println(hostPort(0))
println(hostPort(1))
```

Tuples

- A **tuple** is very convenient in many scenarios.
- Useful when returning several pieces of data.

```
object TestTuple {
   def mcd_mcm(x: Int, y: Int) = {
      var a = x
      var b = y
      while (a != b)
         if (a > b)
            a = a - b
         else
            b = b - a
      (a, x / a * y)
   }
}
```

```
object Main extends App {
   println(TestTuple.mcd_mcm(45, 60))
}
```

Remember

- Remember that **Scala 3** allows a syntax based on indentation.

```
object TestTuple:
   def mcd_mcm(x: Int, y: Int) =
      var a = x
      var b = y
      while a != b do
         if a > b then
            a = a - b
         else
            b = b - a
      (a, x/a*y)
```

```
object TestTuple {
   def mcd_mcm(x: Int, y: Int) = {
      var a = x
      var b = y
      while (a != b)
         if (a > b)
            a = a - b
         else
            b = b - a
      (a, x / a * y)
   }
}
```

String and StringOps

- **StringOps** implements many sequence methods.
- Predef package has an implicit conversion from String to StringOps, so a String can be managed as a sequence.

```
object Main extends App:
  def hasUpperCase(s: String) = s.exists(_.isUpper)
  println(hasUpperCase("Saint Petersbourg"))
  println(hasUpperCase("a little ladybug"))
```

- In this case the `exists` method does not exist for **String**.
- However, the compiler knows that it exists in the **StringOps** so an implicit conversion is performed.
- In addition, the code show an example of functional programming usage

Map and Set

- **Map** and **Set** are provided via the **Predef** object, that includes them as **immutable**.
- If you want them to be mutable you have to import them explicitly from the **`collection.mutable`** package.
- A **Set** is like a List but without order nor repetition.
- A **Map** is a dictionary or associative array with pairs **(key, value)**.
- The default predefinition of **Map** and **Set** in Predef is:

```
object Predef:
  type Map[A, +B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]
  val Map = collection.immutable.Map
  val Set = collection.immutable.Set
end Predef
```

- To do some operations faster you may use **TreeSet** and **TreeMap**.

Common operations on Set

Operation	Description
val num = Set(1, 2, 3)	Creates an immutable Set. nums.toString returns Set(1, 2, 3)
nums + 5	Adds an item to an immutable set. Returns Set(1, 2, 3, 5)
nums - 3	Removes an item from an immutable set. Returns Set(1, 2)
nums ++ List(5, 6)	Adds multiple items. Returns Set(1, 2, 3, 5, 6)
nums -- List(1, 2)	Removes multiple items. Returns Set(3)
nums & Set(1, 3, 5, 7)	Produces the intersection of two sets. Returns Set(1, 3)
nums.size	Returns the size of the set. Returns 3
nums.contains(3)	Checks for inclusion. Returns true
val words = mutable.Set.empty[String]	Creates an empty mutable set. words.toString returns Set()
words += "the"	Adds an item. words.toString returns Set(the)
words -= "the"	Removes an item if it exists. words.toString returns Set()
words ++= List("do", "re", "mi")	Adds multiple items. words.toString returns Set(do, re, mi)
words --= List("do", "re")	Removes multiple items. words.toString returns Set(mi)
words.clear	Removes all items. words.toString returns Set()

Common operations on Map

Operation	Description
val nums = Map("I"->1, "ii"->2)	Creates an immutable Map. num.toString returns Map(i->1, ii->2)
nums + ("vi"->6)	Adds an entry to an immutable Map. Returns Map(i->1, ii->2, vi->6)
nums - "ii"	Removes an entry from an immutable Map. Returns Map(i->1)
nums ++ List("iii"->3, "v"->5)	Adds multiple entries. Returns Map(i->1, ii->2, iii->3, v->5)
nums - List("I", "ii")	Removes multiple entries from an immutable Map. Returns Map()
nums.size	Returns the number of entries in the Map. Returns 2
nums.contains("ii)	Checks for inclusion. Returns true
nums("ii")	Retrieves the value for a specified key. Returns 2
nums.keys	Returns the keys. Returns an Iterable over the strings "i" and "ii"
nums.keySet	Returns the keys as a Set. Returns Set(I, ii)
nums.values	Returns the values. Returns an Iterable over the integers 1 and 2
nums.isEmpty	Indicates whether the Map is empty. Returns false

- **Mutable Maps** behave similarly to **mutable Set**.

Generics

- **Scala** uses the symbols **[]** for generics instead of the <> of Java.
- **Scala** does not allow raw collections (without generics).
- Usually, if you create **MyType[T]** and B inherits from A, MyType[B] does not inherit from MyType[A].
  - If you want it the declare MyType as **MyType[+T]**.
  - This has some other implications, and it is highly discouraged util you master Scala.
- You use the syntax **MyType[T <: Two]** to indicate that MyType must be instantiated with a type that inherits from the class Two.

Generics

```
class One
class Two extends One
class Three extends Two

class Generic_A[T](val x: T):
   def myFunction(y: T) = y
class Generic_B[+T](val x: T):
   def myFunctionWrong(x: T) = 0 //Error
   def myFunction[U >: T](y: U) = y
class Generic_C[T <: Two](val x: T)
```

Check at home how Generic_B behaves. Pages 372 and 377 of the book «Programming in Scala» 5th edition will answer your questions.

```
val z1: Generic_C[Two] = new Generic_C[Two](new Two)
val z2: Generic_C[Two] = new Generic_C(new Three)
val z3: Generic_C[Two] = new Generic_C(new One) //Error
val z4: Generic_C[Three] = new Generic_C(new One) //Error
val z5: Generic_C[One] = new Generic_C[Two](new Two) //Error
```

Inheritance

- **Inheritance** is, together with **polymorphism** (and **dynamic dispatch**), one of the fundamental features of OOP.
- It allows you to build new classes incrementally from others already defined.
- The new classes inherit the attributes and methods of the original class, being able to redefine them and/or extend the original class with new attributes and methods.
- When a class B inherits from a class B, A is called parent or superclass whereas B is the child or subclass.
- In Scala, inheritance is simple: a class cannot inherit from more than one class at a time.

Abstract classes

- An **abstract class** is a class that contains the declaration of a method but not its implementation.
- Any class with at least one unimplemented method should be declared abstract.
- You cannot create objects of an abstract class.

```
abstract class Element {
    def contents: Array[String]
    def heigth: Int = contents.length
    def width: Int = if (heigth==0) 0 else contents(0).length
}
```

Definition of an abstract method, it is not necessary to put *abstract* in front: simply left it unimplemented.

NON-abstract methods
heigth returns the number of lines
width returns the length of the first line

Overriding

- Overwriting is achieved with the **override** keyword.

```
class Rational(n:Int, d:Int) {
   override def toString: String = s"$n/$d"
}
object Rational {
   def apply(n:Int, d:Int): Rational = {
      val p = new Rational(n, d)
      p
   }
}
```

- **override** can be used also with fields (see next slide).
- Keyword **final** can be used to avoid overwriting.
- You cannot inherit from a **final** class.

Invoking superclass constructors

- A call to the super constructor is placed just along the **extends** clause (like in C#).

```
import java.awt.Color

class Figure(val c: Color) {
   def print() = {
      println(c);
   }
}

class Triangle(override val c: Color, length: Int) extends Figure(c) {
   override def print() = {
      super.print()
      println(length)
   }
}
```

Operators

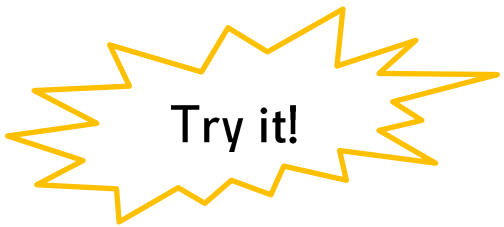
- **Scala** allows to overload hardcoded compiler operators like **+, -, etc.**
- E.g.:

```
class Rational(numer: Int, denom: Int):
  require(d != 0)

  private val g = gcd(numer.abs, denom.abs)
  val n = numer / g
  val d = denom / g
  def this(n: Int) = this(n, 1)

  def +(that: Rational) =
    Rational(this.n * that.d + that.n * this.d, this.d * that.d)

  def +(that: Int): Rational =
    this + Rational(that, 1)
...
```

Operators

- **Scala** allows to overload hardcoded compiler operators like **+, -, etc.**
- E.g.:

```
...
  def -(that: Rational) =
    Rational(this.n * that.d - that.n * this.d, this.d * that.d)

  def -(that: Int): Rational =
    this - Rational(that, 1)

  override def toString = s"$n/$d"

  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
```



Traits

- A **Trait** is like an **interface** in Java.
- It encapsulates methods and field definitions that can be reused by mixing them with other classes and traits. When a trait is mixed it is called **mixin** and the mixing process is called **mixin composition**.
- The traits:
  - Can be partially implemented, i.e. it is possible to define default. implementations for some methods.
  - Can have specific fields that are added to the implementation classes.
  - Cannot have parameters in a constructors.
- To inherit from a trait, you may use two keywords:
  - **extends** when you inherit only from traits.
  - **with** when you also inherits from a regular class.

Traits: extends

```
trait Philosophical:
  def philosophize() = println("I consume memory, therefore I am!")

class Frog extends Philosophical:
  override def toString = "green"

object Test extends App:
  val frog = new Frog
  println(frog)
  println(frog.philosophize())
```

`Philosophical` is subclass of **`AnyRef`**.
`Frog:`
- Is subclass of **`AnyRef`**
- Mixes in **`Philosophical`**

Default implementation.

frog is also **Philosophical**, a trait.

Traits: with

```
class Animal

trait Philosophical:
    def philosophize() = println("I consume memory, therefore I am!")

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs:
    override def toString = "green"
    override def philosophize() =
        println("It isn't easy being " + toString + "! ")


object Test extends App:
    val frog = new Frog
    println(frog)
    println(frog.philosophize())
```

extends is used to indicate the superclass...

... and **with** to mix in the trait. Multiple traits can be mixed

Frog can overwrite the legacy implementation of the trait

```
"C:\Program Files\Java\jdk-21\bin\ja
Picked up JAVA_TOOL_OPTIONS: -Dfile.
green
It isn't easy being green !
()

Process finished with exit code 0
```

Traits and classes

- Differences between a **class** and a **trait** in Scala:
  - **Traits** cannot have a constructor with parameters
  - In the **classes** the calls to **super** are solved **statically**, in the **traits** **dynamically**.
  - For example, when a call to **super.toString** call is executed:
    - In a **class** it is known exactly which implementation of the method will be invoked.
    - In a **trait** the implementation of the method to be invoked is not determined when the **trait** is defined, it will be set again each time the **trait** is mixed into a particular class.
    - The method that will be executed will depend on the order in which the **traits** are inherited: the reverse in which they are specified.

Traits

```
trait Logger:
   def log(msg: String) = {}

trait ConsoleLogger extends Logger:
   override def log(msg: String) = { println(msg) }

trait TimestampLogger extends Logger:
   override def log(msg: String) =
      super.log("" + new java.util.Date() + " " + msg)

trait ShortLogger extends Logger:
   val maxlength = 15
   override def log(msg: String) =
      super.log(  if (msg.length <= maxlength) msg
                  else msg.substring(0, maxlength - 3) + "...")
```

When this super call is made on a trait, the method that will be executed will depend on the order in which the traits will be mixed in.

Traits

```
trait Logger:
   def log(msg: String) = {}

trait ConsoleLogger extends Logger:
   override def log(msg: String) = { println(msg) }

trait TimestampLogger extends Logger:
   override def log(msg: String) =
      super.log("" + new java.util.Date() + " " + msg)

trait ShortLogger extends Logger:
   val maxlength = 15
   override def log(msg: String) =
      super.log(  if (msg.length <= maxlength) msg
                  else msg.substring(0, maxlength - 3) + "...")
```

5

4

3

When a call to withdraw is made, the log methods are invoked in reverse order of the one specified:
* ShortLogger
* TimestampLogger
* ConsoleLogger

```
class SavingsAccount extends Logger:
   private var balance: Double = 0
   def withdraw(amount: Double) =
      if (amount > balance) log ("Insufficient funds")
      else balance -= amount

object Test extends App:
   val acc1 = new SavingsAccount with ConsoleLogger
                        with TimestampLogger
                        with ShortLogger
   acc1.withdraw(1000)
```

2

1

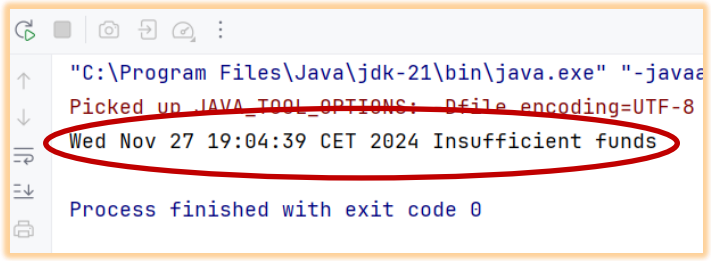
```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaa
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Wed Nov 27 18:39:42 CET 2024 Insufficient...

Process finished with exit code 0
```

Result

Traits

```
trait Logger: …
trait ConsoleLogger extends Logger: …
trait TimestampLogger extends Logger: …

trait ShortLogger extends Logger:
   val maxlength = 15
   override def log(msg: String) =
      super.log(  if (msg.length <= maxlength) msg
                  else msg.substring(0, maxlength - 3) + "...")

class SavingsAccount extends Logger: …

object Test extends App:
   val acc1 = new SavingsAccount with ConsoleLogger
                              with TimestampLogger
                              with ShortLogger:
                                 override val maxlength = 20
   acc1.withdraw(1000)
```

An attribute can be overriden when the trait is mixed in



Ordered trait

- Defining comparison operators (<, >, <=, and >=) in a class is a very common problem.
- Scala provides the **Ordered** trait to help resolve this.
- It doesn't define equals, so even if a class is mixed in with **Ordered**, it should define equals if needed.

```
trait Ordered[T] {
    def compare(that: T): Int

    def <(that: T): Boolean = (this compare that)  < 0
    def >(that: T): Boolean = (this compare that)  > 0
    def <=(that: T): Boolean = (this compare that) <= 0
    def >=(that: T): Boolean = (this compare that) >= 0
}
```

Ordered trait

- To use **Ordered**:
  - Specifying a Type Parameter
  - Implement the **compare** method that compares the receiver, **this**, with the object that is passed to it as an argument. You will need to return:
    - **0** if the objects are the same
    - **Negative integer** if the receiver is less than the argument.
    - **Positive integer** if the receiver is greater than the argument

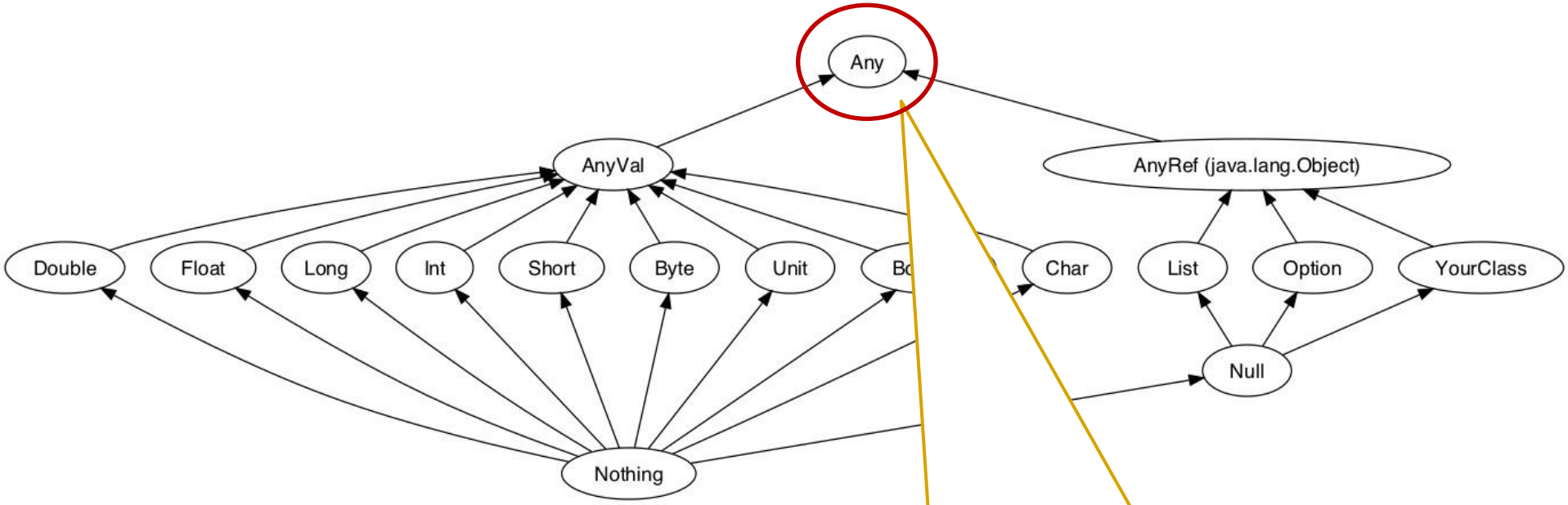
```
class Rational(val n: Int, val d: Int) extends Ordered[Rational] :
   def compare(that: Rational) =
      (this.n * that.d) - (that.n * this.d)

object Test extends App:
   val r1 = new Rational(8, 9) // Rational = 8/9
   val r2 = new Rational(7, 5) // Rational = 7/5
   r1 > r2 // false
   r1 < r2 // true
```

compare method

Generic type

Scala's hierarchy

- In Scala, all values have a type, including numeric values and functions.



All types are subtypes of **Any**.
Any includes a set of universal methods such as **equals**, **hashCode**, or **toString**.

Bottom classes

- Class **Null** is the type of the **null** reference. It is below **AnyRef**.
- **Null** is not compatible with **val** types:

```
val i: Int = null  Error!
```

- **Null** or **null** are hardly used in Scala.
- Instead of Null you have to use **None** (absence of data).
- Type **Nothing** is at the bottom of hierarchy.
- There is no value for the type **Nothing**.
- **Nothing** signals abnormal or no termination. E.g.:

```
def error(msg: String): Nothing  = {
   throw new RuntimeException(msg)
}
```

```
def divide(a:Int, b:Int): Int =
   if (b != 0) a/b
   else sys.error("Can't divide by zero.")
```

Nothing inherits from everything

Classes Option and Some

- Class **Option** is used when a **None** value must be managed.
- For example, if nums is a Map, num("ii") returns the value associated to the key "ii" but, what if such a key does not exist? Exception!!
- In such a case you should use num.get("ii") that returns **Option[Int]** whose "shape" may be **Some[Int]** or **None**.
- Usually, you manage these values through pattern matching, a concept you will learn later.

```
object Main extends App:
   val m: Map[String, Int] = Map("ii"->2, "i"->1)
   val data = m.get("ii")
   data match
      case None => print("Data is missing")
      case Some(value) => print(s"Data associated is $value")
```

Asserts

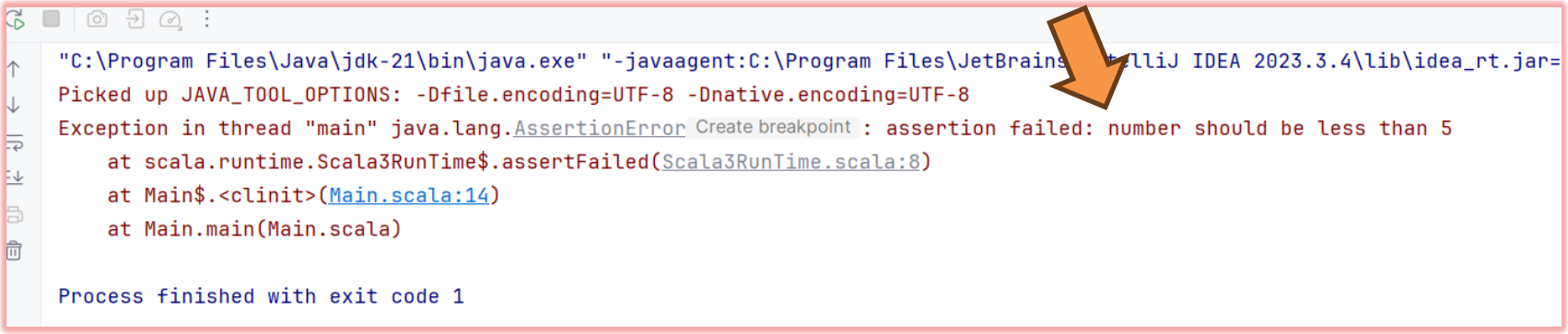
- **Assertions** are used to verify assumptions about the program's behavior during execution.
- Ensure correctness of code by validating expected conditions.
- Typically used in debugging and testing to catch logical errors early.
- **Assertions** are part of **scala.Predef**.
- Can be disabled in production by running Scala with **-Xdisable-assertions**.
- Assertions are powerful debugging tools.
- Ensure they are disabled in production for performance and safety.

```
object Main extends App:
  val x = 5
  assert(x > 0, "x must be positive")
```

How assertions work

- How Assertions Are Evaluated:
  - If the condition evaluates to **true**:
    - Program continues execution normally.
  - If the condition evaluates to **false**:
    - Throws an **AssertionError** with the optional message (if provided).

```
object Main extends App:
  val number = 10
  assert(number < 20, "number should be less than 20") // Passes
  assert(number < 5, "number should be less than 5") // Fails
```



Usage scenarios

- **Testing preconditions**: Ensure input meets specific criteria.
- **Debugging**: Verify intermediate states during development.
- **Safety checks**: Validate invariants in critical sections of code.

  - In functions:

```
object Main extends App:
  def divide(a: Int, b: Int): Int =
    assert(b != 0, "Denominator must be non-zero")
    a / b

  divide(10, 2) // Works fine
  divide(10, 0) // Throws AssertionError
```

  - Conditional assertions:

```
object Main extends App:
  val age = 25
  if (age < 18) then
    assert(false, "Age must be 18 or older")
```

When to use assertions

- Use assertions for internal logic validation, not for user input validation.
- Avoid side effects in assertion expressions (e.g., assert(foo()) where foo modifies state).
- Provide clear and descriptive error messages.

- In production-critical paths: Use exception handling or logging instead.
- For runtime data validation: Use error handling with try-catch or custom exceptions.

require

- **require** is not a specific Scala keyword but a concept often implemented in Scala code, particularly when defining class **constructors or methods to enforce certain conditions**.
- It ensures that certain conditions are satisfied when initializing objects or calling methods, typically using assertions or other mechanisms.
- Enforces mandatory requirements at runtime or compile-time.

```
class Person(val name: String, val age: Int) {
   require(name.nonEmpty, "Name cannot be empty")
   require(age > 0, "Age must be positive")
}

object Main extends App:
   val person = new Person("Alice", 25) // Works fine
   val invalidPerson = new Person("", -5) // Throws IllegalArgumentException
```

Differences between require and assert

Aspect	require	assert
Use case	Enforcing preconditions for public APIs	Validating internal logic during debugging
Exception	Throws IllegalArgumentException	Throws AssertionError
Scope	Inputs or arguments for functions/constructors	General logic checks
Production	Should remain in production code	Can/should be disabled in production

- Use **require** to validate external inputs and throw meaningful exceptions for the caller.
- Use **assert** to verify internal logic during development, especially when debugging.

- Prefer **require** for public-facing APIs to ensure argument validity.
- Use meaningful error messages to guide developers or users.
- Do not replace comprehensive input validation with require; combine with full checks when necessary.