Scala

Data types and Variables
Basic Data Types

- Int
- Float
- Double
- Boolean
- Char
- String

Variables

- `var name: type = value`
- `var` is the keyword of Scala that is used to define a variable
- Variables must be initialized during their definition
- The type of the variable is implicitly inferred from the initialization value if the type is not specified
- Variables are public if the keyword `private` is not specified
Variables: Examples

- var sentence = “Hello world”
  - Implicit inference of the date type of sentence
- var sentence: String = “Hello world”
  - Explicit definition of the date type of sentence
- sentence is a public variable of type String and its initial value is “Hello world”

Variables: Examples

- var weight = 75
- var weight: Int = 75
- weight is a public variable of type Int and its initial value is 75
Variables: Examples

• private var weight = 75
• private var weight: Int = 75
• weight is a private variable of type Int and its initial value is 75

Variables: Examples

• var price = 10.5
• var price: Double = 10.5
• price is a public variable of type Double and its initial value is 10.5
Variables: Examples

- var condition = true
- var condition: Boolean = true
- condition is a public variable of type Boolean and its initial value is true

Variables: Examples

- var firstCharacter = ‘a’
- var firstCharacter: Char = ‘a’
- firstCharacter is a public variable of type Char and its initial value is ‘a’
Immutable Variables (or Values)

- `[private] val name[: type] = value`
- `val` is the keyword of Scala that is used to define immutable variables (i.e., constants) also called `values`
- Immutable variables must be initialized during their definition
- The type of immutable variables is implicitly inferred from the initialization value if the type is not specified
- Immutable variables are public if the keyword `private` is not specified

Immutable Variables: Examples

- `val sentence = “Hello world”`
  - Implicit inference of the data type of `sentence`
- `val sentence: String = “Hello world”`
  - Explicit definition of the data type of `sentence`
- `sentence` is a public immutable variable of type `String` and its initial value is “Hello world”
Standard operators

• Mathematical operators
  • +, -, *, /, %
• Boolean operators
  • |, &, !

Standard operators: Examples

• var num = 10
• var den = 3
• var ris = 10/3
• var ris = 10%3
• var text = “Hello World”
• text = text + “ Paolo”
Strings

- Scala’s String is built on Java’s String
- Scala’s String has also some additional features
  - Multiline literals
  - String interpolation

Strings: Examples

- Use of double quotes and special characters escaped with backslash

```scala
val hello = "Hello There" /* Hello There */
val signature = "With regards, \nYour friend"
/*
With Regards,
Your friend
*/
```
Strings: Examples

- String concatenation (or string addition)
  val hello = "Hello, " + "World" /* Hello, World */

- String comparison
  val matched = (hello == "Hello, World") /* true */
  - Unlike Java, in Scala the equals operator (==) compares the contents of the Strings

Multiline Strings

- A multiline String can be created using triple-quotes
  - Multiline Strings are literal, and do not recognize the use of backslashes, i.e., they do not recognize special characters

  val greeting = """"She suggested reformatting the file by replacing tabs (\t) with newlines (\n);
  "Why do that?", he asked. ""
""""
String interpolation

- Building a string based on other variables/values can be done with string addition
- E.g.,
  ```scala
  val approx = 355/113f
  println("Pi, using 355/113, is about "+ approx + ".")
  /* Pi, using 355/113, is about 3.141593 */
  ```

String interpolation

- String interpolation is another way to combine variables/values inside a string
- The Scala notation for is an “s” prefix added before the first double quote of the string
- The dollar sign operator $ (with optional braces) can be used to insert references to variables/values
String interpolation: Example

val approx = 355/113f
/* approx: Float = 3.141593 */
println(s"Pi, using 355/113, is about ${approx}.")
/* Pi, using 355/113, is about 3.141593. */

Strings: Other particular operations

- Repeat the same sequence of characters multiple times
  
  val repeatHi="Hi "*5
  println(repeatHi)
  val str1="Paolo 
  val num=10
  val rep=str1*num
  println(rep)
Regular expressions

• As many other languages, Scala supports regular expressions

• A regular expression is a string of characters and punctuation that represents a search pattern
  • The format is the same used by the Java class java.util.regex.Pattern

Use of regular expressions:
Example

• matches
  • It is used to check if the content of String matches the provided regular expression

  var sentence= "Test matching operation"
  var res: Boolean = sentence.matches("Test .*")
  println(res)
Use of regular expressions:
Example

• replaceAll
  • Replaces all matches of the regular expression with the specified replacement text

```java
var sentence= "milk, tea, muck"
var res: String = sentence.replaceAll("m[^ ]+k", "coffee")
println(res)
```

Use of regular expressions:
Example

• replaceFirst
  • Replaces the first match of the regular expression with the specified replacement text

```java
var sentence= "milk, tea, muck"
var res: String = sentence.replaceFirst("m[^ ]+k", "coffee")
println(res)
```
The Scala data type hierarchy

The following operations are available on all types

- `asInstanceOf[type]`
  - Converts the value to a value of the desired type
  - Causes an error if the value is not compatible with the new type.
  - E.g., `5.asInstanceOf[Long]`
Common operations

- **getClass**
  - Returns the type (i.e., the class) of a value
  - E.g., (7.0 / 5).getClass

Common operations

- **isInstanceOf**
  - Returns true if the value has the given type
  - E.g., (5.0).isInstanceOf[Float]
Common operations

• **to<type>**
  • Conversion functions to convert a value to a compatible
  • E.g., 20.toByte
  • 47.toFloat

Common operations

• **toString**
  • Renders the value to a String
  • E.g., (3.0 / 4.0).toString
Cast: Data type conversion

- asInstanceOf[<type>] and the to* methods can be used to convert data from one data type to another
  - Obviously only if the content of the “input” variable/value is compatible with the “output” data type
  - The to* methods are preferred

Cast: Examples

```scala
var longAge: Long = 40
var intAge: Int = longAge.toInt
print(longAge+"---"+intAge)
```

```scala
var stringAge: String = "40"
var intAge=stringAge.toInt
print(stringAge+"---"+intAge)
```
Arrays, Lists, Maps, Tuples

Basic operations

Collections and complex data types in Scala

- Array
- List
- Map
- Tuple
Arrays

• Scala provides the Array data type
• Scala supports
  • Homogeneous arrays
    • All the elements of the array are associated with the same data type
  • Heterogeneous arrays
    • The elements of the array belong to different data types

Homogeneous arrays: Example

• Definition of an array of integers containing the values 1, 2, 3
  val numbers = Array(1, 2, 3)
• numbers is of type Array[Int]
Homogeneous arrays: Example

- Print on the console the value of the first element

```scala
val numbers = Arrays(1, 2, 3)
println(numbers(0))
```

Heterogeneous arrays: Example

- Definition of an array containing integers and strings

```scala
val mix = Array(1, "Hello", "World", 10)
mix is of type Array[Any]
```
Collections: List

- Scala provides the List data type
- Scala supports
  - Homogeneous lists
    - All the elements of the list are associated with the same data type
  - Heterogeneous lists
    - The elements of the list belong to different data types

Homogeneous lists: Example

- Definition of a list of integers containing the values 1, 2, 3
  
  val numbers = List(1, 2, 3)
  
  or
  
  val numbers = 1 :: 2 :: 3 :: Nil

- numbers is of type List[Int]
Homogeneous lists: Example

- Print on the console the value of the first element
  val numbers = List(1, 2, 3)
  println(numbers(0))

Heterogeneous lists: Example

- Definition of a list containing integers and strings
  val mix = List(1, “Hello”, “World”, 10)
  or
  val mix = 1 :: “Hello” :: “World” :: 10 :: Nil
- mix is of type List[Any]
Concatenation of lists

• Concatenate lists
  • val res = list1 :: list2
  • Or
    • val res = List.concat(list1, list2)
      • List.concat() receives as arguments two or more lists

Collections: Map

• Scala provides the Map data type
• Map is used to maintain the mapping between keys and values
  • Each key is associated with only one value
Maps

- Add a new pair key -> value
  - `mapvariable += newkey -> newvalue`
- Retrieve the value associated with a key
  - `mapvariable(key)`
  - or
  - `mapvariable.get(key).get`
- Default value for missing keys
  - `mapvariable.get(key).getOrElse(default value)`

Maps: Example

- Definition of a map variable that maps integers (key) to strings (values)

```scala
/* Define the Map and insert the first key -> value pair */
val mapper = Map(1 -> "Hello")
/* Add a new pair */
mapper += 2 -> "World"
println(mapper(1))
```
Maps: Example

- Definition of a map variable that maps integers (key) to strings (values)

```scala
/* Define an empty Map of type Map[Int, String] */
val mapper: Map[Int, String] = Map()
/* Add two new pairs */
mapper += 1 -> "Hello"
mapper += 2 -> "World"
println(mapper(1))
```

Maps: Example #2

- Definition of a map variable that maps string (key) to strings (values)

```scala
val stateCapitals = Map( "Alabama" -> "Montgomery", "Alaska" -> "Juneau")
println("Alabama: " + stateCapitals("Alabama")
 .getOrElse("Unknown"))
println("Italy: " + stateCapitals.get("Italy") .getOrElse("Unknown"))
```
Tuples

- Scala has a specific data type that is used to represent tuples
  - Tuples are groups of $N$ items
  - Elements are unrelated to each other. The data types can be different
  - They are useful to return a set of values from a method without defining a new class or structure
- Pay attention that the items of tuples are immutable

Tuples

- Definition of tuples
  - var name=(comma separated list of items)
- Retrieval of the $N$-th item
  - name._$N$
**Tuples: Examples**

- **Scala:**
  ```scala
  val tuple: Tuple2[Int, String] = (1, "apple")
  val quadruple = (2, "orange", 0.5d, false)
  ```

- **Java:**
  ```java
  Pair<Integer, String> tuple = new Pair<Integer, String>(1, "apple")
  Quadruples: No equivalent in Java
  ```
Generics in Scala

- Analogously to Java, also Scala supports generics

Generics: Examples

- Scala:
  - List[String]
  - List[Int]
  - Map[Int, String]
  - ...

- Java:
  - List<String>
  - List<Integer>
  - Map<Int, String>
Expressions

• Expression
  • A unit of code that returns a value
  • One or more lines of code can be considered an expression if they are collected together using curly braces
    • This is known as expression block
Expressions: Example

• “hello”
  • Is a very simple expression
• “hel”+”lo”
  • Is another very simple expression
• As we already did, expressions can be used to assign values to variables and immutable variables (values)
  • val message=“hello”

Expression blocks

• An expression block is a sequence of one or more lines of code
• An expression has its own scope, and may contain values and variables local to the expression block
• The last expression in the block is the return value for the entire block
Expression blocks: Example

- Example with expressions containing only one line of code
  
  \[
  \text{val } x = 5 \times 20; \\
  \text{val } amount = x + 10
  \]

- Example with an expression block with multiple lines of code
  
  \[
  \text{val } amount = \{ \text{val } x = 5 \times 20; x + 10 \}
  \]

- The value of amount is the same in both cases
Expression blocks: Example

- Example with expressions containing only one line of code
  
  ```scala
  val x = 5 * 20;
  val amount = x + 10
  ```

- Example with an expression block with multiple lines of code
  
  ```scala
  val amount = {val x = 5 * 20; x + 10}
  ```

- The value of amount is the same in both cases

---

Expression blocks: Example

- Example with an expression block with multiple lines of code
  
  ```scala
  val amount = {val x = 5 * 20; x + 10 }' 
  ```

- Is equivalent to
  
  ```scala
  val amount = {val x = 5 * 20
               x + 10 }
  ```
Expression blocks: Example

- Example with an expression block with multiple lines of code
  
  \[
  \text{val amount} = \{ \text{val } x = 5 \times 20; x + 10 \} \\
  \]

- Is equivalent to
  
  \[
  \text{val amount} = \{ \text{val } x = 5 \times 20 \\
  \quad \quad \quad \quad \quad x + 10 \} \\
  \]

  This is the returned value (expression)

Expression blocks: Example

- Example of a three-deep nested expression block
  
  \[
  \text{val res} = \{ \text{val } a = 1; \{ \text{val } b = a \times 2; \{ \text{val } c = b + 4; c \} \} \} \\
  \]
Expression blocks: Example

- Example of a three-deep nested expression block

val res = { val a = 1; { val b = a * 2; { val c = b + 4; c } } }

This is the returned value (expression)

Conditional expressions
if-then-else

• if-then-else in Scala is analogous to those of the Java, C, C++ languages
• However, it is also an expression
• The if expression evaluates a Boolean expression
  • If the result of the Boolean expression is equal to true a block of code is executed
  • Otherwise the block of code associated with the else part of the statement is executed

Scala:

```scala
if (test) {
  /* code */
} else {
  ...
}
```

Java:

```java
if (test) {
  ...
} else {
  ...
}
```
if-then-else

- Scala:
  ```scala
  if (test) {
    ...
  } else if (test2) {
    ...
  } else {
    ...
  }
  ```

- Java:
  ```java
  if (test) {
    ...
  } else if (test2) {
    ...
  } else {
    ...
  }
  ```

if-then-else as an expression

- if-then-else in Scala is also an expression
- If the Boolean condition of the if-then-else is true then last expression of the if expression block is the returned value
- Otherwise, the last expression of the else block is the returned value
if-then-else as an expression:
Example

val x=10
val y=20
val max = {if  (x>y) x else y}
println(max)

if-then-else as an expression:
Example#2

val x=10
val y=20
val max = { if  (x>y) {
println(x +"">" + y)
x }
else {
println(x +"<="+ y)
y }
}
println(max)
Match expressions

- Match expressions are like the “switch” statements in Java and C++
  - A single input item is evaluated and the first pattern that is “matched” is executed and its value returned
• Like C’s and Java’s “switch” statements, Scala’s match expressions support a default or wildcard “catch-all” pattern
• Unlike them, only zero or one patterns can match

The traditional “switch” statement is limited to matching by value
• Scala’s match expressions are flexible and also enable matching such diverse items as types, regular expressions, numeric ranges, and data structure contents
• Moreover, match expressions are expressions
  • Hence, they can return values
Match expressions: Syntax

<expression> match {
    case <pattern match> => <expression>
    [case...]
}

Match expressions: Example

val test: Char = 'a'
test match {
    case 'a' => { println("Code associated with a") }
    case 'b' => { println("Code associated with b") }
}

Match expressions: Example #2

```scala
val test: Char = 'A'

test match {
  case 'a' | 'A' => { println("Code associated with a or A") }
  case 'b' | 'B' | 'c' | 'C' => { println("Code associated with b, B, c, or C") }
}
```

Match expressions: Returned values

- Match expressions are expressions
- Hence, they return values if the last expression of the executed code is an expression
Match expressions: Example #3

val max = x > y match {
    case true => x
    case false => y
}

The match expression is evaluated on this Boolean test
Match expressions: Example #3

val max = x > y match {
  case true => x
  case false => y
}

If x > y is true then the value of x is returned
If x > y is false then the value of y is returned
Match expressions: Example #4

```scala
val day = "MON"
val kind = day match {
  case "MON" | "TUE" | "WED" | "THU" | "FRI" => "weekday"
  case "SAT" | "SUN" => "weekend"
}
println(kind)
```

Match expressions: “Default” value

- There are two kinds of wildcard patterns you can use in a match expression
  - Value binding and
  - Wildcard (aka “underscore”) operators
Match expressions: Value binding

- With **value binding** the input to a match expression is bound to a local value (immutable variable)
  - The local value can then be used in the body of the case block.
  - Because the pattern contains the name of the value to be bound there is no actual pattern to match against
    - Thus value binding is a wildcard pattern because it will match any input value

---

Match expressions: Value binding example

- The following example sets status (integer) to
  - 200 if the message is “Ok”
  - -1 otherwise
Match expressions: Value binding example

val message = "Ok"
val status = message match {
  case "Ok" => { println("matched Ok\n"
    200
  case other => { println(other+" matches nothing")
    -1
  }
}
println(status)
Match expressions: Wildcard operator

- The **wildcard** cannot be accessed on the right side of the arrow, unlike with value binding.
  - If you need to access the value of the wildcard in the case block, consider using a value binding, or just accessing the input to the match expression (if available).

Match expressions: Value binding example

- The following example sets status (integer) to
  - 200 if the message is “Ok”
  - -1 otherwise
Match expressions: Value binding example

val message = "Ok"
val status = message match {
  case "Ok" => { println("matched Ok")
    200
  }
  case _ => { println("matches nothing")
    -1
  }
}
println(status)
Match expression: Pattern guards

- A **pattern guard** adds an if expression to a value-binding pattern
  - It allows mixing conditional logic into match expressions
  - When a pattern guard is used the pattern will only be matched when the if expression returns true

---

Match expression: Pattern guard example

```scala
val response: String = null
response match {
  case s if (s != null) => println("Received "+s)
  case s => println("Error! Received a null response")
}
```
Match expression: Pattern guard example

val response: String = null
response match {
  case s if (s != null) => println("Received "+s)
  case s => println("Error! Received a null response")
}

Value binding. The value of response is assigned to s

Match expression: Pattern guard example

val response: String = null
response match {
  case s if (s != null) => println("Received "+s)
  case s => println("Error! Received a null response")
}

Pattern guard. If the condition is true the code of this case is executed
Match expression: Pattern guard example

``` scala
val response: String = null
response match {
  case s if (s != null) => println("Received "+s)
  case s => println("Error! Received a null response")
}
```

Pattern guard. If the condition is false the next cases are considered.

Match expression: Pattern guard example

The following code is equivalent to the previous one

``` scala
val response: String = null
response match {
  case s if (s != null) => println("Received "+s)
  case _ => println("Error! Received a null response")
}
```
Match expression: Pattern guard example

The following code is equivalent to the previous one:

```scala
val response: String = null
response match {
  case s if (s != null) => println("Received "+s)
  case _ => println("Error! Received a null response")
}
```

Here the wildcard is sufficient.

---

Match expression: Matching types

- In Scala you can specify a matching also on the type of the input
- Java and C++ do not support this type of test
Match expression: Matching type example

val list = List("a", 1, 'c', 34.5)

list(0) match {
  case v: String => println("This is a string")
  case v: Int => println("This is an integer")
  case v: Char => println("This is a char")
  case v => println("This is another type")
}
for loop

- The for loop in Scala is different with respect to those of C, C++
- The for loop in Scala always iterates over an input collection
  - For each element \( e \) of the input collection, the block of code associated with the for loop is executed
- The for loop in Scala is a “functional for loop”

for loop: Example 1

- Scala:
  
  ```scala
  for (i <- 0 to 3) {
    /* code */
  }
  ```
  
  - The expression “0 to 3” defines a collection of integers containing the values 0, 1, 2, 3

- Java:
  
  ```java
  for (int i = 0; i < 4; i++) {
    /* code */
  }
  ```
for loop: Example 2

- Scala:
  ```scala```
  for (s <- args) {
    println(s)
  }
  ```

- `args` is the collection of arguments of the application. Each element is a string

- Java:
  ```java```
  for (String s : args) {
    System.out.println(s);
  }
  ```

for loop: yield option

- A for loop can return a collection
- At each iteration, the last expression is the returned value
- Syntax
  ```
  for (<identifier> <- <iterator>) yield {<expression>}
  ```
  ```
  for (x <- 1 to 7) yield { s"Day $x:" }
  ```
for loop: yield option example

- Returns ("Day 1", "Day 2", .., "Day 7",)
  
  \[
  \text{val res=for (x <- 1 to 7) yield \{ "Day " + x \}}
  \]

- Returns ("Even", "Odd", "Even", .., "Even",)
  
  \[
  \text{val res=for (x <- 1 to 7) yield \{ if (x%2==0) \{"Odd"\} else \{"Even"\} \}}
  \]

while loop

- The while loop in Scala is analogous to those of the Java, C, C++ languages.
- The while loop executes the code of block associated with it as long as the evaluated condition is true
while loop

- Scala:
  ```scala
  while (expression==true) {
      ...
  }
  ```

- Java:
  ```java
  while (expression==true) {
      ...
  }
  ```

do-while loop

- The do-while loop in Scala is analogous to those of the Java, C, C++ languages.

- The do-while loop executes the code of block associated with it as long as the evaluated condition is true
  - The code is always executed at least one time
do-while loop

- Scala:
  ```scala
  do {
    ...
  } while (expression==true)
  ```
- Java:
  ```java
  do {
    ...
  } while (expression==true)
  ```

Console: Basic operations

- Scala:
  ```scala
  Console.println("Hello")
  
or simply
  println("Hello")
  ```
- Java:
  ```java
  System.out.println("Hello");
  ```

```scala
  var i:Int = 10
  println("Value of i:" + i)
  ```
- Java:
  ```java
  Integer i=10;
  System.out.println("Value of i:" + i)
  ```
Console: Basic operations

- Scala:
  ```scala
  val line = Console.readLine()
  or simply
  val line = readLine()
  ```

- Java:
  ```java
  BufferedReader r = new BufferedReader(new InputStreamReader(System.in))
  String line = r.readLine();
  ```

Console: Read operations

- The read* methods of the Console Object are used to read data from the console in Scala
  - Console.readLine
  - Console.readChar
  - Console.readDouble
  - Console.readInt
  - Console.readLine
  - ...

Read operations: Example

```java
Console.println("Insert your name")
var name=Console.readLine()
Console.println("Hi "+name)
Console.println("How old are you?")
var age=Console.readInt()
Console.println("Do you like Scala?")
var like=Console.readBoolean()
Console.println("Name: "+name + " Age: "+ age + "Like Scala: "+like )
```

Console: readf* methods

- The methods readf, readf1, readf2, and readf3 can be used to read multiple values at the same time
- `Console.readf(String)`
  - Returns a list of values of type Any (List[Any])
- `Console.readf1(String)`
  - Returns one value of type Any
- `Console.readf2(String)`
  - Returns two values of type Any (Any,Any)
Console: readf* methods

- The parameter of the readf* methods is a string that specify the type of the expected input values
- E.g.
  - "{0} {1,number}" means that the expected values are a string and then a number
  - "{0,number} {1,number} {2}" means that the expected values are two numbers and a string
- An parse exception is generated if the input values are not consistent with the expected data types

Console: readf* methods

- The main problem of readf, readf1, readf2, and readf3 is the type of the returned values
  - All the returned values are of type Any
  - Values must be casted to the correct data type
Console: readf* methods:

Example

```scala
Console.println("Insert your name and age")
val (a,b)=Console.readf2("{0} {1,number}
var name=a.toString
var age=b.toString.toInt
Console.println("Name: "+name + ", Age:" + age)
```

```scala
Console.println("Insert your name and age")
val values: List[Any]=Console.readf("{0} {1,number}
var name=values(0).toString
var age=values(1).toString.toInt
Console.println("Name: "+name + ", Age:" + age)
```
Console: Read operations

- Since version 2 of Scala the `Console.read*` methods are deprecated. However, the “equivalent” methods `scala.io.StdIn.read*` are provided
  - `scala.io.StdIn.readBoolean`
  - `scala.io.StdIn.readChar`
  - `scala.io.StdIn.readDouble`
  - `scala.io.StdIn.readInt`
  - `scala.io.StdIn.readLine`
  - …

Read operations based on the split method

- Another approach is based on the split method of `String`
  - `split("splitting character")` returns an `Array of Strings`
- Also in this case a manual cast is needed
Read operations based on the split method: Example

```scala
Console.println("Insert your name and age")
val input=Console.readLine()
val vals: Array[String]=input.split(" ")
var name=vals(0).toString
var age=vals(1).toString.toInt
Console.println("Name: "+name + " Age:" + age)
```

Read operations based on Java Scanner

- Scala can also use the Java java.util.Scanner class to read data from the console
- However, if the input does not match what you expect and error/exception will be thrown
Read operations based on Java Scanner: Example

val scanner=new java.util.Scanner(System.in)
print("How old are you?")
val age=scanner.nextInt()
Console.println("You are "+age+"years old")