Object-oriented programming

HelloWorld

- The following code print "Hello World" on the console

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

object

- The keyword **object** in Scala is used to define a "singleton" class
  - There is one single instance of each singleton class
  - Calls to the methods of a singleton class may look like static method calls in Java

3

Classes

- The keyword **class** in Scala is used to define a class
  - There are multiple instances of each class
    - If you need only one instance of a class it means that probably it is a singleton class, i.e., use the keyword object in that case
- The keyword **new** is used to create a new instance of a class

4

Classes: Definition and instantiation

```
class FirstClass {
    /* variables and methods of the class */
}
……
var fc = new FirstClass
        or
var fc = new FirstClass()
```

5

Methods

- Methods (i.e., functions of classes) are defined by using the following syntax
- def name[(arguments)][: returned data type] = { /* code */}
- def is the keyword of Scala that is used to define methods
- Arguments and the returned data type are optional
  - You can have method without arguments or methods returning nothing

6

Methods: Examples

- Scala

```
def add(x: Int, y: Int): Int =
{
   return x + y
}
```

The method returns the sum of the two arguments

- Java

```
public int add(int x, int y)
{
   return x + y;
}
```

7

Methods: Examples

- All the following definitions are equivalent

```
def add(x: Int, y: Int): Int = {
     return x + y
}
def add(x: Int, y: Int): Int = {
     x + y
}
def add(x: Int, y: Int): Int = x + y
```

8

Methods: Examples

- All the following definitions are equivalent

```
def add(x: Int, y: Int): Int = {
    return x + y
}

def add(x: Int, y: Int): Int = {
    x + y  /* Return the result of the last operation */
}

def add(x: Int, y: Int): Int = x + y
```

9

Methods: Examples

- Scala

```
def print2Times(text:
String) {
    println(text)
    println(text)
}
```

- Java

```
public void print2Times
(String text) {
    println(text)
    println(text)
}
```

10

How to invoke a method

- Scala:

var myObject=new …

myObject.myMethod(1)

or

myObject myMethod(1)

or

myObject myMethod 1

- Java:

ClassType myObject=new…

myObject.myMethod(1);

11

How to invoke a method (2)

- Scala:

var myObject=new …

myObject.myMethod(1, 2)

or

myObject myMethod(1, 2)

- Java:

ClassType myObject=new…

myObject.myMethod(1,2);

12

How to invoke a method (3)

- Scala:

var myObject=new …

myObject.myMethod()

or

myObject myMethod()

or

myObject myMethod

- Java:

ClassType myObject=new…

myObject.myMethod();

13

Override methods

- Scala:

override def toString = { /* new code */ }

- Java:

@Override

public String toString() {/* new code */}

14

Constructors

- Scala distinguishes between
  - Primary constructor
    - All classes have a primary constructor
  - Auxiliary constructors
    - Optional

15

Primary constructor

- The signature of the primary constructor of each class is given by the list of parameters listed after the name of the class name
- The code of the primary constructor is the entire body of the class

16

Primary constructor

- Scala automatically defines a private attribute inside the class for each parameter of the primary constructor
- If the parameter is of type val, Scala automatically defines also a public read method that has the same name of the parameter
- If the parameter is of type var, Scala automatically defines also a public write method that has the same name of the parameter

17

Primary constructor

- If the keyword private is specified before the name of a parameter, the read and write methods are not generated
- Otherwise the parameter is public and the read and write methods are automatically defined

18

Primary constructor: Example

- A Java class representing a person with one attribute/variable: name

```
public class Person {
  private String name;

  public Person(String name) {
      this.name = name;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

19

Primary constructor: Example

- The same code in Scala

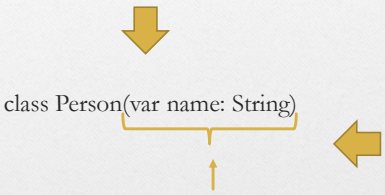
```
class Person(var name: String)
```

```
public class Person {
  private String name;

  public Person(String name) {
      this.name = name;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

20

Primary constructor: Example

- The same code in Scala



```
class Person(var name: String)
```

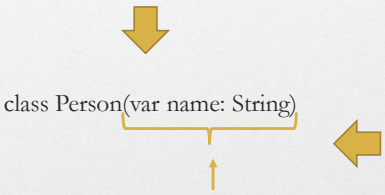
Definition of the attribute/variable ***name*** of the class Person and signature of the primary constructor

```
public class Person {
  private String name;

  public Person(String name) {
      this.name = name;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

21

Primary constructor: Example

- The same code in Scala



```
class Person(var name: String)
```

Also the get and set methods are automatically defined and have the same name of the parameter

```
public class Person {
  private String name;

  public Person(String name) {
      this.name = name;
  }

  public String getName() {
      return this.name;
  }

  public void setName(String name) {
      this.name = name;
  }
}
```

22

Primary constructor: Example 2

- Definition of the class PersonProfile
- Each instance of PersonProfile is characterized by
  - Name
  - Surname
  - Title
  - Age

Primary constructor: Example 2

- Definition of the class PersonProfile
- Each instance of PersonProfile is characterized by
  - Name
  - Surname
  - Title
  - Age

Primary constructor: Example 2

- Create a variable of type PersonProfile and print on the console the message

  Hello <title> <name> <surname>

  You are <age> years old

25

Primary constructor: Example 2

/* Definition if the class PersonProfile

The primary constructor has four parameters.

Hence, all instances of this class have those four attributes/variables */

class PersonProfile(var name: String, var surname: String, var title: String, var age: Int)

26

Primary constructor: Example 2

```
object TestConstructor {
  def main(args: Array[String]): Unit = {
    /* Create an instance of PersonProfile */
    var pp=new PersonProfile("Paolo", "Garza", "Mr.", 40)
    /* Print a message about the user on the console */
    println("Hello "+pp.title+ " "+ pp.name + " " + pp.surname + " ")
    println("You are "+ pp.age + " years old")
  }
}
```

27

Primary constructor: Example 3

- Change the content of the class PersonProfile in order to automatically print the message “Hi <name>” every type a new instance of PersonProfile is instantiated
  - We can achieve this goal by means of a println in the body (code) of the primary constructor

28

Primary constructor: Example 3

```
class PersonProfile(var name: String, var surname: String,
var title: String, var age: Int) {
  println("Hi "+ name)
}
```

29

Primary constructor: Example 3

```
object TestConstructor {
   def main(args: Array[String]): Unit = {
      /* Create an instance of PersonProfile */
      var pp=new PersonProfile("Paolo", "Garza", "Mr.", 40)
      /* Print a message about the user on the console */
      println("Hello "+pp.title+ " "+ pp.name + " " + pp.surname + " ")
      println("You are "+ pp.age + " years old")
   }
}
```

30

Auxiliary constructors

- Every class can have multiple auxiliary constructors
- They are defined by means of the following syntax

```
def this([list of parameters]) = {
    /* call to the primary constructor or to another
       auxiliary constructor */
    /* code executed by the auxiliary constructor */
}
```

31

Auxiliary constructors

- Pay attention that all the parameters of the primary constructor must be initialized
  - Hence, call the primary constructor (or another auxiliary constructor) in the code of the auxiliary constructor to initialize all the parameters of the primary constructor
  - All the instances of the class have all the attributes associated with the primary constructor

32

Auxiliary constructor: Example

- Add an auxiliary constructor to the class PersonProfile
- The auxiliary constructor is characterized only by three parameters
  - Name
  - Surname
  - Title
- Age is set to -1 when a new instance of PersonProfile is created by using this auxiliary constructor

33

Auxiliary constructor: Example

```
class PersonProfile(var name: String, var surname: String, var title: String, var age: Int) {

  def this(name: String, surname: String, title: String) = {
    this(name, surname, title, -1)
  }

  println("Hi "+ name)
}
```

Auxiliary constructor

34

Auxiliary constructor: Example

```
class PersonProfile(var name: String, var surname: String, var title:
String, var age: Int) {

  def this(name: String, surname: String, title: String) = {
    this(name, surname, title, -1)
  }

  println("Hi "+ name)
}
```

Invoke the primary constructor

Auxiliary constructor: Example

```
object TestConstructor {
   def main(args: Array[String]): Unit = {
      /* Create an instance of PersonProfile by using the auxiliary
         constructor */
      var pp=new PersonProfile("Paolo", "Garza", "Mr.")
      /* Print a message about the user on the console */
      println("Hello "+pp.title+ " "+ pp.name + " " + pp.surname + " ")
      println("You are "+ pp.age + " years old")
   }
}
```

Auxiliary constructor: Example 2

- Add another auxiliary constructor to the class PersonProfile
- The second auxiliary constructor is characterized only by two parameters
  - Name
  - Surname
- Title is set to the empty string (“”) and Age is set to -1 when a new instance of PersonProfile is created by using this auxiliary constructor

37

Auxiliary constructor: Example 2

```
class PersonProfile(var name: String, var surname: String, var title: String,
var age: Int) {
  def this(name: String, surname: String, title: String) = {
    this(name, surname, title, -1)
  }
  def this(name: String, surname: String) = {
    this(name, surname, "")
  }
  println("Hi "+ name)
}
```

Second auxiliary constructor

38

Auxiliary constructor: Example 2

```
class PersonProfile(var name: String, var surname: String, var title: String,
var age: Int) {
  def this(name: String, surname: String, title: String) = {
    this(name, surname, title, -1)
  }
  def this(name: String, surname: String) = {
    this(name, surname, "")
  }
  println("Hi "+ name)
}
```

Invoke the other auxiliary constructor

39

Auxiliary constructor: Example 2

```
object TestConstructor {
   def main(args: Array[String]): Unit = {
      /* Create an instance of PersonProfile by using the auxiliary
         constructor */
      var pp=new PersonProfile("Paolo", "Garza")
      /* Print a message about the user on the console */
      println("Hello "+pp.title+ " "+ pp.name + " " + pp.surname + " ")
      println("You are "+ pp.age + " years old")
   }
}
```

40

No "static" in Scala

- Static methods and fields do not exists in Scala
- However, they can be "represented" defining methods and fields of Object ("singleton" classes)

41

Examples

- Scala:

```
object PersonUtil {
val AgeLimit = 18

def countPersons(persons:
List[Person]) = ...
}
```

- Java:

```
public class PersonUtil {
public static final int
AGE_LIMIT = 18;
public static int
countPersons(List<Person
>  persons) { ... }
}
```

42

Companion Objects

- If a class and an object are
  - Declared in the same file
  - Declared in the same package
  - And they have the same name
- they are called **companion class** and **companion object**, respectively

43

Companion Objects

- Companion objects can be used to define something similar to static methods and fields for a class
- The companion object defines all the "static" methods and fields of the companion class
- Companion objects can read the companion classes private fields

44

Companion Objects: Example

```
class Person(private val age: Int)
object Person {
  def getPersonAge(p: Person) = p.age
}
val personInstance = new Person(30)
val age = Person.getPersonAge(personInstance)
…
```

45

Companion object: apply method

- The apply method is usually defined in the companion object
- The apply method is used to create a new instance of the companion class without "explicitly" use the new keyword
  - The new keyword in used in the implementation of the apply method

46

Companion object: apply method

```
class Person private(val age: Int)
object Person {
  def apply(age: Int) = new Person(age)
}

var personInstance = Person.apply(30)
personInstance = Person(30)
```

47

Inheritance

- Scala is characterized by single class inheritance
  - i.e., each class or object can extend only one parent class
- However, it supports multiple inheritance by means of traits
  - Traits are similar to Java interfaces
  - They allow implementing "multiple inheritance"

48

Inheritance: Example

```
class MyClass(myString: String)

class MySubClass(myString: String, myInt: Int)
    extends MyClass(myString) {
    println("MyString: '" + myString + "', MyInt: '" +
myInt + "'")
}
```

49

Traits

- Traits are similar to Java interfaces
  - Unlike the other types traits cannot be instantiated
- They allow implementing "multiple inheritance"
  - Each class, object, or trait can extend many traits at the same time

50

Traits: Example #1

- Scala:

```
trait Shape {
    def area: Double
}

class Circle extends Object
with Shape
```

- Java:

```
interface Shape {
    public double area();
}

public class Circle extends
Object implements Shape
```

51

Traits: Example #2

```
trait Ordered[A] {
  def compare(that: A): Int
  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
}
```

52

Traits: Example #2

```
trait Ordered[A] {
  def compare(that: A): Int
  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
}
```

Abstract method that must be implemented by the classes implementing this trait

53

Traits: Example #2

```
class Person(private val age: Int) extends
Ordered[Person]{
  def compare(other: Person) = this.age - other.age
}
....
```

Implementation of the compare method of trait Ordered

54

Traits: Example #2

```
….
val person1 = new Person(21)
val person2 = new Person(31)
println(person1 < person2)  // true
println(person1 <= person2) // true
println(person1 >= person2) // false
```

55

Dynamic mixins

```
class Person(age: Int) {
  override def toString = "my age is " + age
}

trait MyTrait {
  override def toString = "I am sure " + super.toString
}
```

56

Dynamic mixins

```
...
/* A new instance of Person implementing MyTrait */
val person = new Person(30) with MyTrait
println(person)
/* => I am sure my age is 30 */
/* The method toString of MyTrait is used */
```

57

Miscellaneous

Packages

- Scala:
- package mypackage
- ...

- Java:
- package mypackage;
- ...

59

Imports

- Scala:
- import java.util.{List, ArrayList}
- import java.io._

- Java:
- import java.util.List
- import java.util.ArrayList
- import java.io.*

60

Exceptions

- Scala:
- throw new Exception("...")
- Java:
- throw new Exception("...")

61

Exceptions

- Scala:
- try {
- } catch {
- case e: IOException => ...
- } finally {
- }
- Java:
- try {
- } catch (IOException e) {
- ...
- } finally {
- }

62