

# PROGRAMMING III

# JAVA LANGUAGE

**COURSE 2. CLASSES & OBJECTS**

# COURSE CONTENT

## ☐ **Classes**

- ☐ Class modifiers
- ☐ Fields modifiers
- ☐ Method modifiers

## ☐ **Objects**

## ☐ **Display objects**

- ☐ `toString()` method
- ☐ `StringBuffer` class
- ☐ `StringBuilder` class

## ☐ **Organize classes**

- ☐ Packages
- ☐ Modules ( $\geq$  Java 1.9)

# CLASSES

## ❑ Classes

- ❑ Groups objects with similar characteristics

## ❑ Syntax

```
[classModifier] class ClassName [extends  
BaseClassName] [implements Interface1 [, Interface2]  
...[, InterfaceN]...]{  
    member fields and methods  
}
```

*Can appear only once in class declaration  
Types: public, abstract, final*

*Variable name that starts with upper case  
(does not contain spaces)*

# CLASS MODIFIERS

## □ `public`

- Class is **visible in all packages**
- The **name of the class** has to be the **same** like the **name of the file**

## □ `abstract`

- Used for classes that contain **abstract methods**
- Used for classes that inherits abstract methods from a base class
  - If the class does not implement all the methods exposed by an interface
- Can **not be instantiated**

## □ `final`

- The class **definition is complete**
- The class **cannot be base class** for other classes
  - A class cannot be in the same time final and abstract

# OBJECT CLASS

## ❑ All Java classes are inherited from Object class

- ❑ package: `java.lang`
- ❑ Every class is directly or indirectly derived from the `Object` class.
- ❑ If a class does not extend any other class then it is direct child class of `Object` and if extends other class then it is an indirectly derived.

## ❑ Some of most common used Object class methods

- ❑ `protected void equals(Object obj)`
  - ❑ Tests if the current object is equal with the one passed like parameter
- ❑ `protected void finalize()`
  - ❑ Method called by *garbage collection* when there is no reference to the current object
- ❑ `public Class getClass()`
  - ❑ Method that returns the current class of the object
- ❑ `public int hashCode()`
  - ❑ Homework: which is the role of `hashCodeMethod()`
- ❑ `public String toString()`
  - ❑ Returns the string representation of the object

# CLASSES

## ❑ Classes

- ❑ Groups objects with similar characteristics

## ❑ Syntax

```
[classModifier] class ClassName [extends  
BaseClassName] [implements Interface1 [, Interface2]  
...[, InterfaceN]...]{  
    member fields and methods  
}
```

- ❑ Class member fields/variables/attributes
- ❑ Describe the properties of a class

# CLASS. MEMBER ATTRIBUTES

## □ Syntax

- `[fieldsModifier] variableType variableName [, variableName1 ...[, variableNameN]];`
- `[fieldsModifier] variableType variableName [=variable initialization];`
- `[fieldsModifier] variableType variableName [ ] [=variable initialization];`

## □ Attributes modifiers

### □ Access modifiers

- `public, protected, private`
- If no modifier is specified for a class attribute then the default specifier package is used

### □ Others

- `final, static, transient, volatile`

# MEMBER ATTRIBUTES MODIFIERS

## ☐ Access modifiers

☐ `public`

☐ Visible **all classes and packages**

☐ `protected`

☐ Visible in **derived classes**

☐ `implicit/default`

☐ Visible in **all classes in same package**

☐ `private`

☐ Visible in **current class**



# MEMBER FIELDS

## MODIFIERS

### ☐ Others

#### ☐ `final`

- ☐ **Constants** - the value of the attribute is the same during the whole program execution
- ☐ In many cases is used with static modifier
- ☐ Constants in Java are written with upper cases
- ☐ Must be initialized when they are declared

#### ☐ `static`

- ☐ Allocates a **single memory location** that is **shared** by **all class objects**
- ☐ Accessible by class name

#### ☐ `transient`

- ☐ Variables that does not persist (are not serializable)

#### ☐ `volatile`

- ☐ The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory";
- ☐ Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

# CLASS MEMBERS TYPES

## ❑ Local variables

- ❑ Variables **defined inside methods**, constructors or blocks are called local variables.
- ❑ The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

## ❑ Instance variables

- ❑ Instance variables are variables within a class but **outside any method**.
- ❑ These variables are **initialized** when the **class** is **instantiated**.
- ❑ Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

## ❑ Class variables

- ❑ Class variables are variables declared within a class, **outside any method**, with the **static** keyword.

# CLASS MEMBER FIELDS

## □ Example

```
public class ExVariables {  
    public static final int MAXIMUM_CAPACITY = 100;  
    int age;  
    public String name;  
    transient double mean;  
    protected double marks[];  
    private int I, j, k=9;  
    private double b[] = new double [10];  
  
    public static void main(String []args){  
        ExVariables obj = new ExVariables()  
        obj.name = "Course Java";  
        System.out.println(ExVariables.MAXIMUM_CAPACITY);  
        ...  
    }  
}
```

*Instance variable*



*Static variable*



# CLASS MEMBERS

## `this` keyword

- A reference to the current object
- Cannot be accessed in a static context

## `super` keyword

- A reference to base class
- Cannot be accessed in a static context


# CLASSES

## ❑ Classes

- ❑ Groups objects with similar characteristics

## ❑ Syntax

```
[classModifier] class ClassName [extends  
BaseClassName] [implements Interface1 [, Interface2]  
...[, InterfaceN]...] {  
    member fields and methods  
}
```

- 
- ❑ Class member methods/functions
  - ❑ Describe the behavior of an object of the class

# CLASS METHODS

## □ Syntax

```
[methodModifiers] returnType methodName  
([parameter list]) [throws Exception1[, ...,  
ExceptionN]] { ...}
```

*public, protected,  
private, default (package),  
abstract, final, static,  
synchronized, native*

*formal parameters list*

*void, primitive or reference type*

# CLASS METHODS MODIFIERS

## ☐ Access modifiers

- ☐ `public`

- ☐ Visible **all classes** and **packages**

- ☐ `protected`

- ☐ Visible in **derived** classes

- ☐ `Default/package`

- ☐ Visible in all classes in **same package**

- ☐ `private`

- ☐ Visible in **current class**

# CLASS METHODS MODIFIERS

## ☐ Others

- ☐ abstract
  - ☐ Offers only the signature of the method
  - ☐ The method does not provide an implementation
  - ☐ **Cannot be:** private, static, final, native or synchronized
- ☐ static
  - ☐ Class method
  - ☐ Does not have access to `this` reference
- ☐ final
  - ☐ **Cannot be overwritten**
- ☐ synchronized
  - ☐ Only one **thread** can access the method when is executed
- ☐ native
  - ☐ A **native method** in other programming language (like C, C++)



# JAVA METHODS WITH VARIABLE ARGUMENTS LENGTH

## ❑ Example

```
class X {  
    void method1 (int a, String ... words) {  
        for (String s: words) {  
            System.out.println("argument: " + s);  
        }  
    }  
    void method2 (double ... numbers) { }  
}
```

## ❑ Properties

- ❑ It must be the **last argument** of the method
- ❑ The argument is an **array of objects** of the type of the argument

## ❑ Call

- ❑ method1 (10)
- ❑ method1 (10, "s1", "s2")
- ❑ method1 (10, "s1", "s2", "s3")
- ❑ ...
  
- ❑ method2 ()
- ❑ method2 (4.5)
- ❑ method2 (5.7, 7.8)
- ❑ ...

# CONSTRUCTORS

## □ Properties

- Function that an object calls when an **object** is **instantiated**
- Has the **same name** like the **class**
- It does **not** have a **return value**
- If no constructor is defined a default constructor is provided by the compiler

## □ Remark

- No constructor is called when the object is declared, the constructor is called when the object is instantiated using `new` operator

## □ Example

```
public class X {  
    int y;  
    public X() { }  
    public X(int v) {  
        this.v = x;  
    }  
}
```

Default constructor

Constructor with parameters

Object declaration and instantiation

Constructor call:

```
X obj1 = new X();  
X obj2 = new X(3);  
X obj;
```

Object declaration

# CLONABLE AND COPY CONSTRUCTOR

## ❑ Object copy

## ❑ Copy constructor

❑ `X(X obj)`

## ❑ Cloneable interface

❑ `clone(Object x)`

❑ If the class contains an **array** of objects then the `clone()` method has to be **overwritten** in order to provide a **deep copy** of the object

```
class X implements Cloneable {  
    public Object clone() throws  
        CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

*obj1 and obj2 reference  
same memory location*

```
X obj1 = new X();  
X obj2 = obj1;  
X obj3 = obj1.clone();
```

*obj1 and obj3 reference to  
different memory locations*

```
Person p("maria");
```

```
Person p1 = p;
```

```
Void fct(Person p);
```

```
Fct(p1)
```

```
Person fct1(){
```

```
    return Person("mihai");
```

```
}
```

# OBJECTS

- ❑ Objects have states and behaviors

- ❑ Objects creation steps

- ❑ **Declaration**

- ❑ A variable declaration with a variable name with an object type.

- ❑ **Initialization**

- ❑ The `new` keyword is followed by a call to a constructor.

- ❑ This call initializes the new object.

# OBJECTS & ARRAYS

## ❑ Declare an array

- ❑ `String [] s;`

## ❑ Declare and allocate space

- ❑ `String s[] = new String[3];`

## ❑ Declare and initialize

- ❑ `String s[] = {"Java", "Course", "2"}`

## ❑ Looping through an array

- ❑ 

```
for (String ss : s)
    System.out.println(ss);
```
- ❑ 

```
for (int i=0; i < s.length; i++)
    System.out.println(s[i]);
```

# TRANSFORMING OBJECTS TO STRING

❑ **Overwrite `toString()` method inherited from `Object` class**

❑ `toString()` is automatically called when an object is transformed to string

## ❑ Variants

### ❑ `String` class

❑ Pro: **Easy** to implement

❑ Cons: `String` immutable -> a **lot of** temporary **objects** created

### ❑ `StringBuilder`

❑ mutable objects in java and provide `append()`, `insert()`, `delete()` and `substring()` methods for `String` manipulation.

❑ Is **not** thread safe and synchronized

❑ Recommended for non-multi threaded environment

### ❑ `StringBuffer`

❑ mutable objects in java and provide `append()`, `insert()`, `delete()` and `substring()` methods for `String`


❑ is thread safe and synchronized

# TRANSFORMING OBJECTS TO STRING

## STRING

```
class Person{
    String name;
    int age;
    Person (int age, String name){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        return "Name:" + name + "age: " + age;
    }
}
```

How many objects of type  
String are created?



## STRINGBUILDER

```
class Person{
    String name;
    int age;
    Person (int age, String name){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        StringBuilder sb = new StringBuilder();
        sb.append("Name:").append(name);
        sb.append("Age:").append(age);
        return sb.toString();
    }
}
```



# ORGANIZE JAVA CLASSES

## ❑ Packages

- ❑ Organize closely related java **classes**

## ❑ Modules

- ❑ Organize closely related **packages** and **resources**

# PACKAGES

- ❑ **Groups a collection of related classes that form a library**
  - ❑ Avoid **name-clashes**.
  
- ❑ **Physical organization**
  - ❑ .java and .class files in a **directory tree** that mimics package structure
  
- ❑ **E.g. for the class called A.B.SomeClass, the files will be:**
  - <sourceroot>/A/B/SomeClass.java
  - <classroot>/A/B/SomeClass.class
  
- ❑ **Only **public** classes from a package are **visible outside the package****

# PACKAGES

## ❑ Package names

- ❑ Package names are separated by periods
- ❑ Packages can contain classes or packages
- ❑ The **fully qualified name** of a class **includes** its **package** name:
  - ❑ A.B.SomeClass indicates a class called "SomeClass" within the "A.B" package.

## ❑ To put a class into a package, one uses the "package" statement

- ❑ The package statement **must be** the **first line of code** within the file.
- ❑ It can be preceded by comments.

## ❑ If no package statement is supplied, the class is placed in the **"default" package**

- ❑ The default package is a package with **no name**

# IMPORTS

- ❑ To avoid having to use the fully qualified class name for all classes

- ❑ **Can import**

- ❑ Some of the classes from a given package

- ❑ `import java.util.Random;`

- ❑ All classes from a package

- ❑ `import java.util.*;`

- ❑ **Static import**

- ❑ Static import allows you to refer to the members of another class without writing that class's name

- ```
import static java.lang.Math.*;
double angle = sin(PI / 2) + log(E * E);
```

*Random class can be used  
without the fully qualified name*

*Imports other classes that are  
not used in the application*

# JAR FILES

## **JAR: Java ARchive.**

- A group of Java classes and supporting files combined into a single file compressed with ZIP format, and given .JAR extension.

## **Advantages of JAR files:**

- compressed; quicker download
- just one file; less mess
- can be executable

# JAR FILES

## ❑ Creating a JAR archive

### ❑ Command line

- ❑ `jar -cvf filename.jar files`

### ❑ Example

- ❑ `jar -cvf MyProgram.jar *.class *.gif  
*.jpg`

### ❑ Using IDEs (e.g. Eclipse) can create JARs automatically

- ❑ File → Export... → JAR file

## ❑ Running a Jar from command line

- ❑ `java -jar filename.jar`

# JAR FILES

## ❑ Making a runnable JAR

- ❑ **manifest file**: Used to create a JAR runnable as a program.

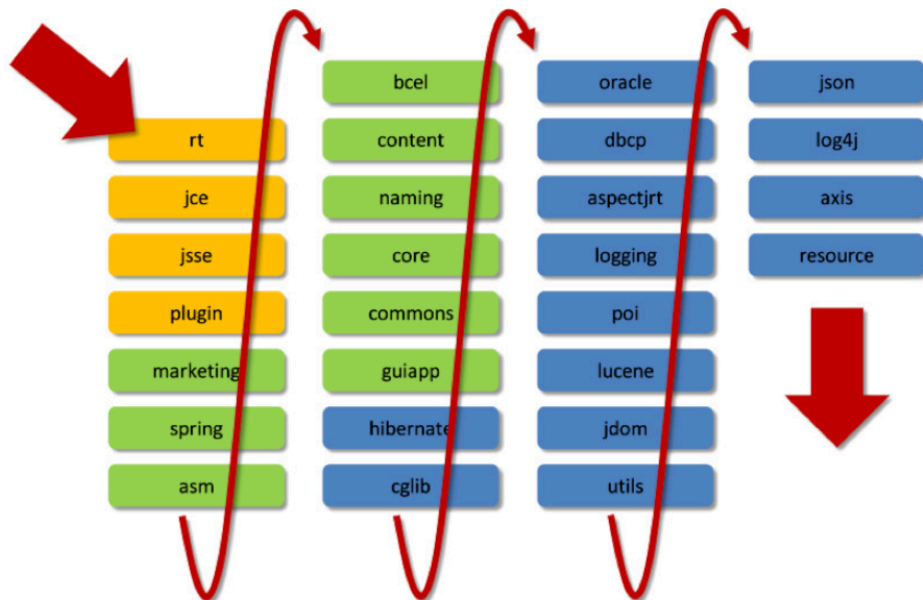
- ❑ Contents of MANIFEST file

  - ❑ Main-Class: MainClassName

- ❑ Eclipse will automatically generate and insert a proper manifest file into your JAR if you specify the main-class to use.

# MODULES

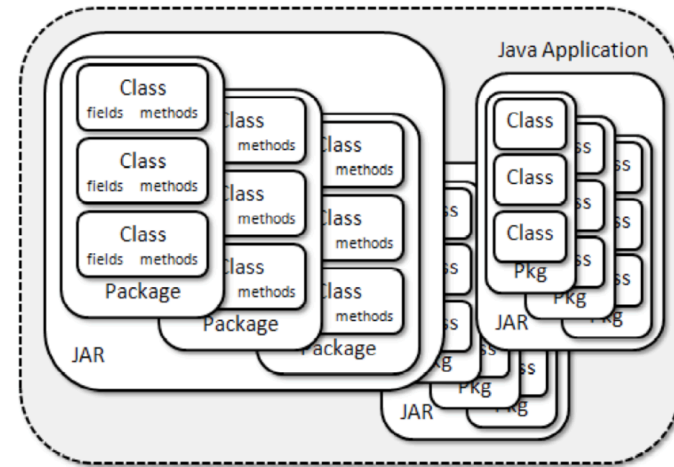
## JAR DEPENDENCES



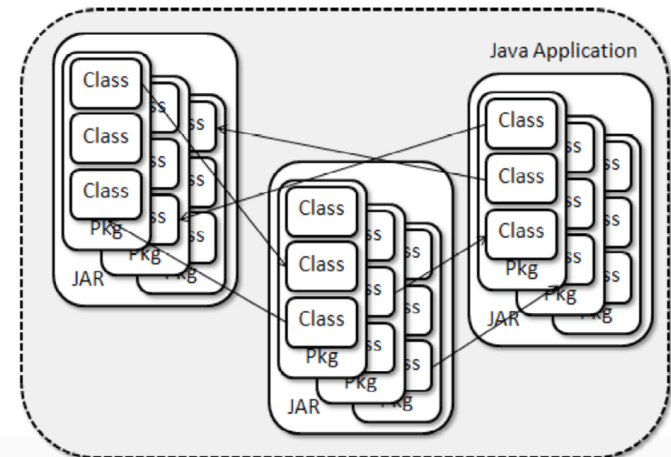
Building tools

- Automatically resolve dependencies
- Ant, Maven, Gradle, ...

## JAVA APPLICATION



## JAVA APPLICATION DEPENDENCES





# MODULES

- ❑ “package of Java Packages” abstraction that allows to make code even more reusable.
- ❑ Module file descriptors
  - ❑ **Name**
    - ❑ the name of the module
  - ❑ **Dependencies**
    - ❑ a list of other modules that this module depends on
  - ❑ **Public Packages**
    - ❑ a list of all packages we want accessible from outside the module
  - ❑ **Services Offered**
    - ❑ can provide service implementations that can be consumed by other modules
  - ❑ **Services Consumed**
    - ❑ allows the current module to be a consumer of a service
  - ❑ **Reflection Permissions**
    - ❑ explicitly allows other classes to use reflection to access the private members of a package

# MODULES

## ❑ Example

```
module Provider {
    requires ServiceInterface;
    provides javax0.serviceinterface.ServiceInterface
        with javax0.serviceprovider.Provider;
}
module Consumer {
    requires ServiceInterface;
    uses javax0.serviceinterface.ServiceInterface;
}
module ServiceInterface {
    exports javax0.serviceinterface;
}
```