Advanced Programming Methods

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

Basic concepts in OOP - Recap

- **Class**: represents a data type.
- **Object**: an instance of a class.
- **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects.
- **Abstraction**: separating an object's *specification* from its *implementation*.
- **Inheritance**: allowing code to be reused between related types.
- **Polymorphism**: allowing an object to be one of several types, and determining at runtime how to "process" it, based on its type.

Declaration/definition of a class in Java

```
[public] [abstract] [final] class ClassName
{
    [member data (attributes) declarations]
    [method declarations and implementations]
    [nested classes]
}
```

- If the class (ClassName) is public then it must be in a file called *ClassName.java*.
- A .java file may contain several class definitions, but only one can be public.
- As opposed to C++ classes:
  - There is no need for 2 separate files (.h and .cpp).
  - Methods are implemented where they are declared.

Declaring data members I

```
[public] [abstract] [final] class ClassName
{
    [access_modifier] [abstract] [static] [final]
                                    Type name [=initial_value];
}
```

- **access_modifier** can be: private, protected, public
- The default access modifier in Java is package private.

Declaring data members II

```
public class Doctor {
    private String name;
    private String speciality;
    private double salary;

    // ...
}
```

```
public class Point
{
    private double x;
    private double y;

    //...
}
```

Declaring data members III

```
public class Circle {
    double radius;
    Point centre;
    static final double PI = 3.14;

    //...
}
```

Attribute initialisation

- Attributed can be initialised:
  - In their declaration:

```
private double radius = 0;
```

  - In the class constructor.
  - In initialisation blocks:

```
public class Rational
{
    private int numerator;
    private int denominator;

    {
        numerator = 0
        denominator = 1;
    }
}
```

  - If an attribute is not explicitly initialised, its value will be the implicit value of its type.

Constructors

- A constructor is called when an instance of the class is created.
- When calling the constructor, memory for the object is allocated.
- It must always have the exact same name as the class.
- It has no return type.
- *If a class does not declare any constructors a default constructor is automatically generated, with the access modifier public.*

```
[...] class ClassName
{
    [access_modifier] ClassName([parameter_list])
    {
        // constructor body
    }
}
```

Constructor overloading I

- Having more constructors, with different parameters.
- Each constructor performs a different task, depending on the given situation.

```java
public class Rational {
    private int numerator;
    private int denominator;

    public Rational()  {
        this.numerator = 0;
        this.denominator = 0;
    }

    public Rational(int num) {
        this.numerator = num;
        this.denominator = 1;
    }
```

Constructor overloading II

```
public Rational(int num, int den) {
    this.numerator = num;
    this.denominator = den;
}
```

- A constructor can call another constructor using this.
- A call to this must be the first statement in the constructor body.

```
public Rational(int num) {
    this(num, 1);
}
```

Constructor overloading III

- It is not possible to call two different constructors.

```
public Rational(int num) {
    this(num, 1);
    this(1, 0); // ERROR
}
```

- A constructor cannot be directly called from another method within the same class.

```
public Rational add(Rational r)  {
this(this.numerator + r.numerator,
    this.denominator + r.denominator); // ERROR
return this;
}
```

Object creation

- When the program is executed objects created with the new operator are allocated in the heap.
- We get references to the created objects.
- null - can be used to initialise a reference (it will have no object associated).

```
 public static void main(String[] args) {
    Rational r1 = new Rational(1, 2);
    Rational r2 = new Rational(3);
    Rational r3 = null; // r3 has no object associated

    r1 = r2; // both r1 and r2 refer the same object
    System.out.println(r1);
}
```

Object destruction

- There are no destructors in Java.
- The memory is deallocated by the *garbage collector*.
- The garbage collector is always running in the background.
- Its main objective is to free heap memory by destroying unused/unreferenced objects.

Method declaration

```
[...] class ClassName
{
    [access_modifier][method_attributes]
                    Type methodName ([parameter_list])
    {
        // method body
    }
}
```

- Type can be any primitive type, class, array or void.
- The default access modifier is package private.

Methods with a variable number of arguments

- Variable arguments (varargs) feature: the method can be called with zero or more arguments.
- While defining method signature, varargs must always be last.
- A method can have only one varargs parameter.

```
[access_modifier] methodName(type ... args) {
    // method body
}
```

Example

Variable arguments (class Varargs).

Method overloading

- Several methods with the same name, but with different signatures.
- Method overloading is an example of static polymorphism (compile-time binding or early binding).
- The binding of a method call to its definition is made at compile time.
- Overloading:
  - different number of parameters;
  - different data types;
  - method return type does not matter in case of overloading.

Methods toString(), equals() and hashCode()

- Their signatures must not be changed when implementing a new class.
- toString() returns the string representation of the class.
- The Java compiler internally invokes toString() when printing an object of the class.
- equals() allows checking the equality of the objects, not the references.
- bluehashCode() returns an integer value used in hash-based collections.
- Implement equals() and hashCode() together.

Example

Rational.java.

Static members I

- Static attributes: only one copy of each class variable per class, regardless of how many objects are created from it.
- Memory is allocated only once for a static data member, regardless of the number of objects of that class that have been created.

```
public class Rational {
    private int numerator;
    private int denominator;

    static double PRIMEGAME_FIRST = 17/91;
    // John H. Conway's prime producing machine
    // ...
}
```

- They can be referred to using the class name or using an object.

Static members II

```
System.out.println(Rational.PRIMEGAME_FIRST);
System.out.println(r1.PRIMEGAME_FIRST);
```

- Static methods (class methods): are not specific to objects, they are associated to the class.
- They are shared among all objects created from the same class.
- A static method cannot use non-static members of the class (attributes or methods).
- Static methods can only call other static methods or work with static attributes.
- A static method cannot use the this keyword.

Example

MathUtils - Rational.java.

Inheritance

- *Is-a* relationship.
- The base class makes its structure and behaviour available to its derived classes.
- In Java it is realised using the keyword extends.

```
class Base {
    // ...
}

class Derived extends Base {
    // ...
}
```

- Java does not allow multiple inheritance.

Constructors in inheritance

- If no base class constructor is explicitly invoked in the derived class, the default constructor from the base class is invoked automatically.
- If there are no default constructors in the base class, the derived class constructor must explicitly call any of the existing constructors in the base class.
- Calling the base class constructor is achieved using the super keyword.
- The base class constructor call (using super) must be the first instruction in the derived class constructor.
- To refer member data/functions of the base class within the derived class: super.member or super.memberFunction() (must be protected or public).

Example

Inheritance.java.

The Object superclass

- In Java any class is derived from the *Object* class.
- Methods of the *Object* class:
  - **toString()**: returns the string representation.
  - **equals()**: checks object equality.
  - **hashCode()**: returns the hash code of an object. equals() and hashCode() must use the same set of fields; if two objects are equal, their hash codes must be equal as well.
  - **getClass()**: returns the object's class.
  - **clone()**: creates a copy of the object
  - **finalize()**: called by the garbage collector before object destruction.
  - ... and others.

Method overriding

- Use the **@Override** annotation:
  - Ensures compiler checking.
  - Makes the code easier to understand.
- Static methods cannot be overriden.
- The return type of an overridden method can be a subtype of the type returned by the base class (*covariant return type*).

Example

Inheritance.java.

Polymorphism

```
Animal p3 = new Penguin("white and black", 8, "Emperor");
System.out.println(p3.toString());
```

- In Java all methods (except static or private or final) are implicitly virtual.
- All objects in Java are polymorphic, since they pass the "is-a" test for the Object type.

The *final* keyword

- *Final methods*: cannot be overridden in derived classes.
- *Final classes*: cannot be derived.
- *Final attributes*: must be initialised when they are declared or in an initialisation block. The value of a final variable cannot be changed.

Example

FinalExample.java.

Abstract classes I

- Are used to define abstract concepts.
- A method is *abstract* if it is declared but not defined. An abstract method is declared using the keyword abstract.

```
[access_modifier] abstract
                           Type name([parameters_list]);
```

- An abstract class is a class that may contain abstract methods (it is not compulsory).
- If a class has at least one abstract method, the class has to be declared abstract.

Abstract classes II

- An abstract class is defined using the keyword abstract.

```
abstract class ClassName {
    // ...
}
```

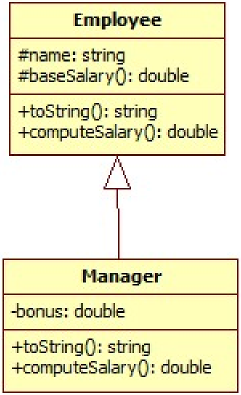
- An abstract class cannot be instantiated unless the abstract methods are defined (even at object creation).
- If a derived class does not define all the abstract base class methods, it also has to be declared abstract.

Example

AbstractClasses.java.

Exercise I

Write the Java code corresponding to the following UML class diagram related to companies and their employees.



Exercise II

- The company has several employees, some of them are managers.
- The **toString** method from Employee returns a string with the name of the employee.
- The **toString** method from Manager returns a string with the word "Manager" and the name of the employee.
- The **computeSalary** method from Employee returns the base salary.
- The **computeSalary** method from Manager returns the base salary, to which the manager bonus is added.

Exercise III

- Write a test program that creates several employees (both regular employees and managers), add all the employes into a list (vector).
- Create a function that for a list of employees will print out the proper name and salaries for all the employes, using the values returned by the **toString** and **computeSalary** methods.

Interfaces

- An interface is similar to an abstract class, except all methods are abstract.
- Declaration: using the keyword interface.

```
public interface InterfaceName {
    [method declaration];
}
```

- An interface contains only method declarations (no implementations).
- An interface does not have constructors.
- All methods in an interface are implicitly public.
- Any attribute declared in an interface is implicitly public, static and final.

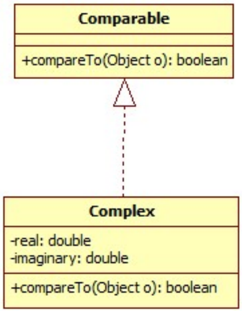
Interface implementation I

- A class that defines the methods declared in an interface will *implement* that interface.
- This is declared by using the keyword implements.

```
[public] class ClassName implements InterfaceName
    [implementations of interface methods]
    // other definitions
}
```

- The class must implement all methods in the interface.
- If there is at least one method that is not implemented, then the class must be declared abstract.

Interface implementation II



Interface implementation III

- An interface can extend one or more other interfaces and add new methods (keyword extends).
- A class can implement multiple interfaces.
- The class will then have to implement all methods from all interfaces. Otherwise, it has to be declared abstract.
- When implementing multiple interfaces, be careful of collisions.
- It is possible to declare an object having an interface as type.
- Such an object can be initialised with any object whose class implements the interface.
- When using such objects only the methods from the interface can be accessed.

Interface implementation IV

- Before Java 8 interfaces could only have abstract methods, which had to be implemented by the classes implementing the interface (unless the classes were abstract).
- **Problem**: if the interface was changed (e.g. a new method was added), all classes implementing the interface had to provide an implementation for that method.
- Since Java 8 interfaces can have *static* and *default* methods: the interface provides a default implementation. Thus the implementation classes are not affected.

Interface implementation V

```
[public] default Return_type
                        method([parameters_list]) {
    // default implementation
}

[public] static Return_type
                        method([parameters_list]) {
    // implementation
}
```

- Default methods can be overridden in classes implementing the interface.
- Static methods should contain the complete definition of the function and cannot be overridden.

Abstract classes vs. interfaces

Abstract class	Interface
May contain methods with any type of access modifier.	May only declare public methods.
May declare and define any attributes.	All attributes are implicitly static and final.
May have constructors.	Does not have constructors.
Might not have any abstract methods.	Non-default and non-static methods are abstract.
Cannot be instantiated.	Cannot be instantiated.

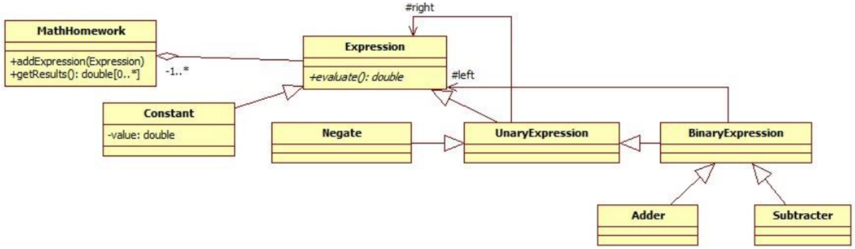
The **instanceof** operator

- Is used to test whether an object is an instance of a specified type.
- It allows testing if an object is an instance of a class, of a derived class or of a class that implements an interface.
- *instanceof* applied on a null variable returns false.

Example

Interfaces.java.

Interfaces - exercise I



Interfaces - exercise II

Write an application for computing mathematical expressions, as follows:

- The interface **Expression** contains the method *evaluate()*.
- The class **Constant** represents a constant and contains a value. The evaluation result of a constant expression is its own value.
- The classes **UnaryExpression** and **Negate** each aggregate an expression and the first returns the evaluation of the aggregated expression, while the second returns the negation of the evaluation of the aggregated expression.
- The class **BinaryExpression** aggregates two other expressions.
- The classes **Adder** and **Subtracter** are evaluated as the sum and the difference, respectively, of their aggregated expressions.

Interfaces - exercise III

- The class **MathHomework** allows the creation of a homework with several expressions that need to be evaluated. The method *getResults()* will return all the results of the contained expressions.
- Write a function which creates homework that must evaluate the following two expressions: -5 + (9 - 3) and -(4 + 2) - (-10). Print the results of the evaluation.

Summary

- Classes and objects in Java.
- Inheritance and polymorphism.
- Abstract classes and interfaces.
- *Next week*:
    - Packages.
    - Nested classes.
    - Generic types.
    - Java Collections Framework.