Advanced Programming Methods

Lecture 1 – Java Basics

Course Overview

Object-oriented languages:

- Java (the first 10 lectures): Basics, Generics, Collections, IO, Functional Programming, Concurrency, GUI, Metaprogramming

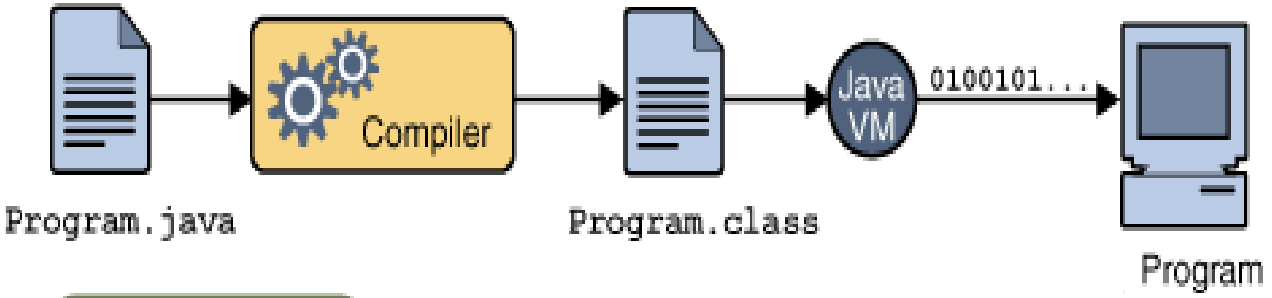
- C# (the last 3 lectures): Basics, Collections, IO, Linq

- perhaps something about Scala(NOT Sure)

Java References

- Bruce Eckel, *Thinking in Java*
- The Java Tutorials, 2019.
  https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html
- Java 13 API, 2019.
  https://docs.oracle.com/en/java/javase/13/docs/api/index.html
- Any Java book, tutorial,...

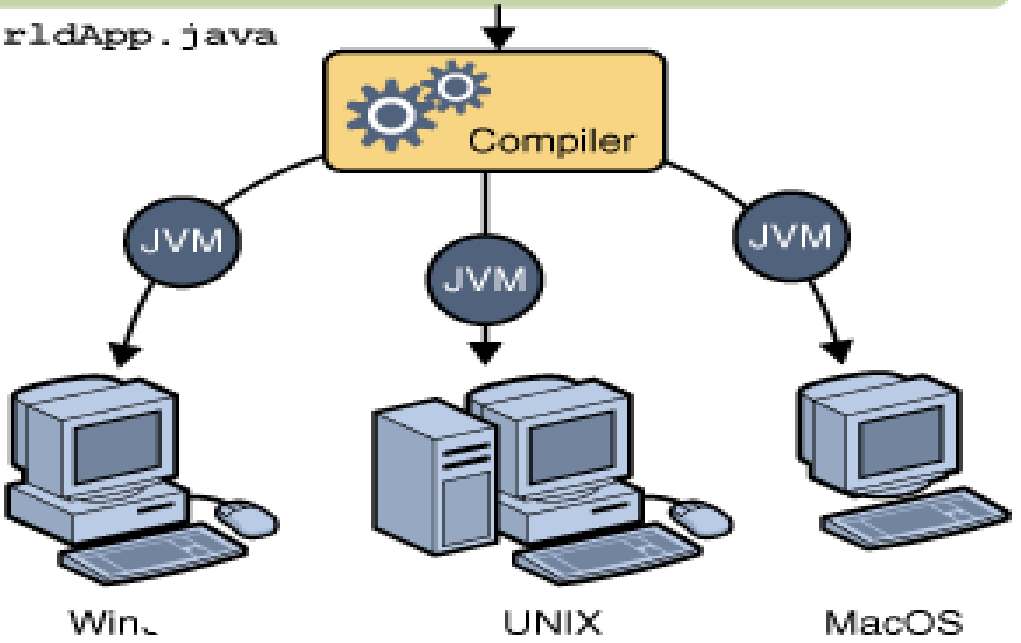
Java Technology



Java Program

```
class HelloWorldApp {
      public static void main(String[] args) {
            System.out.println("Hello World!");
      }
}
```

HelloWorldApp.java



A simple Java program

```
//Test.java
    public class Test {
        public static void main(String[] args) {
            System.out.println("Hello");
            for(String el : args)
            System.out.println(el);
        }
    }
```

Compilation:

```
javac Test.java
```

Execution:

```
java Test

java Test ana 1 2 3
```

!!! You can use int value=Integer.parseInt(args[i]) in order to transform a string value into an int value.

Lecture Overview

1. Recap OO concepts
2. Java classes and objects
3. Java Primitive Data
4. Passing Arguments in Java
5. Java Arrays
6. Java Char and String
7. Java code reusing: composition and inheritance
8. Java polymorphism: abstract classes, interfaces

Object-oriented programming Concepts

Class: represents a new data type

- Corresponds to an implementation of an ADT.

Object: is an instance of a class.

- The objects interact by messages.

Message: used by objects to communicate.

- A message is a method call.

Encapsulation(hiding)

- data (state)
- Operations (behaviour)

Inheritance: code reusing

Polymorphism – the ability of an entity to react differently depending on the context

Java Classes and Objects

- Class Declaration/Definition:

```
//ClassName.java
   [public] [final] class ClassName{
   [data (fields) declaration]
   [methods declaration and implementation]
   }
```

1. A class defined using `public` modifier it is saved into a file with the class name `ClassName.java.`
2. A file `.java` may contain multiple class definitions, but only one can be public.
3. Java vs. C++:
   - No 2 different files (.h, .cpp).
   - Methods are implemented when are declared.
   - A class declaration does not end with `;`

Java Classes and Objects

- Examples:

```
//Persoana.java
public class Persoana{
//...
}

// Complex.java
class Rational{
//...
}

class Natural{
//...
}

public class Complex{
//...
}
```

Java Classes and Objects

- Class Members (Fields) declaration:

```
... class ClassName{
    [access_modifier][static][final] Type name[=init_val];
}
```

`access_modifier` can be `public, protected, private`.

1. Class members can be declared anywhere inside a class.
2. Access modifier must be given for each field.
3. If the access modifier is missing, the field is visible inside the package (directory).

Java Classes and Objects

- Examples:

```
//Persoana.java
 public class Persoana{
    private String nume;
    private int varsta;
 //...
 }

 //Punct.java
 public class Punct{
 private double x;
 private double y;
 //...
 }
```

Java Classes and Objects

- Initializing fields
  - at declaration-site:

```
private double x=0;
```

  - in a special initialization block:

```
public class Rational{
        private int numarator;
        private int numitor;
        {
            numarator=0;
            numitor=1;
        }
        //...
}
```

  - in constructor.

Any field that is not explicitly initialized will take the default value of its type.

Constructors

- The constructor body is executed after the object memory space is allocated in order to initialize that space.

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

```
acces_modifier ∈ {public, protected, private}
```

```
list_formal_parameters takes the following form:
   Type1 name1[, Type2 name2[,...]]
```

1. The constructor has the same name as the class name (case sensitive).
2. The constructor does not have a return type.
3. For a class without any declared constructor, the compiler generates an implicit public constructor (without parameters).

Overloading Constructors

- A class can have many constructors, but they must have different signatures. .

```
//Complex.java
public class Complex{
  private double real, imag;

public Complex(){       //implicit constructor
 real=0;
  imag=0;
}

  public Complex(double real){
     this.real=real;
     imag=0;
  }

  public Complex(double real, double imag){ //...
   }

  public Complex(Complex c){ //...
   }
}
```

this

- It refers to the current (receiver) object.
- It is a reserved keyword used to refer the fields and the methods of a class.

```
//Complex.java
public class Complex{
  private double real, imag;
  //...
  public Complex(double real){
    this.real=real;
    imag=0;
  }

  public Complex(double real, double imag){
      this.real=real;
      this.imag=imag;
  }

  public Complex suma(Complex c){
  //...
  return this;
  }
}
```

Calling another constructor

- `this` can be used to call another constructor from a given constructor.

```
//Complex.java
public class Complex{
  private double real, imag;

 public Complex(){
  this(0,0);
  }

 public Complex(double real){
     this(real,0);
  }

public Complex(double real, double imag){
     this.real=real;
     this.imag=imag;
  }
  //...
 }
```

Creating objects

- Operator **`new:`**

```
Punct p=new Punct();   //the parentheses are compulsory
Complex c1=new Complex();
Complex c2=new Complex(2.3);
Complex c3=new Complex(1,1.5);

Complex cc; //cc=null, cc does not refer any object
cc=c3;      //c3 si cc refer to the same object in the memory
```

1. The objects are created into the heap memory.
2. The operator new allocates the memory for an object;

Static fields

```
public class Natural{
   private long val;
  public static long MAX=232.... //2^63-1
 //....
}
public class Produs {
    private static long counter;
    private final long id=counter++;
    public String toString(){
        return ""+id;
    }
   //....
```

Static fields are shared by all class instances. They are allocated only once in the memory.

Defining methods

```
[...] class ClassName{
   [access_modifier] Result_Type methodName([list_formal_param]){
       //method body
   }
}
```

`access_modifier ∈ {public, protected, private}`

`list_formal_param` **takes the form** `Type1 name1[, Type2 name2[, ...]]`

`Result_Type` poate can be any primitiv type, reference type, array, or `void.`

1. If the access_modifier is missing, that method can be called by any class defined in that package (director).
2. If the return type is not `void,` then each execution branch of that method must end with the statement `return.`

Defining methods

```
//Persoana.java
public class Persoana{
   private byte varsta;
   private String nume;
   public Persoana(){
      this(“”,0);
   }
   public Persoana(String nume, byte varsta){
    this.nume=nume;
    this.varsta=varsta;
   }
   public byte getVarsta(){
      return varsta;
   }
   public void setNume(String nume){
    this.nume=nume;
   }
   public boolean maiTanara(Persoana p){//...
   }
}
```

Overloading methods

- A class may contain multiple methods with the same name but with different signature. A signature = return type and the list of the formal parameters

```
public class Complex{
  private double real, imag;
  // constructors ...
  public void aduna (double real){
   this.real+=real;
   }
   public void aduna(Complex c){
        this.real+=c.real;
        this.imag+=c.imag;
   }
   public Complex aduna(Complex cc){
   this.real+=cc.real;
   this.imag+=cc.imag;
   return this;
   }
}
//Errors?
```

- Java does not allow the operators overloading.
- Class String has overloaded operators `+` and `+=`.

```
String s="abc";
     String t="EFG";
     String r=s+t;
     s+=23;
     s+=' ';
     s+=4.5;
//s="abc23 4.5";
//r="abcEFG"
```

- Destructor: In Java there is no any destructor.
  - The garbage collector deallocates the memory .

Static methods

- Are declared using the keyword **`static`**
- **They are shared by all class instances**

```
public class Complex{
    private double real, imag;
    public Complex(double re, double im){
     //...
    }
    public static Complex suma(Complex a, Complex b){
        return new Complex(a.real+b.real, a.imag+b.imag);
    }
//...
}
```

Static methods

- They are called using the class name:

```
Complex a,b;
//... initialization a and b
Complex c=Complex.aduna(a, b);
```

1. A static method cannot use those fields (or call those methods) which are not static. It can use or call only the static members.

Java Primitive Data Types

Variables Declaration:

```
type name_var1[=expr1][, name_var2[=expr2]…];
```

Primitive Data Types

Type	Nr. byte	Values	Default value
boolean	-	true, false	false
byte	1	-128 … +127	(byte)0
short	2	-2^{15} …$2^{15}-1$	(short)0
int	4	-2^{31} …$2^{31}-1$	0
long	8	-2^{63} …$2^{63}-1$	0L
float	4	IEEE754	0.0f
double	8	IEEE754	0.0d
char	2	Unicode 0, Unicode $2^{16}-1$	‘\u0000’ (null)

Passing arguments

- Primitive type arguments (boolean, int, byte, long, double) are **passed by value**. Their values are copied on the stack.
- Arguments of reference type are **passed by value**. A reference to them is copied on the stack, but their content (fields for objects, locations for array) can be modified if the method has the rights to accces them.

1. There is not any way to change the passing mode(like & in C++).

```
class Parametrii{
    static void interschimba(int x, int y){
        int tmp=x;
        x=y;
        y=tmp;
  }
  public static void main(String[] args) {
        int x=2, y=4;
        interschimba(x,y);
        System.out.println("x="+x+" y="+y);   //?
    }
}
```

Passing arguments

```
class B{
    int val;
    public B(int x){
        this.val=x;
    }
    public String toString(){
        return ""+val;
    }
     static void interschimba(B x, B y){
        B tmp=x;
        x=y;
        y=tmp;
        System.out.println("[Interschimba B] x="+x+" y="+y);
    }
    public static void main(String[] args) {
        B bx=new B(2);
        B by=new B(4);
        System.out.println("bx="+bx+" by="+by);
        interschimba(bx,by);
        System.out.println("bx="+bx+" by="+by);   //?
     }
}
```

Passing arguments

```
class B{
    int val;
    public B(int x){
        this.val=x;
    }
    public String toString(){
        return ""+val;
    }
     static void interschimbaData(B x, B y){
        int tmp=x.val;
        x.val=y.val;
        y.val=tmp;
        System.out.println("[InterschimbaData] x="+x+" y="+y);
    }
    public static void main(String[] args) {
        B bx=new B(2);
        B by=new B(4);
        System.out.println("bx="+bx+" by="+by);
        interschimbaData(bx,by);
        System.out.println("bx="+bx+" by="+by);   //?
     }
}
```

Java Arrays

One dimension Array

Array declaration

```
type[] name;
type name[];
```

Array allocation:

```
array_name=new type[dim]; //memory allocation
    //index  0 … dim-1
```

Accessing an array element: array_name`[index]`

Examples:

```
float[] vec;
vec=new float[10];
int[] sir=new int[3];
float tmp[];
tmp=vec;      //vec and tmp refer to the same array
```

One dimension Array

Built-in length: returns the array dimension.

```
int[] sir=new int[5];
int lung_sir=sir.length;  //lung=5;
sir[0]=sir.length;
sir.length=2;     //error

int[] y;
int lung_y=y.length;  //error, y was not created

double[] t=new double[0];
int lung_t=t.length;  //lung_t=0;
t[0]=2.3       //error: index out of bounds
```

the shortcut syntax to create and initialize an array:

```
int[] dx={-1,0, 1};
```

Rectangular multidimensional array

Declaration:

```
type name[][][]…[];
type[][][]…[] name;
```

Creation:

```
name=new type[dim1][dim2]…[dimn];
```

Accessing an element:

```
name[index1][index2]…[indexn];
```

Examples:

```
int[][] a;
a=new int[5][5];
a[0][0]=2;
int x=a[2][2];
```

Non-Rectangular Multidimensional Array

Examples:

```
int[][] a=new int[3][];
for(int i=0;i<3;i++)
    a[i]=new int[i+1];
int x=a.length;      //x=?
int y=a[2].length;   //y=?
```

Declaration+creation+initialization:

```
char[][] lit={{'a'},{'b'}};
int[][] b={{1,2},
           {2,5,8},
            {1}};
double[][] mat=new double[][]{{1.3, 0.5}, {2.3, 4.5}};
```

Array of objects

- Each array element must be allocated and intialized.

```
public class TablouriObiecte {
   static void genereaza(int nrElem, Complex[] t){
        t=new Complex[nrElem];
        for(int i=0;i<nrElem;i++)
            t[i]=new Complex(i,i);
   }
   static Complex[] genereaza(int nrElem){
        Complex[] t=new Complex[nrElem];
        for(int i=0;i<nrElem;i++)
            t[i]=new Complex(i,i);
        return t;
   }
   static void modifica(Complex[] t){
        for(int i=0;i<t.length;i++)
            t[i].suma(t[i]);
   }
//...
```

Array of objects

```
  static Complex suma(Complex[] t){
        Complex suma=new Complex(0,0);
        for(int i=0; i<t.length;i++)
            suma.aduna(t[i]);
        return suma;
    }
   public static void main(String[] args) {
        Complex[] t=genereaza(3);
        Complex cs=suma(t);
        System.out.println("suma "+cs);
        Complex[] t1=null;
        genereaza(3,t1);
        Complex cs1=suma(t1);
        System.out.println("suma "+cs1);
        modifica(t);
        System.out.println("suma dupa modificare "+suma(t));

    }
}
```

Java Char and String

```
char[] sir={'a','n','a'}; //comparison and printing
                          //is done character by character
sir[0]='A';
```

A constant Sequence of Chars :

```
"Ana are mere";           //object of type String
```

String class is immutable:

```
String s="abc";
s=s+"123";          //concatenating strings
String t=s;//t="abc123"
t+="12";            //t=?, s=?
```

String content can not be changed: `t[0]='A';`

```
char c=t.charAt(0);
```

```
method length(): int lun=s.length();
t.equals(s)
/*Returns true if and only if the argument is a String object that
represents the same sequence of characters as this object. */
compareTo(): int rez=t.compareTo(s)
/*Compares two strings lexicographically. Returns an integer indicating
whether this string is greater than (result is > 0), equal to (result is =
0), or less than (result is < 0) the argument.*/
```

Java code reusing: composition and inheritance

Code reusing

- *Composing*: The new class consists of instance objects of the existing classes

- *Inheritance*: A new class is created by extending an existing class (new fields and methods are added to the fields and methods of the existing class)

Composing

The new class contains fields which are instance objects of the existing classes.

```
class Adresa{
 private String nr, strada, localitate, tara;
 private long codPostal;
//...
}
```

```
class Persoana{
  private String nume;
  private Adresa adresa;
  private String cnp;
//...
}
```

```
class Scrisoare{
  private String destinatar;
  private Adresa adresaDestinatar;
  private String expeditor;
  private Adresa adresaExpeditor
//...
}
```

Inheritance

- Using the keyword **`extends:`**

```
class NewClass extends ExistingClass{
    //...
}
```

- **`NewClass`** is called subclass,or child class or derived class.
- **`ExistingClass`** is called superclass, or parent class, or base class.
- Using inheritance, **`NewClass`** will have all the members of the class **`ExistingClass`**. However, **`NewClass`** may either redefine some of the methods of the class **`ExistingClass`** or add new members and methods.

Inheritance

```
public class Punct{
  private int x,y;
  public Punct(int x, int y){
     this.x=x;
     this.y=y;
  }
  public void muta(int dx, int dy){
  //...
  }
  public void deseneaza(){
    //...
    }
}
public class PunctColorat extends Punct{
    private String culoare;
    public PunctColorat(int x, int y, String culoare){...}
    public void deseneaza(){...}
    public String culoare(){...}
}
```

Inheritance

- Notions
  - `deseneaza` is an overridden method.
  - `muta` is an inherited method.
  - `culoare` is a new added method.
- Heap memory:

```
Punct punct =new Punct(2,3);
PunctColorat punctColorat=new PunctColorat(2,3,"alb");
```

Method overriding

- A derived class may override methods of the base class
- Rules:

1. The class `B` overrides the method `mR` of the class `A` if `mR` is defined in the class B with the same signature as in the class `A`.
2. For a call `a.mR()`, where `a` is an object of type `A`, it is selected the method `mR` which correspond to the object referred by `a`.

```
A a=new A();
a.mR();     //method mR from A
a=new B();
a.mR();     //method mR from B
```

3. The methods which are not overridden are called based on the receiver type.

Method overriding

4. annotation `@Override` in order to force a compile-time verification

```
public class A{
  public void mR(){
    //...
  }
}
```

```
public class B extends A{
  @Override
  public void mR(){
  }
}
```

4. The return type of an overriding method may be a subtype of the return type of the overridden method from the base class (*covariant return type*).

```
public class C{
  public SuperA m(){...}
}
public class D extends C{
  public SubB m(){...}
}
```

Method overloading

- A subclass may overload a method from the base class.
- An instance object of a subclass may call all the overloaded methods including those from the superclass.

```
public class A{
  public void f(int h){ //...
  }
  public void f(int i, char c){ //...
  }
}
```

```
public class B extends A{
  public void f(String s, int i){
    //...
  }
}
```

```
B b=new B();
b.f(23);
b.f(2, 'c');
b.f("mere",5);
```

Calling the superclass constructors

- A constructor of a subclass can call a constructor of the base class.
- It is used the keyword **super**.
- The call of the base class must be the first instruction of the subclass constructor.

```
public class Persoana{
  private String nume;
  private int varsta;
  public Persoana(String nume, int varsta){
    this.nume=nume;
    this.varsta=varsta;
  }
  //...
}
public class Angajat extends Persoana{
  private String departament;
  public Angajat(String nume, int varsta, String departament){
    super(nume, varsta);
    this.departament=departament;
  }
  //...
}
```

The keyword super

- It is used in the followings:
  - To call a constructor of the base class.
  - To refer to a member of the base class which has been redefined in the subclass.

```
public class A{
  protected int ac=3;
     //...
}
```

```
public class B extends A{
  protected int ac=3;
  public void f(){
    ac+=2;  super.ac--;
  }
}
```

  - To call the overridden method (from the base class) from the overriding method (from the subclass).

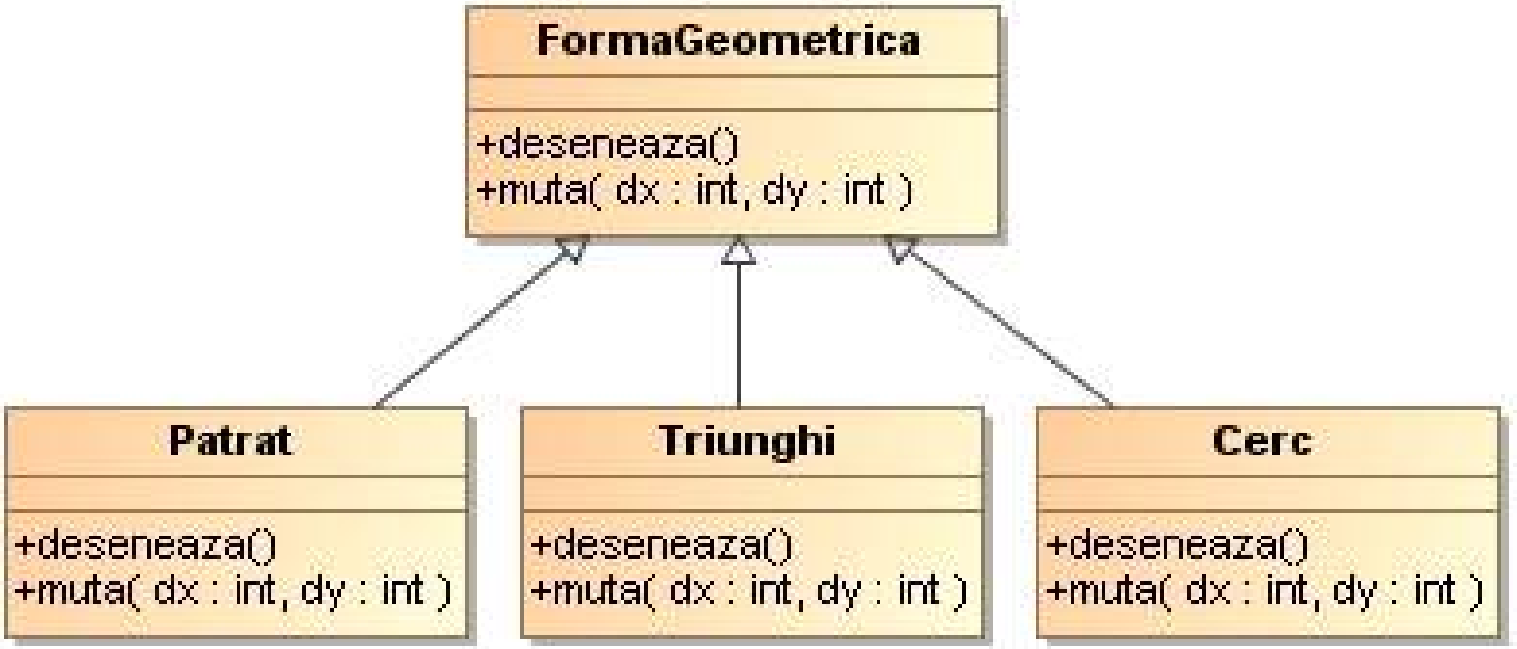
```
public class Punct{
  //...
  public void deseneaza(){
    //...
  }
}
```

```
public class PunctColorat extends Punct{
  private String culoare;
  public void deseneaza(){
    System.out.println(culoare);
    super.deseneaza();
  }
}
```

Java polymorphism: abstract classes, interfaces

Polymorphism

- The ability of an object to have different behaviors according to the context.
- 3 types of polymorphism:
  - ad-hoc: method overloading.
  - Parametric: generics types.
  - inclusion: inheritance.



Polymorphism

- *early binding*: the method to be executed is decided at compile time
- *late binding*: the method to be executed is decided at execution time
- Java uses late binding to call the methods. However there is an exception for static methods and final methods.

```
void deseneaza(FormaGeometrica fg){
   fg.deseneaza();
}

//...
FormaGeometrica fg=new Patrat();
deseneaza(fg);   //call deseneaza from Patrat
fg=new Cerc();
deseneaza(fg);   //call deseneaza from Cerc
```

Polymorphic collections

```
public FiguraGeometrica[] genereaza(int dim){
   FiguraGeometrica[] fg=new FiguraGeometrica[dim];
     Random rand = new Random(47);
     for(int i=0;i<dim;i++){
      switch(rand.nextInt(3)) {
    case 0: fg[i]= new Cerc(); break;
    case 1: fg[i]= new Patrat(); break;
    case 2: fg[i]= new Triunghi(); break;
              default:
     }
   }
    return fg;
}

public void muta(FiguraGeometrica[] fg){
  for(FiguraGeometrica f: fg)
     f.muta(3,3);
}
```

Abstract classes

- An abstract method is declared but not defined. It is declared with the keyword **`abstract`**.

```
[modificator_acces] abstract ReturnType nume([list_param_formal]);
```

- *An abstract class may contain abstract methods*.

```
[public] abstract class ClassName {
     [fields]
     [abstract methods declaration]
     [methods declaration and implementation]
}

public abstract class Polinom{
  //...
   public abstract void aduna(Polinom p);
}
```

Abstract classes

1. An abstract class cannot be instantiated.

```
Polinom p=new Polinom();
```

2. If a class contains at least one abstract method then that class must be abstract.
3. A class can be declared abstract without having any abstract method.
4. If a class extends an abstract class and does not define all the abstract methods then that class must also be declared abstract.

```
public abstract class A{
  public A(){}
  public abstract void f();
  public abstract void g(int i);

}
```

```
public abstract class B extends A{
  private int i=0;
  public void g(int i){
   this.i+=i;
  }
}
```

Java interfaces

- Are declared using keyword `interface.`

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

1. Only method declaration, no method implementation
2. No constructors
3. All declared methods are implicitly public.
4. It may not contain any method declaration.
5. It may contain fields which by default are `public`, `static` and constant (`final`).

```
public interface LuniAn{
 int IANUARIE=1, FEBRUARIE=2, MARTIE=3, APRILIE=4, MAI=5,
    IUNIE=6, IULIE=7, AUGUST=8, SEPTEMBRIE=9, OCTOMBRIE=10, NOIEMBRIE=11,
      DECEMBRIE=12;
}
```

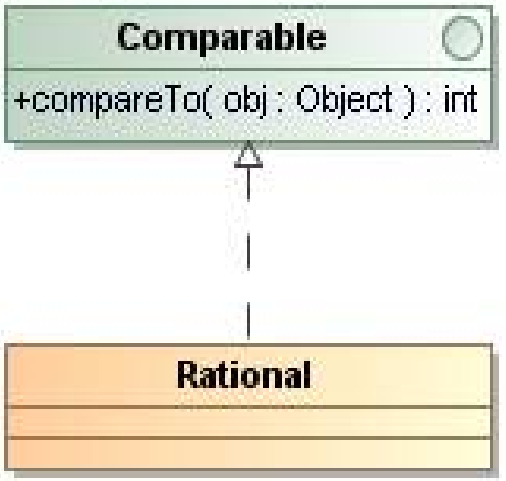
Interface implementation

- A class can implement an interface, using `implements`.

```
[public] class ClassName implements InterfaceName{
    [interface method declarations]
    //other definitions
}
```

1. The class must implement all the interface methods

```
public interface Comparable{
     int compareTo(Object o);
}
public class Rational implements Comparable{
  private int numarator, numitor;
  //...
  public int compareTo(Object o){
  //...
  }
}
```



Extending an interface

- An interface can inherit one or more interfaces

```
[public] interface InterfaceName extends Interface1[, Interface2[, ...]
    ]{
   [declaration of new methods]

}
```

1. Multiple inheritance.

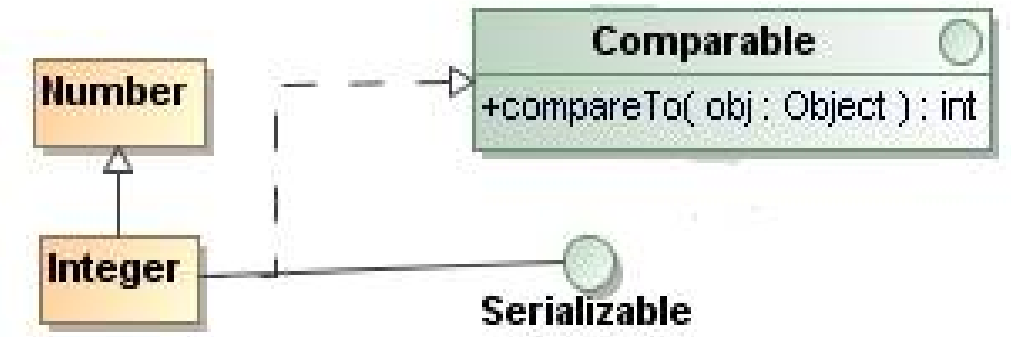
```
public interface A{
  int f();
}
public interface B{
  double h(int i);
}
public interface C extends A, B{
  boolean g(String s);
}
```

Inheritance and interfaces

- A class can inherit one class but can implement multiple interfaces

```
[public] class NumeClasa extends SuperClasa implements Interfata1,
     Interfata2, ..., Interfatan{
  //...
}
```

```
Example:
 public class Integer extends Number implements Serializable, Comparable{
     //...
     public int compareTo(Object o){
     //...
     }
}
```



Variables of type interface

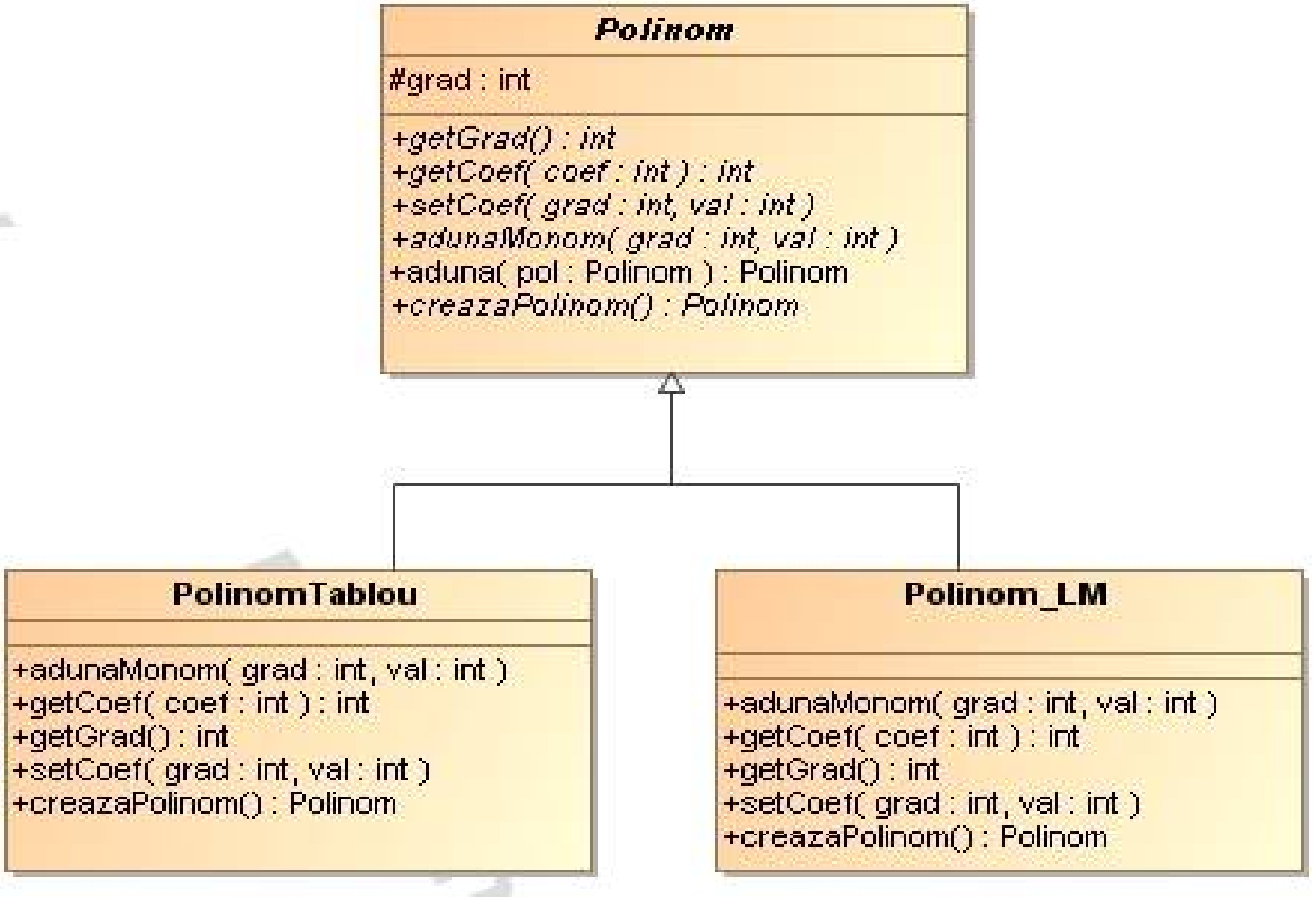
- An interface is a referrence type
- It is possible to declare variables of type interface. These variables can be initialized with objects instances of classes which implement that interface. Through those variables only interface methods can be called

```
public interface Comparable{
  //...
}
public class Rational implements Comparable{
  //...
}
Rational r=new Rational();
Comparable c=r;
Comparable cr=new Rational(2,3);
cr.compareTo(c);
c.aduna(cr); //ERROR!!
```

Abstract Class vs Interface

Public, protected, private methods	only public methods.
Have fields	Can have only static and final fields
Have constructors	No constructors.
It is possible to have no any abstract method.	It is possible to have no any methods
Both do not have instance objects	

Abstract classes vs Interfaces



Abstract Classes vs Interfaces