Advanced Programming Methods
Lecture 2

Code reusing

Creation of new classes using the existing classes

- *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;
//...
}
```

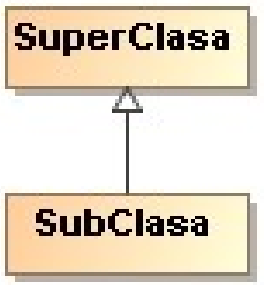
```
class Scrisoare{
  private String destinatar;
  private Adresa adresaDestinatar;
  private String expeditor;
  private Adresa adresaExpeditor
//...
}
```

Inheritance

- Using the keyword `extends:`

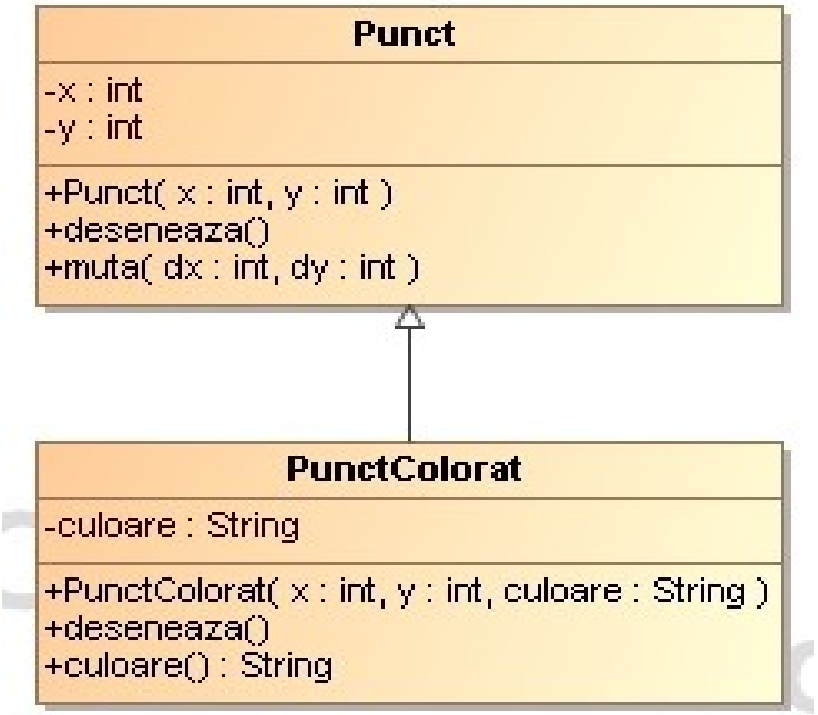
```
class NewClass extends ExistingClass{
    //...
}
```

- `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 to redefine some of the methods of the class `ExistingClass` or to add new members and methods.
- UML notation:



Inheritance

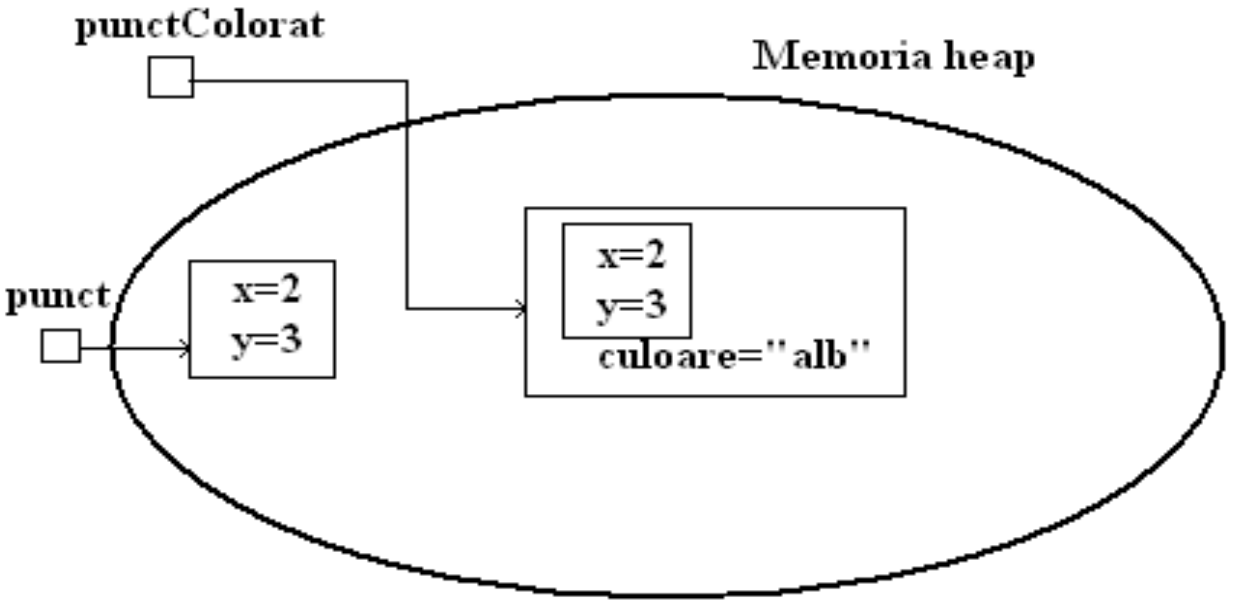
```
public class Punct{
  private int x,y;
  public Punct(int x, int y){
     this.x=x;
     this.y=y;
  }
  public void muta(int dx, int dy){
  //...
  }
  public void deseneaza(){
     //...
     }
}
public class PunctColorat extends Punct{
     private String culoare;
     public PunctColorat(int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

Protected

- The fields and methods which are declared `protected` are visible inside the class, inside the derived classes and inside the same package.

```
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){
    this.nume=nume;
    this.varsta=varsta;
    this.departament=departament;
  }
  //...
}
```

Protected

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

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;
  }
  //...
}
```

Fields initialization

- The initialization order:
  1. Static fields.
  2. Non-static fields which are initialized at the declaration site.
  3. The other fields are intialized with their types default values.
  4. The constructor is executed

Fields initialization

```
public class Produs{
  static int contor=0;        //(1)
  private String denumire;    //(2)
  private int id=contor++;    //(3)
  public Produs(String denumire){ //(4)
    this.denumire=denumire;
  }
  public Produs(){            //(5)
    denumire="";
  }
  //...
}

Produs prod=new Produs();         // (1), (3), (2), (5)

Produs prod2=new Produs("mere"); //?
```

Fields initialization and inheritance

- First the base class fields are initialized and then those of the derived class
- In order to initialize the base class fields the default base class constructor is called by default. If the base class does not have a default constructor, each constructor of the derived class must call explicitly one of the base class constructors.

```
public class Punct{
  private int x,y;
  public Punct(int x, int y){
     this.x=x;
     this.y=y;
  }
}
public class PunctColorat extends Punct{
  private String culoare;
  public PunctColorat(int x, int y, String culoare){
    super(x,y);
    this.culoare=culoare;
  }
}
```

Fields initialization and inheritance

```
public class Singleton{
  private static Singleton instance;
  private Singleton(){
   //...
  }
  public static Singleton getInstance(){
    if (instance==null)
   instance=new Singleton();
    return instance;
  }
  //...
}
public class SubClasa extends Singleton{
  public SubClasa(){
    //...
  }
  //...
}
//?
```

The keyword super

- It is used in the followings:
  - To call a constructor of the base class.
  - To refer to a meber 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--;
  }
}
```

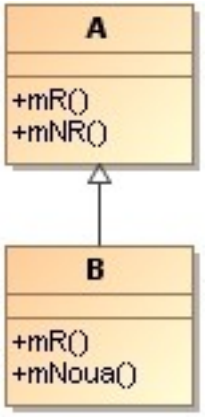
  - To call the overridden method (from the base class) from the overriding method (from the subclass).

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

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

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 variable type.

Method overriding

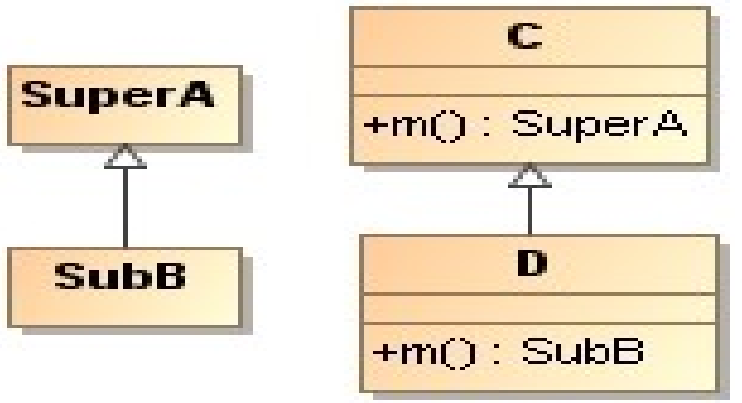
4. adnotation `@Override` (JSE >=5) in order to force a compile-time verification

```
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*). (JSE>=5).

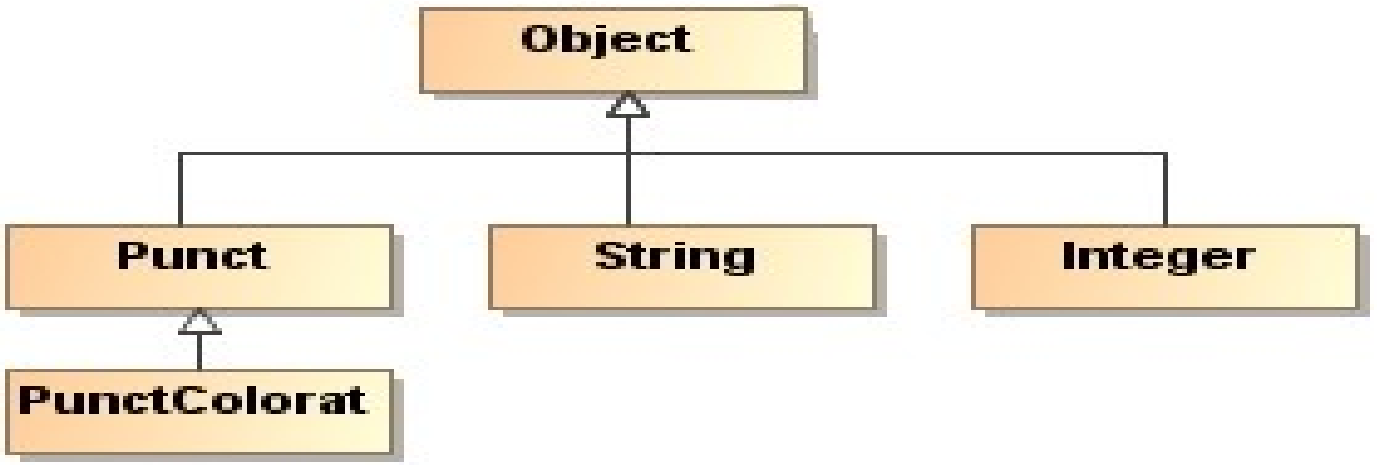
```
public class C{
  public SuperA m(){...}
}
public class D extends C{
  public SubB m(){...}
}
```



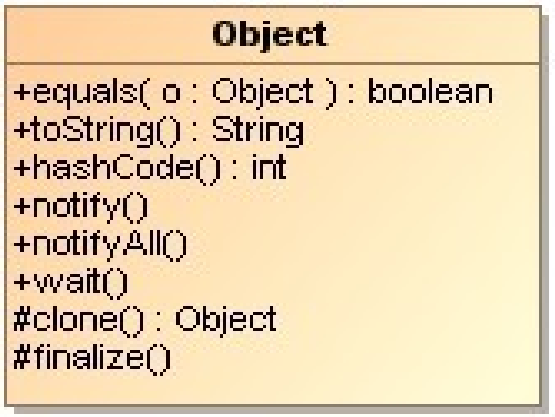
The class Object

- It is the top of the java classes hierarchy.
- By default Object is the parent of a class if other parent is not explicitly defined

```
public class Punct{
  //...
}
public class PunctColorat extends Punct{
  //...
}
```



Class Object - methods



- `toString`() is called when a String is expected
- `equals`() is used to check the equality of 2 objects. By default it compares the references of those 2 objects.

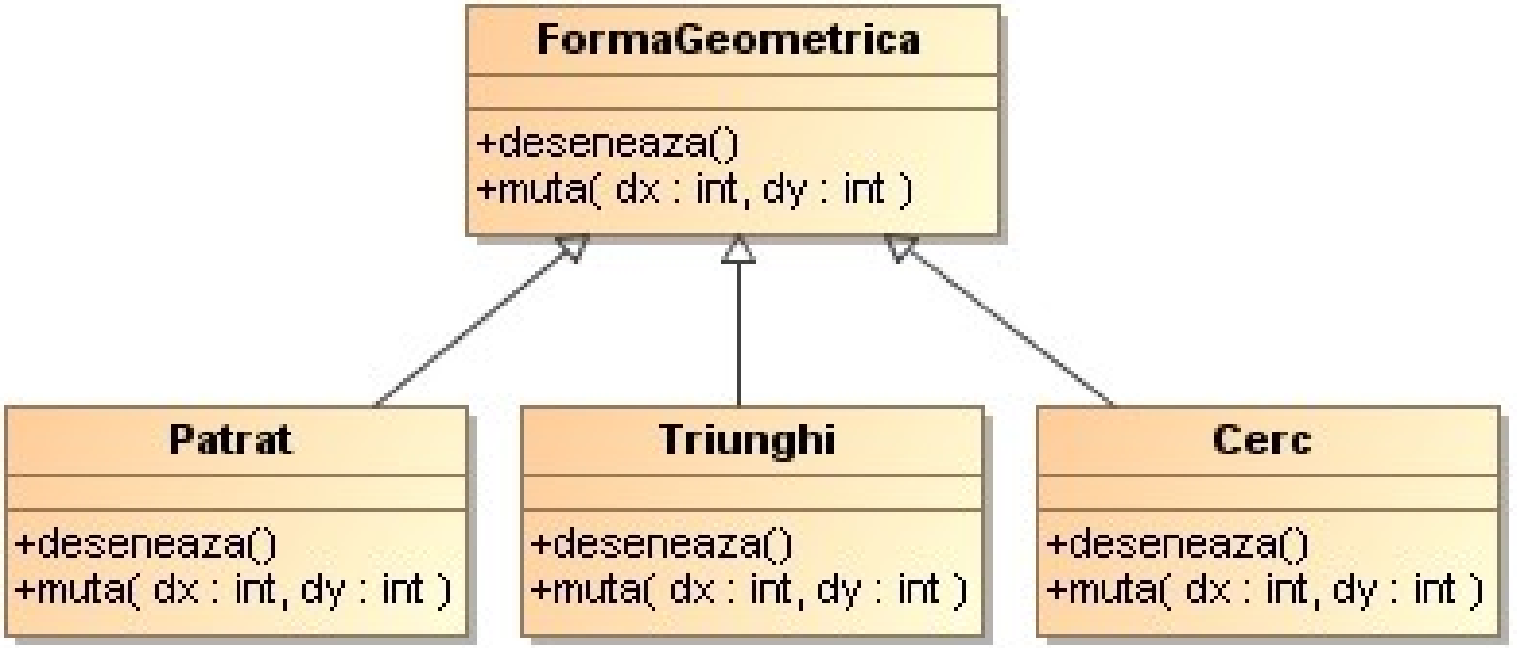
```
Punct p1=new Punct(2,3);
Punct p2=new Punct(2,3);
boolean egale=(p1==p2);   //false;
egale=p1.equals(p2);     //true, Punct must redefine equals
System.out.println(p1);   //toString is called
```

Class Object - methods

```
public class Punct {
    private int x,y;
    public Punct(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object obj) {
        if (! (obj instanceof Punct))
            return false;
        Punct p=(Punct)obj;
        return (x==p.x)&& (y==p.y);
    }

    @Override
    public String toString() {
        return ""+x+' '+y;
    }
    //...
}
```

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 delared with the keyword **`abstract`**.

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

- *An abstract class may contain abstract methods*.
- An abstract class is defined using **`abstract.`**

```
[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;
}
```

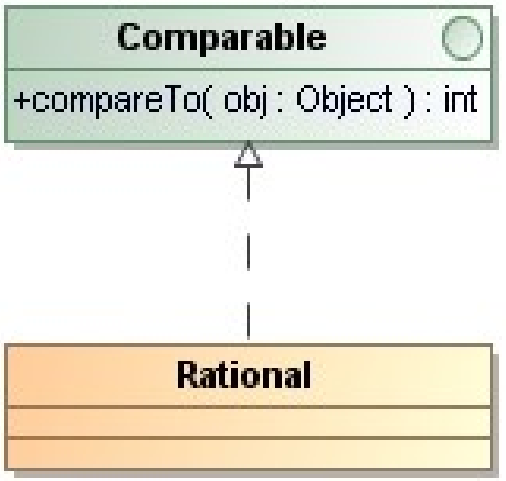
Interface implementation

- 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
2. If at least one method is not defined then the class must be declared

```
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);
}
```

Collisions

```
interface I1 {
  void f();
}
interface I2 {
  int f(int i);
 }
interface I3 {
  int f();
}
class C {
  public int f() {
    return 1;
  }
}
interface I6 extends I1, I2{}
interface I4 extends I1, I3 {}  //error
```

Implementing multiple interfaces

- A class can implement multiple interfaces.

```
[public] class ClassName implements Interface1, Interface2, ...,
     Interfacen{
  //...
}
```

- The class must implement the methods from all interfaces.It may occur collisions between methods declared in different interfaces

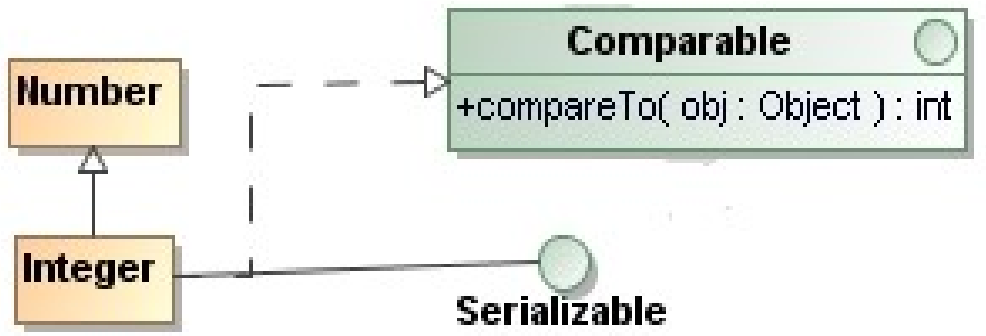
```
class C2 implements I1, I2 {
     public void f() {}
     public int f(int i) { return 1; }
     //overloading
}
class CC implements I1, I3{  //error at compile-time
  //...
}
```

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){
     //...
     }
}
```



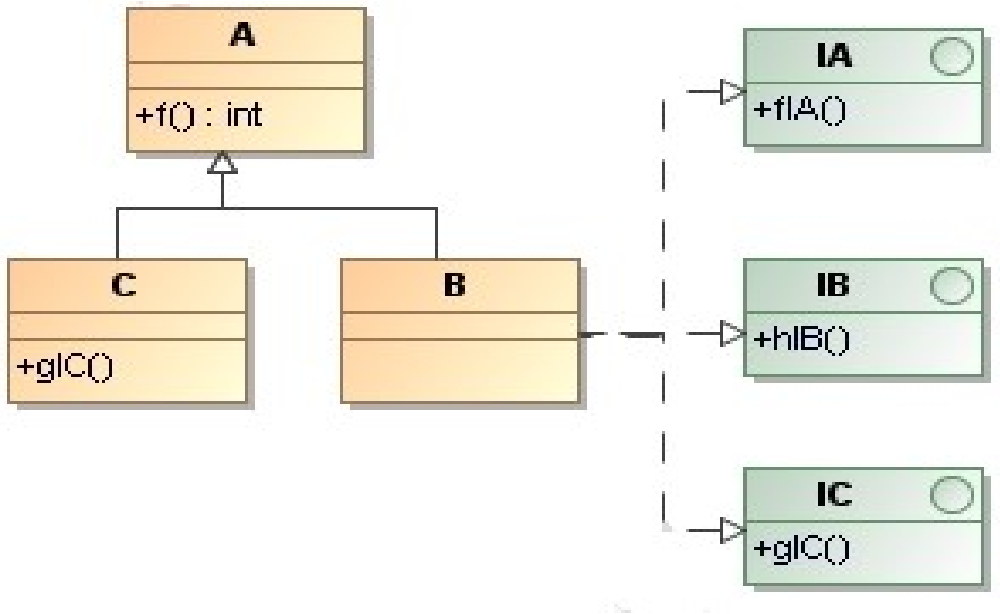
Variables of type interface

- 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);
```

Variable of type interface

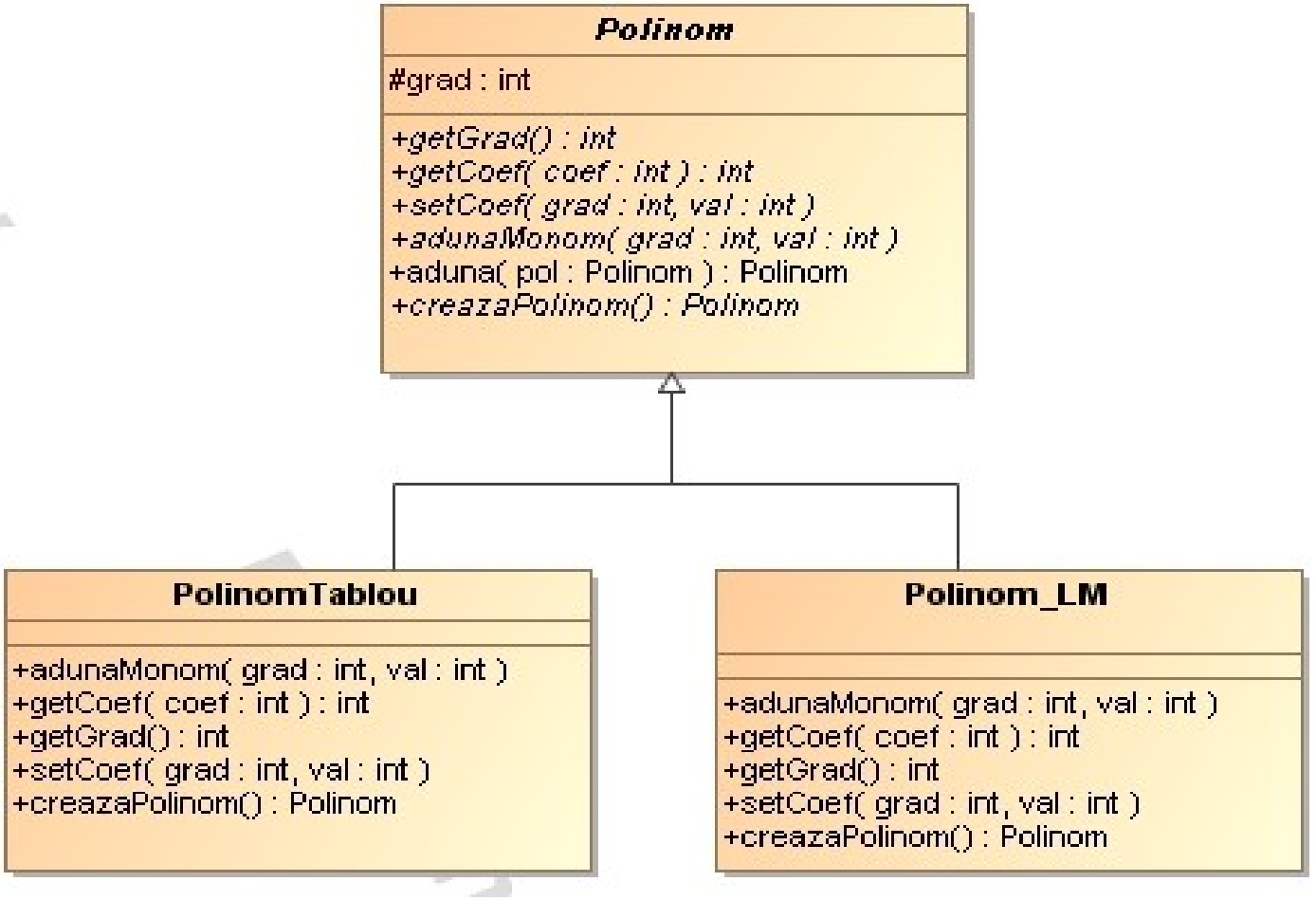
```
B b=new B();
IA ia=b; ia.fIA();

IB ib=b; ib.hIB();
IC ic=b; ic.gIC();

ic.f(); //?

C c=new C();
IC ic=c;
ic.gIC();

//?
```



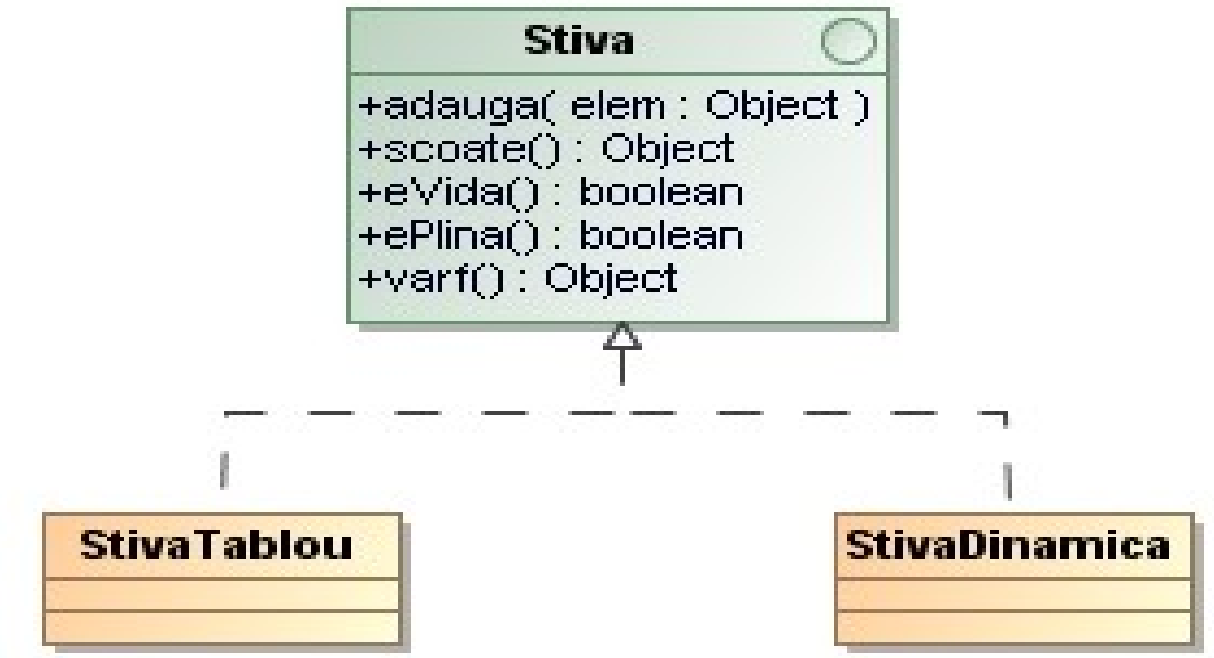
Abstract Class vs Interface

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 class.	It is possible to have no any methods
Both do not have instance objects	

Abstract classes vs Interfaces

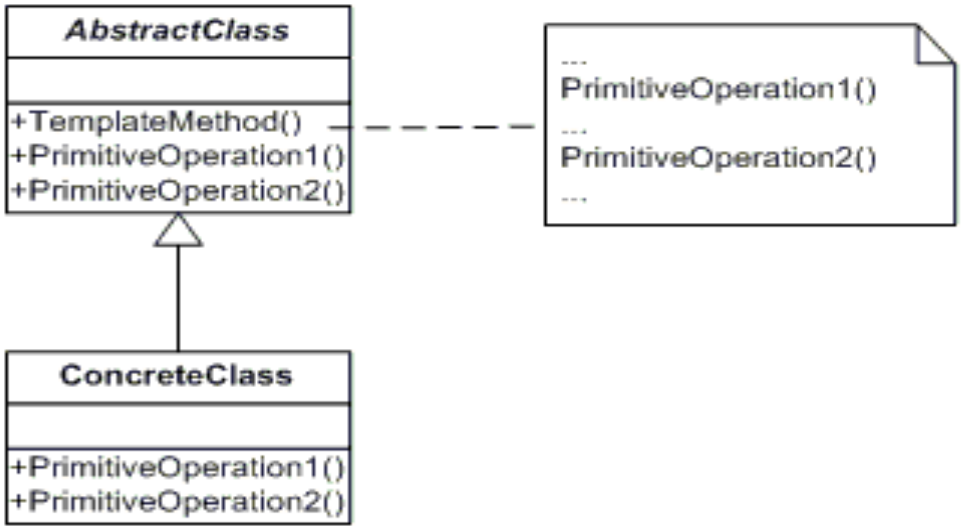


Abstract Classes vs Interfaces



Template Method Pattern

Defines the general structure of an algorithm, but it does not define all the steps. The subclasses may redefine some steps without modifying the general structure of the algorithm.



Template Method Pattern

```
abstract class Polinom{
//...
public Polinom aduna(Polinom pol){
  Polinom rez=creazaPolinom();
  for(int i=0;i<getGrad();i++)
     rez.adunaMonom(i, getCoef(i));
  for(int i=0;i<pol.getGrad();i++)
     rez.adunaMonom(i, pol.getCoef(i));
  return rez;
}
public abstract int getCoef(int gr);
//...
}
```