Advanced Programming Methods

Lecture 1 – Java Basics

Course Overview

Object-oriented languages:

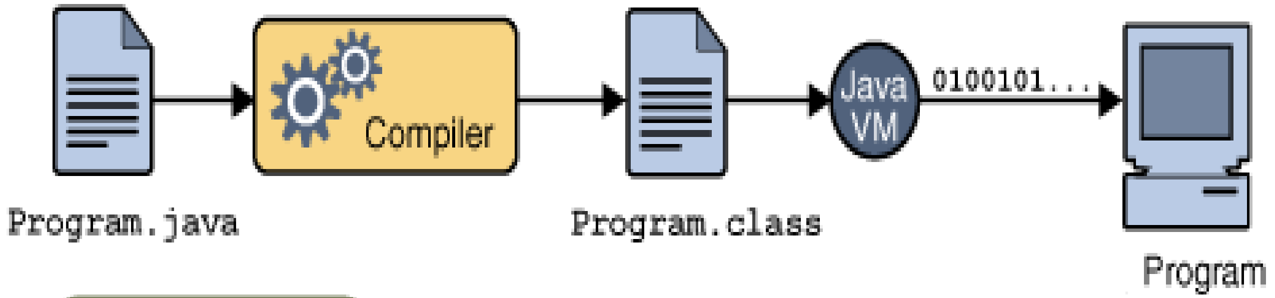
- Java (the first 10 lectures): Basics, Collections, IO, Functional Programming, Concurrency, XML, GUI, Metaprogramming

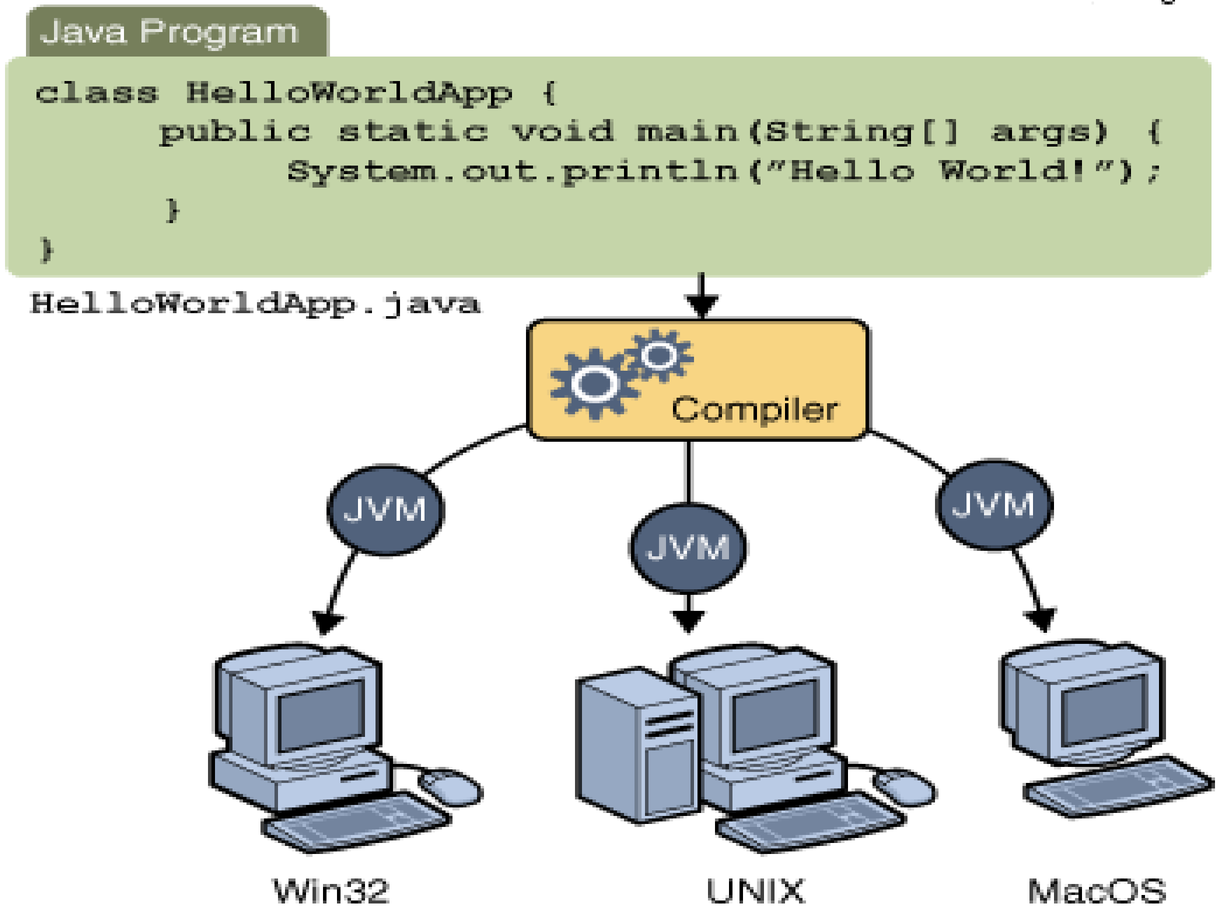
- C# (the last 3 lectures): Basics, Collections, IO

Java References

- Bruce Eckel, *Thinking in Java*
- The Java Tutorials, 2017. **http://docs.oracle.com/javase/tutorial/index.html**
- Java 9 API, 2017. **https://docs.oracle.com/javase/9/docs/api/overview-summary.html**

Java Technology





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

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)

Primitive Data Types

Examples:

```
boolean gasit=true;
int numar=34, suma;
float eroare=0.45;
char litera=‘c’;
litera=‘f’;
litera=litera+1;
```

Java Comments

1. `// entire line`

2.
   ```
   /* multiple
     lines */
   ```

3.
   ```
   /** used by documentation Javadoc tool
    multiple
    lines*/
   ```

Obs: Comments canot be nested into strings.

```
/* … /*
*/ … */   NOT OK!
```

```
/* …
   //
   //
*/   OK!!
```

Java Constants

```
final type name [=value];
```

Examples:

a)
```
final int MAX=100;
```

b)
```
final int MAX;
…
MAX=100;
…
MAX=150; //error
```

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:

$\texttt{name=new type[dim}_1\texttt{][dim}_2\texttt{]…[dim}_n\texttt{];}$

Accessing an element:

$\texttt{name[index}_1\texttt{][index}_2\texttt{]…[index}_n\texttt{];}$

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}};
```

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.*/
```

Operators

arihtmetic: `+, -, *, /, %`

relational: `>, >=, <, <=, !=, ==`

increment/decrement: `++, --`

```
prefix:  int a=1;
     int b=++a; //a=2, b=2
postfix: int a=2;
     int b=a++; //a=3, b=2
```

assignment: `=, +=, -=, *=, /=`

conditional: `&&, ||, !`

`bitwise`: - shift `>>, <<, >>>,`

- conditional `&, |, ~ (not), ^ (exclusive or)`

ternary operator: `?:`

```
ex: logical_expr ? expr_1 : expr_2
 If logical_expr is TRUE then expr1 else expr2
```

Operators	Precedence (higher precedence on top, the same precedence on the same line)
1. Postfix	`expr++ expr--`
2. Unari	`++expr --expr +expr -expr ~ !`
3.	`* / %`
4.	`+ -`
5.	`<< >> >>>`
6.	`< > <= >= instanceof`
7.	`== !=`
8.	`&`
9.	`^`
10.	`|`
11.	`&&`
12.	`||`
13.	`? :`

Statements

Sequential Composition:

```
{
     instr1;
     instr2; …
}
```

Conditional:

```
if (logica_expr)
    instr;

if (logical_expr)
    instr1;
else
    instr2;
```

Obs: `logical_expr` is evaluated to `true or false`. Numerical values are not allowed.

Loop Statements

While statement:

```
while(logical_expr)
    instr
```

do-while statement:

```
do
    instr
while(logical_expr);
```

Obs: `Instr` is executed as long as `logical_expr is true.`

Loop Statement

FOR statement:

```
for(initialization;termination; step)
    instr
```

Obs: none of the `initialization, termination, step` are mandatory

```
int suma=0;
for(int i=1;i<10;i++)
   suma+=i;

for(int i=0,suma=0;i<10;i++)
        suma+=i;

    for(;;)
        // instruction
```

Enhanced FOR (EACH) statement

Syntax(JSE >=5):

```
 for(Type elemName : tableName)
    instr;

int[] x={1, 4, 6, 9, 10};
for(int el:x)
  System.out.println(el);

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

Obs: Table elements cannot be modified by using enhanced for statement

```
  int[] x={1,4,6,10};
     for(int el:x){
        System.out.print(" "+el);
        el+=2;
     }
 //1 4 6 10
     for(int e:x){
        System.out.print(" "+e);        //?
     }
```

Return statement:

```
return;
return value;
```

Break statement: terminates the execution of a loop

```
int[] x= { 2, 8, 3, 5, 12, 8, 62};
 int elem = 8;
boolean gasit = false;
 for (int i = 0; i < x.length; i++) {
            if (x[i] == elem) {
                gasit = true;
                break;
            }
    }
```

Continue statement

- skips the current iteration of a loop statement
- stops the execution of the loop instructions and forces the re-evaluation of the loop termination condition

```
int[] x= { 2, 8, 3, 5, 12, 8, 62};
int elem = 8;
int nrApar=0;
 for (int i = 0; i < x.length; i++) {
        if (x[i] != elem)
          continue;
        nrApar++;
}
```

Switch statement

```
switch(integral-selector) {
    case integral-value1 : statement; [break;]
    case integral-value2 : statement; [break;]
    case integral-value3 : statement; [break;]
    case integral-value4 : statement; [break;]
    case integral-value5 : statement; [break;]
    // ...
    default: statement;
}
```

Switch example

```
switch (luna) {
           case 1:
           case 3:
           case 5:
           case 7:
           case 8:
           case 10:
           case 12: nrZile = 31; break;
           case 4:
           case 6:
           case 9:
           case 11: nrZile = 30; break;
           case 2: if ( anBisect(an) )
                            nrZile = 29;
                else
                            nrZile = 28;
                break;
           default:
               System.out.println("Luna invalida");
       }
```

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.`

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;
  }
  //...
}
```

Calling another constructor

1. The call of another constructor must be the first instruction in the caller constructor.
2. The callee constructor cannot be called twice.
3. It is not possible to call two different constructors.
4. A constructor cannot be called from a method.

```
    //Punct.java
  public class Punct{
     private int x, y;

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

      public Punct(int x, int y){
           this.x=x;
           this.y=y;
      }

     public void muta(int dx, int dy){
           this(x+dx, y+dy);
      }
  }
//Erorrs?
```

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;

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;
   }
}
//Erorrs?
```

- 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 .

Objects as Parameters

- Objects can be formal parameters for the methods
- A method can return an object or an array of objects.

```
public class Rational{
private int numarator, numitor;
//Constructors ...

public void aduna(Rational r){
//...
}
public Rational scadere(Rational r){
//...
}

}
```

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);   //?
     }
}
```

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));

    }
}
```

The methods toString and equals

```
public class Complex{
    private double real, imag;
    public Complex(double re, double im){
      //...
    }
    public String toString(){
        if (imag>=0)
            return "("+real+"+"+imag+"i)";
        else
            return "("+real+imag+"i)";
    }

    public boolean equals(Object obj){
        if (obj instanceof Complex){
            Complex c=(Complex)obj;
             return (real==c.real) && (imag==c.imag);
        }
        return false;
    }
//...
}
```

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.

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.

Static fields

- Initialization:
  - At declaration site:

```
public static long MAX=2000;
```

  - In a special intialization block

```
public class Natural {
         public  static long MAX;
 static {
     MAX=2000;
 }
```

If a static field is not intialized, it will take the default value of its type:

```
private static long counter; //0
```

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;
//...
}
```

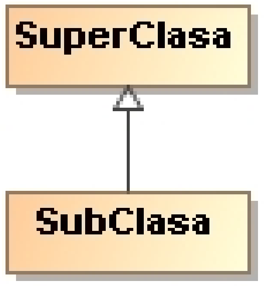
```
class Scrisoare{
  private String destinatar;
  private Adresa adresaDestinatar;
  private String expeditor;
  private Adresa adresaExpeditor
//...
}
```

Inheritance

- Using the keyword `extends:`

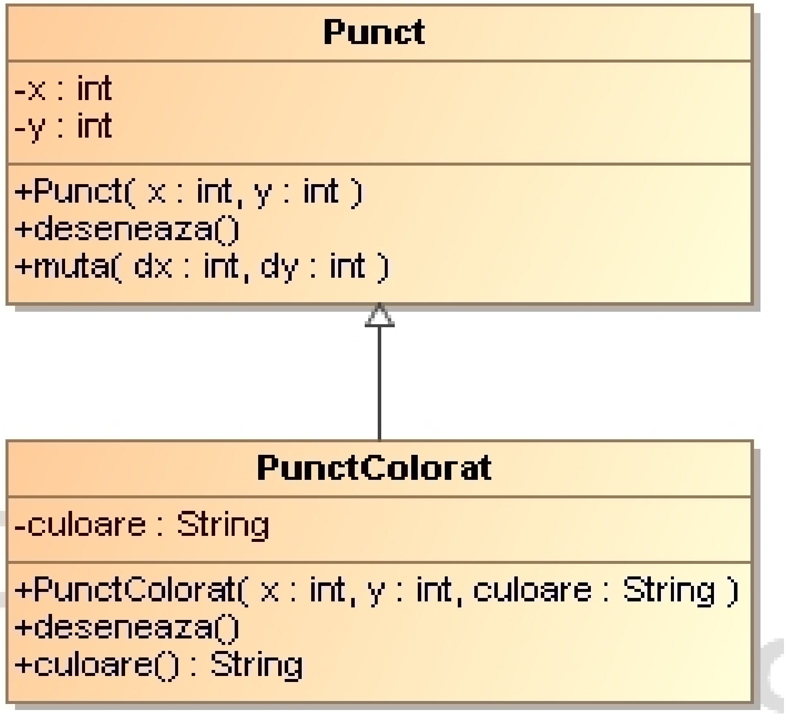
```
class NewClass extends ExistingClass{
    //...
}
```

- `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.
- UML notation:



Inheritance

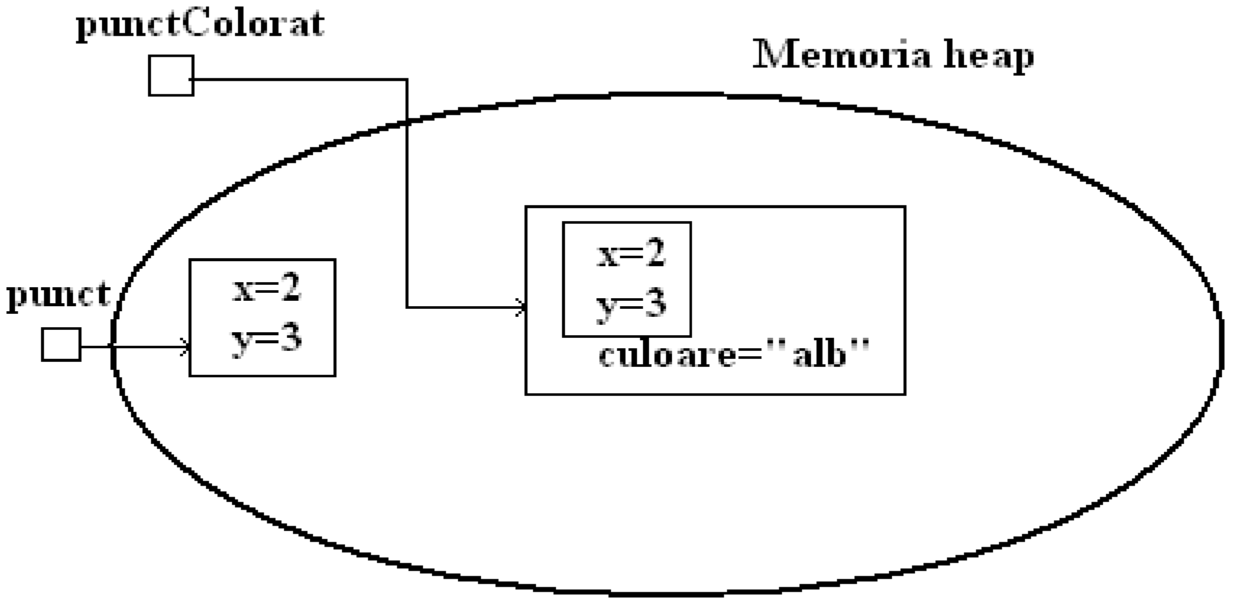
```
public class Punct{
  private int x,y;
  public Punct(int x, int y){
     this.x=x;
     this.y=y;
  }
  public void muta(int dx, int dy){
  //...
  }
  public void deseneaza(){
    //...
    }
}
public class PunctColorat extends Punct{
    private String culoare;
    public PunctColorat(int x, int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


Punct
-x : int
-y : int
+Punct(x : int, y : int)
+deseneaza()
+muta(dx : int, dy : int)
PunctColorat
-culoare : String
+PunctColorat(x : int, y : int, culoare : String)
+deseneaza()
+culoare() : String

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);
```

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;
  }
}
```

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--;
  }
}
```

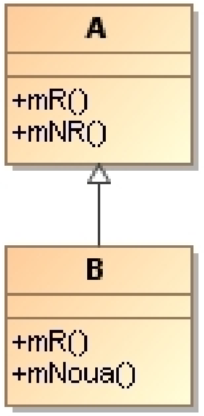
  - To call the overridden method (from the base class) from the overriding method (from the subclass).

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

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

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

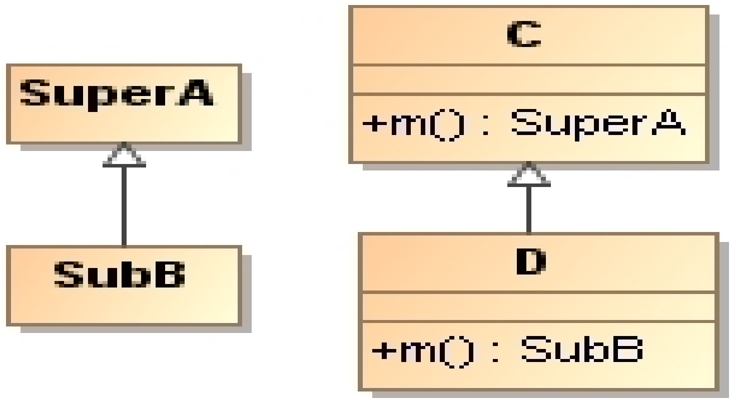
4. adnotation `@Override` (JSE >=5) in order to force a compile-time verification

```
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).

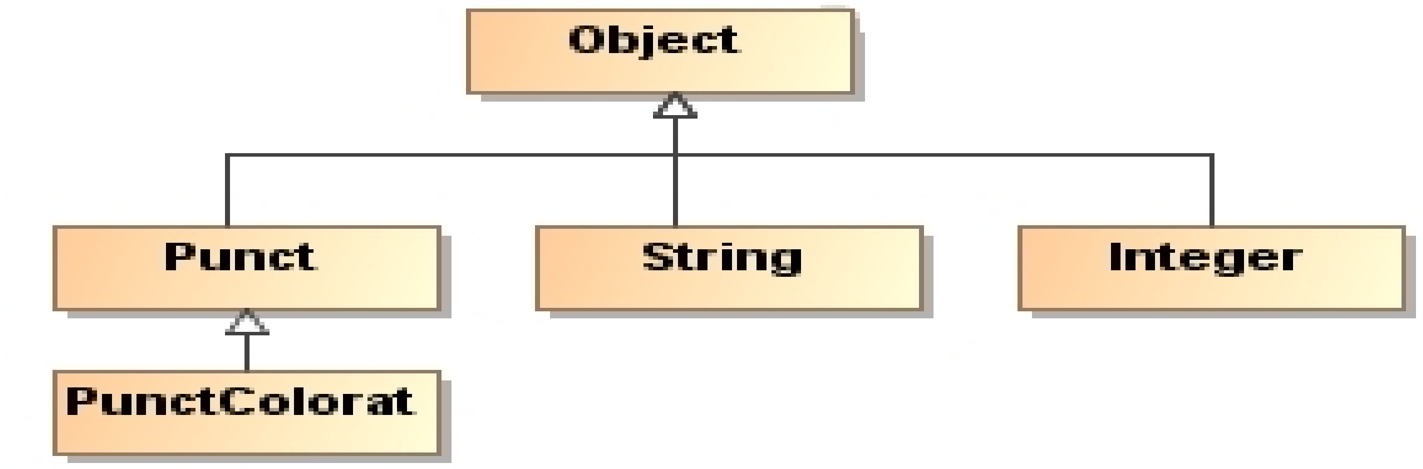
```
public class C{
  public SuperA m(){...}
}
public class D extends C{
  public SubB m(){...}
}
```



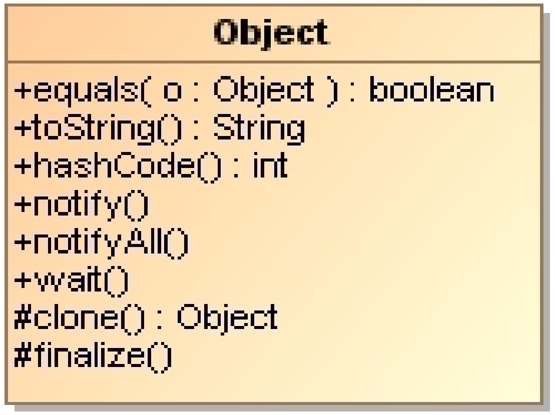
The class Object

- 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.

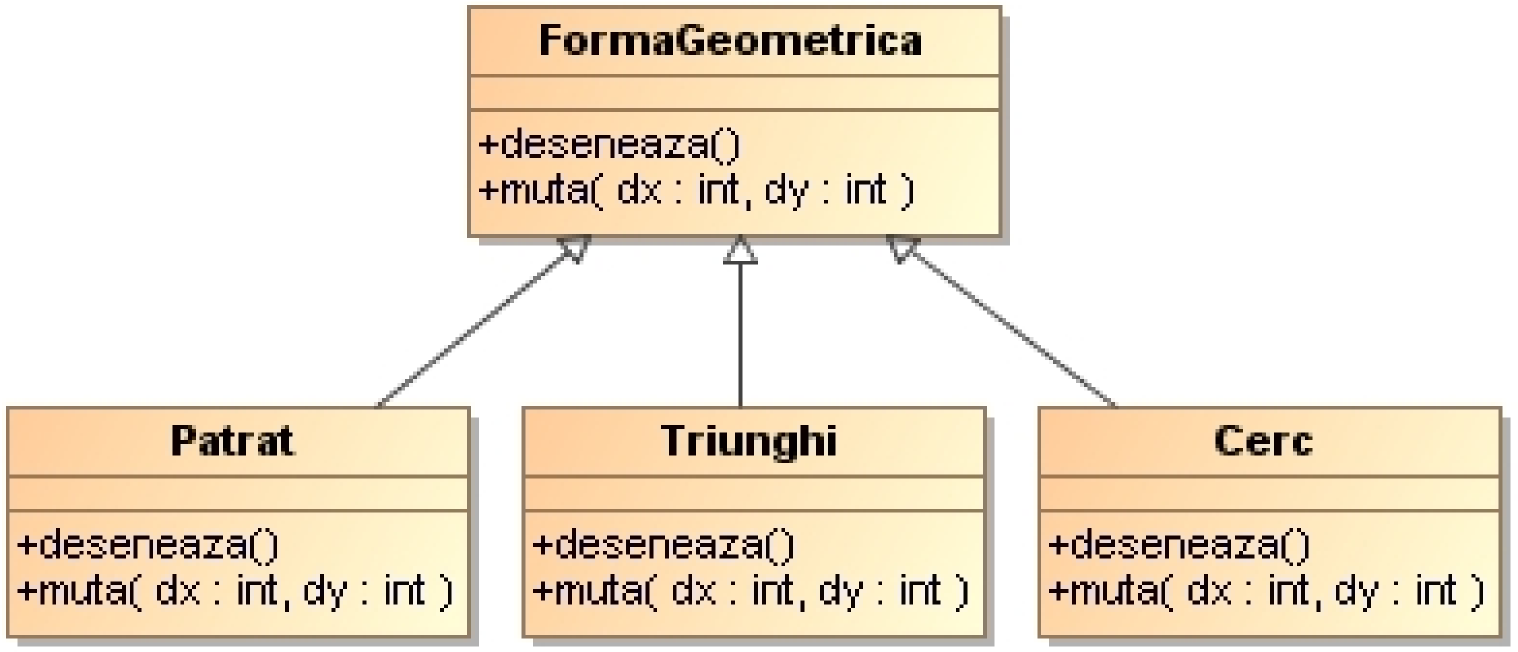
```
Punct p1=new Punct(2,3);
Punct p2=new Punct(2,3);
boolean egale=(p1==p2);   //false;
egale=p1.equals(p2);     //true, Punct must redefine equals
System.out.println(p1);   //toString is called
```

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.


FormaGeometrica
+deseneaza()
+muta(dx : int, dy : int)
Patrat
+deseneaza()
+muta(dx : int, dy : int)
Triunghi
+deseneaza()
+muta(dx : int, dy : int)
Cerc
+deseneaza()
+muta(dx : int, dy : int)

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;
}
```

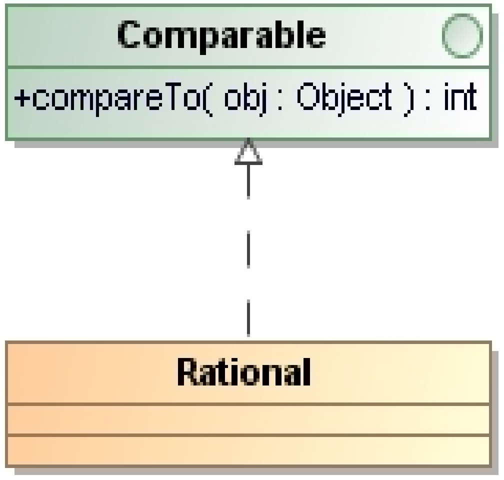
Interface implementation

- 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);
}
```

Collisions

```
interface I1 {
  void f();
}
interface I2 {
  int f(int i);
 }
interface I3 {
  int f();
}

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

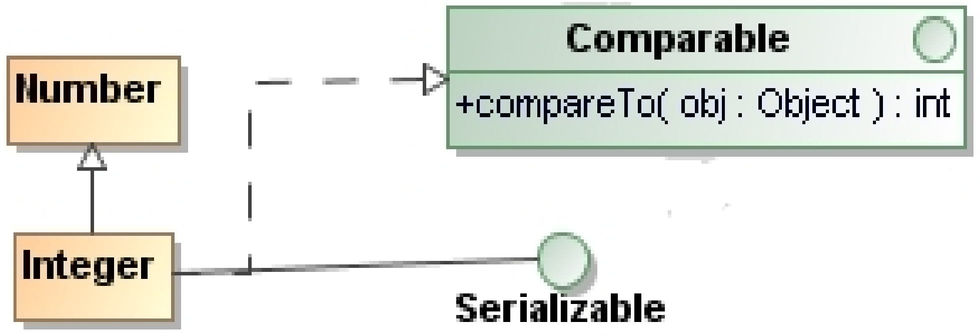
```
class C2 implements I1, I2 {
     public void f() {}
     public int f(int i) { return 1; }
     //overloading
}
class CC implements I1, I3{  //error at compile-time
  //...
}
```

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



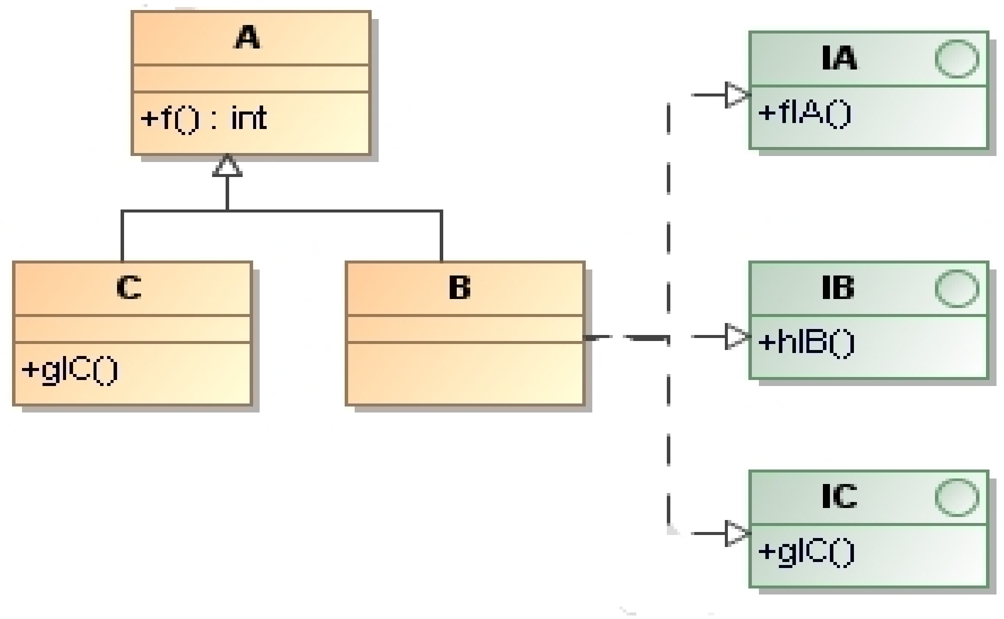
Variables of type interface

- 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); //ERROR!!
```

Variable of type interface

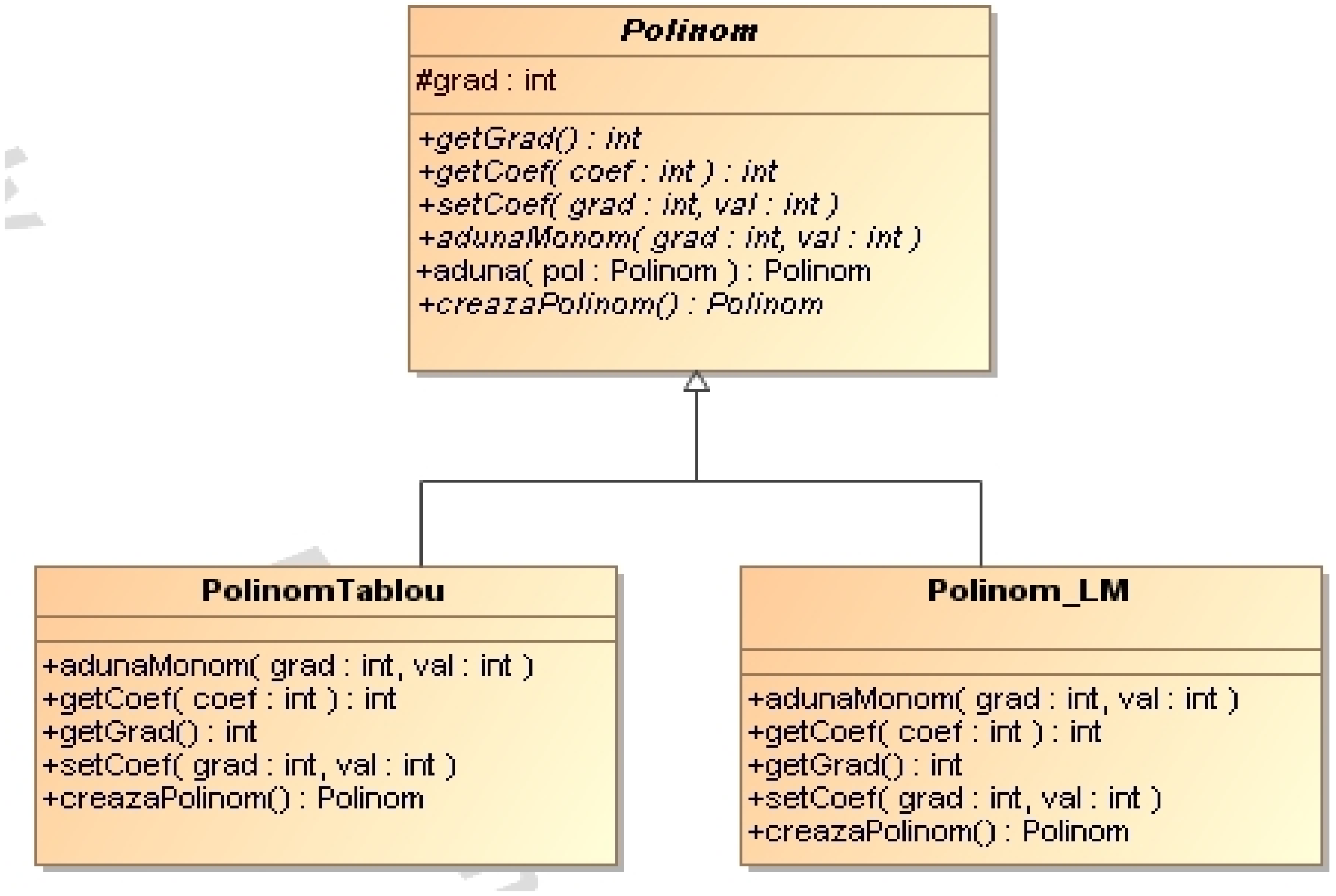
```
B b=new B();
IA ia=b; ia.fIA();

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

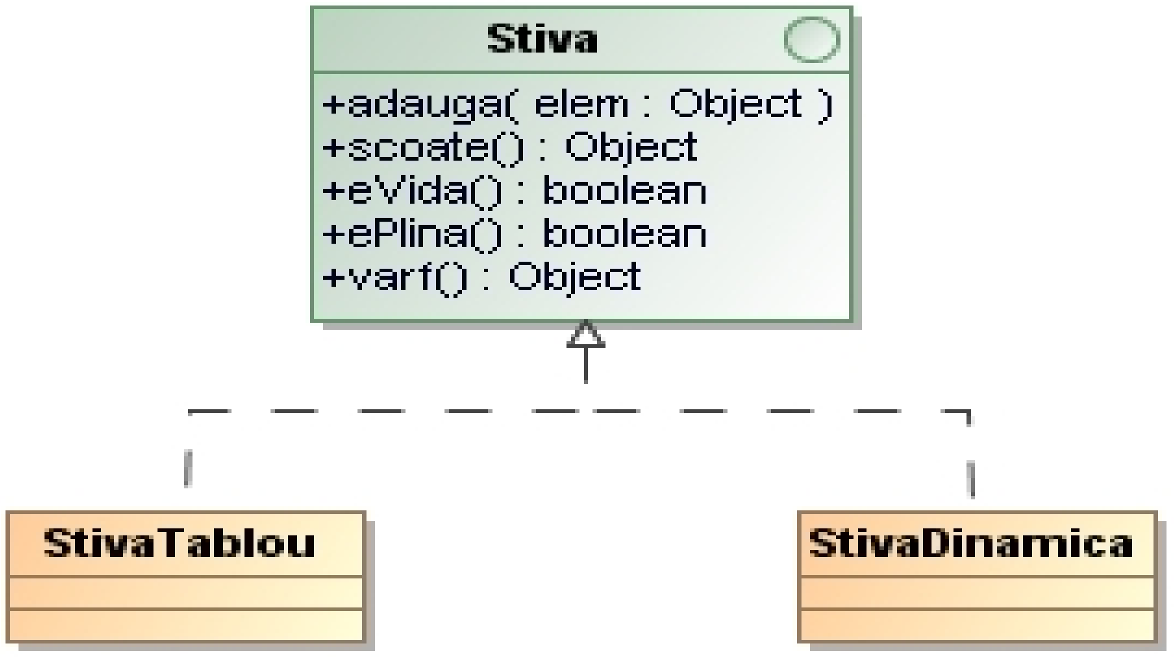
Public, protected, private methods	only public methods.
Have fields	Can have only static and final fields
Have constructors	No constructors.
It is possible to have no any abstract method.	It is possible to have no any methods
Both do not have instance objects	

Abstract classes vs Interfaces



Polinom
#grad : int
+getGrad() : int
+getCoef(coef : int) : int
+setCoef(grad : int, val : int)
+adunaMonom(grad : int, val : int)
+aduna(pol : Polinom) : Polinom
+creazaPolinom() : Polinom
PolinomTablou
+adunaMonom(grad : int, val : int)
+getCoef(coef : int) : int
+getGrad() : int
+setCoef(grad : int, val : int)
+creazaPolinom() : Polinom
Polinom_LM
+adunaMonom(grad : int, val : int)
+getCoef(coef : int) : int
+getGrad() : int
+setCoef(grad : int, val : int)
+creazaPolinom() : Polinom

Abstract Classes vs Interfaces



Packages

- Groups classes and interfaces
- Name space management

- ex. package `java.lang` contains the classes `System, Integer, String`, etc.
- A package is defined by the instruction `package:`

```
//structuri/Stiva.java
package structuri;
public interface Stiva{
   //...
}
```

Obs:

1. `package` must be the first instruction of the java file
2. The file is saved in the folder `structuri` (case-sensitive).

```
//structuri/liste/Lista.java
package structuri.liste;     //folder structuri/liste/Lista.java
public interface Lista{
   //...
}
```

Packages

- Compilation:

if the file Stiva.java is in the folder

```
C:\users\maria\java\structuri
```

the current folder must be:

```
C:\users\maria\java
```

```
C:\users\maria\java> javac structuri/Stiva.java
C:\users\maria\java> javac structuri/liste/Lista.java
```

File `.class` **is saved in the same folder.**

```
C:\users\maria\java\structuri\Stiva.class
C:\users\maria\java\structuri\liste\Lista.class
```

Package

- Using the class

```
package structuri.liste;
public class TestLista{
  public static void main(String args[]){
    Lista li=...
  }
}
```

Compilation:

```
C:\users\maria\java> javac structuri/liste/TestLista.java
```

Running:

```
C:\users\maria\java> java structuri.liste.TestLista
```

Using the classes declared in the packages

```
// structuri/ArboreBinar.java
package structuri;
public class ArboreBinar{
  //...
}
```

- The classes are referred using the following syntax:

```
  [pac1.[pac2.[...]]]NumeClasa
//TestStructuri.java
public class TestStructuri{
  public static void main(String args[]){
    structuri.ArboreBinar ab=new structuri.ArboreBinar();
    //...
  }
}
```

Using the classes declared in the packages

- Instruction `import`:
  - one class:

```
import pac1.[pac2.[...]]NumeClasa;
```

  - All the package classes, but not the subpackages:

```
import pac1.[pac2.[...]]*;
```

- A file may contain multiple import instructions. They must be at the beginning before any class declaration.

```
//structuri/Heap.java
package structuri;
public class Heap{
  //...
}
//Test.java
//fara instructiuni import
public class Test{
  public static void main(String args[]){
    structuri.ArboreBinar ab=new structuri.ArboreBinar();
    structuri.Heap hp=new structuri.Heap();
  }}
```

Using the classes declared in the packages

```
//Test.java
import structuri.ArboreBinar;
public class Test{
  public static void main(String args[]){
    ArboreBinar ab=new ArboreBinar();
    structuri.Heap hp=new structuri.Heap();
  }
}
//Test.java
import structuri.*;
import structuri.liste.*;
public class Test{
  public static void main(String args[]){
    ArboreBinar ab=new ArboreBinar();
    Heap hp=new Heap();
    Lista li=new Lista();
  }
}
```

Package+import

- The instruction `package` must be before any instruction `import`

```
//algoritmi/Backtracking.java
package algoritmi;
import structuri.*;

public class Backtracking{
  //...
}
```

- The package `java.lang` is implicitly imported by the compiler.

Static import

- Starting with version 1.5

```
import static pac1.[pac2.[. ...]]NumeClasa.MembruStatic;
import static pac1.[pac2.[...]]NumeClasa.*;
```

- Allow to use static members of class `NumeClasa` without using the class name.

```
package utile;
public class EncodeUtils {
    public static String encode(String txt){...}
    public static String decode(String txt){...}
}
//Test.java
import static utile.EncodeUtils.*;
public class Test {
    public static void main(String[] args) {
        String txt="aaa";
        String enct=encode(txt);
        String dect=decode(enct);
        //...
    }
}
```

Anonymous package

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

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

//Test.java
public class Test{
  public static void main(String args[]){
    Persoana p=new Persoana();
    Complex c=new Complex();
    //...
  }
}
```

If a file `.java` does not contain the instruction `package`, all the file classes are part of anonymous package.

Name Collision

```
// unu/A.java
package unu;
public class A{
  //...
}
```

```
// doi/A.java
package doi;
public class A{
  //...
}
```

```
//Test.java
import unu.*;
import doi.*;
public class Test{
  public static void main(String[] args){
    A a=new A();  //compilation error
    unu.A a1=new unu.A();
    doi.A a2=new doi.A();
  }
}
```

Access modifiers

- 4 modifiers for the class members:
  - `public:` access from everywhere
  - `protected:` access from the same package and from subclasses
  - `private:` access only from the same class
  - `:` access only from the same package
- Classes (excepting inner classes) and interfaces can be public or nothing.

Access modifiers

```
// structuri/Nod.java
package structuri;
class Nod{
   private Nod urm;
   public Nod getUrm(){...}
   void setUrm(Nod p){...}
  //...
}
```

```
// structuri/Coada.java
package structuri;
public class Coada{
   Nod cap;
   Coada(){ cap.urm=null;}
   Coada(int i){...}
  //...
}
```

```
//Test.java
import structuri.*;
class Test{
  public static void main(String args[]){
     Coada c=new Coada();
     Nod n=new Nod();     //class is not public
     Coada c2=new Coada(2);   //constructor is not public
 }
}
```

Access modifiers

```
package unu;
public class A{
   A(int c, int d){...}
   protected A(int c){...}
   public A(){...}
   protected void f(){...}
   void h(){...}
}
```

```
package unu;
class DA extends A{
   DA(int c){ super(c);}
}
```

```
package doi;
import unu.*;
class DDA extends A{
   DDA(int c){super(c);}
   DDA(int c, int d) {super(c,d);}
   protected void f(){
      super.h();
   }
}
```

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;
  }
  //...
}
```

Advanced Programming Methods

Lecture 2 – Java Exceptions, Generics and Collections

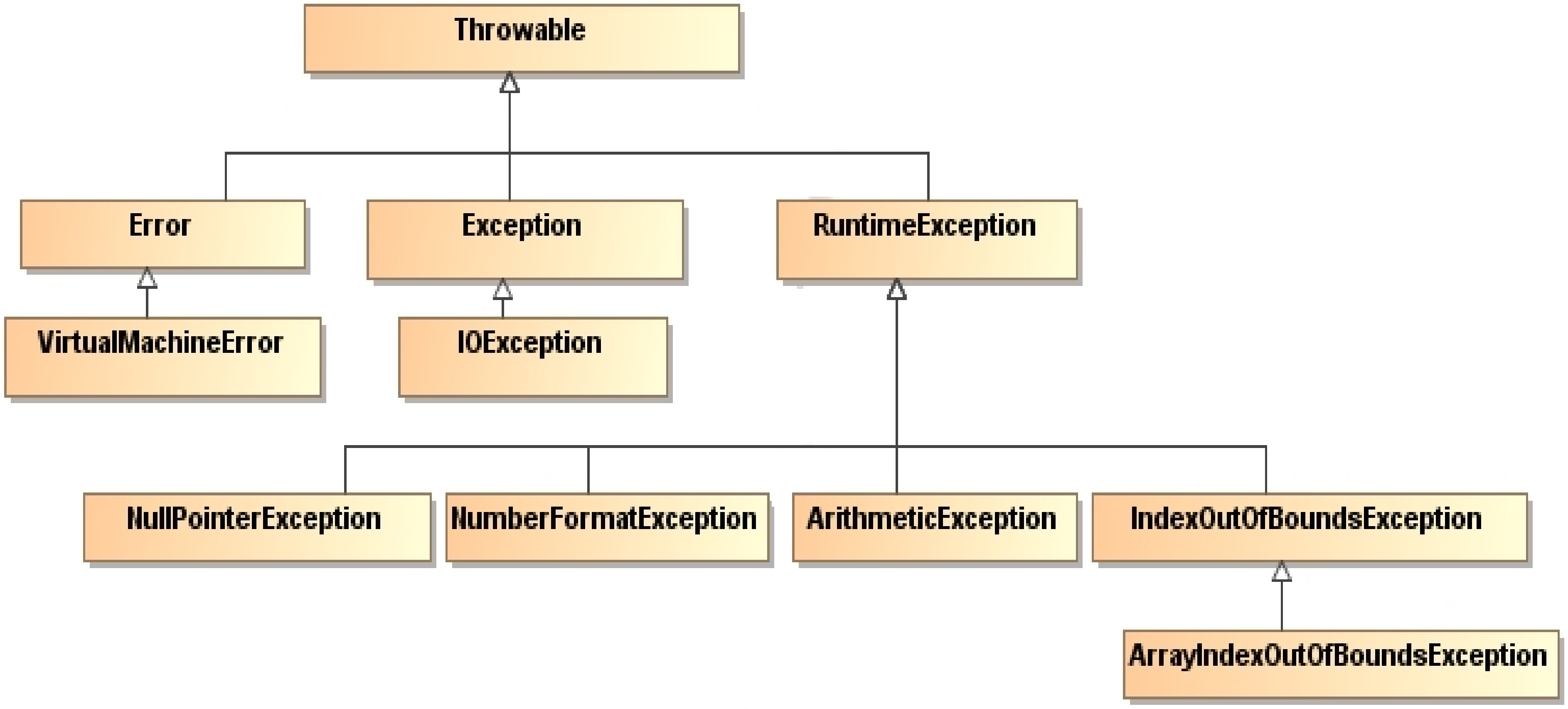
Content

1. **Java Exceptions**
2. **Java Generics**
3. **Java Collections**

JAVA EXCEPTIONS

Java Exceptions

Three types of exceptions:Errors(external to the application), Checked Exceptions(subject to try-catch), and Runtime Exceptions(correspond to some bugs)



Example 1

Program for ax+b=0, where a, b are integers.

```
class P1{
 public static void main(String args[]){
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                    //(3)
      System.out.println("Solutie "+(-b/a));           //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
    }
}
java P1 1 1 //-1
java P1 0 3 //exception, divide by 0
            //Lines 4 or 5 are not longer executed
```

Example 1

Java VM creates the exception object corresponding to that abnormal situation and throws the exception object to those program instructions that generates the abnormal situation.

Thrown exception object can be caught or can be ignored (in our example the program P1 ignores the exception)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at P1.main(P1.java:13)
```

Catching exceptions

Using try-catch statement:

```
try{
  //code that might generates abnormal situations
}catch(TipExceptie numeVariabila){
  //treatment of the abnormal situation
}
```

Execution Flow:

- If an abnormal situation occurs in the block try(), JVM creates an exception object and throws it to the block catch.
- If no abnormal situation occurs, try block normally executes.
- If the exception object is compatible with one of the exceptions of the catch blocks then that catch block executes

Example 2

```
class P2{
    public static void main(String args[]){
     try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                //(3)
        System.out.println("Solutie "+(-b/a));    //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
     }catch(ArithmeticException e){
        System.out.println("Nu exista solutie");  //(6)
     }
    }
}
java P2 1 1 //Solutie -1
java P2 0 3 // Nu exista solutie
            // (1), (2), (3), (6) are executed
```

Multiple catch clauses

```
try{
  //code with possible errors
}catch(TipExceptie_1 numeVariabila_1){
  //instructions
}catch(TipExceptie_2 numeVariabila_2){
  //instructions
}...
catch(TipExceptie_n numeVariabila_n){
  // instructions
}
```

Example 3

```
class P3{
    public static void main(String args[]){
     try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      if (b % a==0)                //(3)
        System.out.println("Solutie "+(-b/a));     //(4)
      else
        System.out.println("Nu exista solutie intreaga"); //(5)
     }catch(ArithmeticException e){
        System.out.println("Nu exista solutie");  //(6)
     }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("java P3 a b");      //(7)
     }    }
}
java P3 1 1 //Solution -1
java P3 0 3 // Nu exista solutie
            // (1), (2), (3), (6) are executed
java P3 1  //java P3 a b
```

Nested try statements

```
class P4{
  public static void main(String args[]){
    try{
      int a=Integer.parseInt(args[0]);    //(1)
      int b=Integer.parseInt(args[1]);    //(2)
      try{
         if (b % a==0)                 //(3)
          System.out.println("Solutie "+(-b/a));         //(4)
         else
          System.out.println("Nu exista solutie intreaga"); //(5)
      }catch(ArithmeticException e){
          System.out.println("Nu exista solutie");  //(6)
      }
      }catch(ArrayIndexOutOfBoundsException e){
        System.out.println("java P4 a b");      //(7)
       } }}
java P4 1 1 //Solutie -1
java P4 0 3 // Nu exista solutie
java P4 1  //java P4 a b
```

Nested try statements

```
try{
 //...
  try{
    //...
  }catch(TipExceptie_ii numeVar_ii){
   //...
   }
}catch(TipExceptie_1 numeVar_1){
  //instructiuni
}catch(TipExceptie_n numeVar_n){
  // ...
   try{
    //...
   }catch(TipExceptie_in numeVar_in){
   //...
   }
}
```

Finally clause

The finally clause is executed in any situation:

```
try{
  //...
}catch(TipExceptie_1 numeVar_1){
  //instructiuni
}[catch(TipExceptie_n numeVar_n){
  // ...
}]
[finally{
  //instructiuni
}]
```

Finally Clause

```
A
try{
   B
}catch(TipExceptie nume){
    C
}finally{
  D
}
E
```

Block D executes:

- After A and B (before E) if no exception occurs in B. (A, B, D, E)
- After C, if an exception occurs in B and that exception is caught (A, a part of B, C, D, E).
- Before exit from the method:
  - » An exception occurs in B, but is not caught (A, a part of B, D).
  - » An exception occurs in B, it is caught but a return exists in C (A, a part of B, C, D).
  - » If a return exists in B (A, B, D).

Finally Clause

```
public void writeElem(int[] vec) {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileWriter(“fisier.txt"));
         for (int elem:vec)
             out.print(" "+elem);
    } catch (IOException e) {
         System.err.println("IOException: "+e);
    }finally{
         if (out != null)
             out.close();
    }
}
```

General form of Try statement

```
try{
  //code with possible errors
}[catch(TipExceptie_1 e_1){
  //...
}]
//...
[catch(TipExceptie_n e_n){
   //...
}]
[finally{
  //instructions
 }]
```

Defining exception classes

- By deriving from class `Exception:`

```
public class ExceptieNoua extends Exception{
  public ExceptieNoua(){}
  public ExceptieNoua(String mesaj){
    super(mesaj);
  }
}
```

Exceptions Specification

- Use keyword `throws` in method signatures:

```
public class ExceptieNoua extends Exception{}

public class A{
  public void f() throws ExceptieNoua{
    //...
  }
}
```

- Many exceptions can be specified (their order does not matter):

```
public class Exceptie1 extends Exception{}
public class B{
  public int g(int d) throws ExceptieNoua, Exceptie1{
    //...
 }
}
```

Throwing exceptions

- Statement `throw :`

```
public class B{
  public int  g(int d) throws ExceptieNoua,Exceptie1{
    if (d==3)
        return 10;
    if (d==7)
        throw new ExceptieNoua();
    if (d==5)
        throw new Exceptie1();
    return 0;
  }
  //...
}
```

- Statement throw throws away the exception object and the method execution is interrupted.
- All exceptions thrown inside a method must be specified in the method signature.

Calling a method having exceptions

- use try-catch to treat the exception:

```
public class C{
  public void h(A a){
    try{
      a.f();
    }catch(ExceptieNoua e){
      System.err.println(" Exceptie "+e);
    }
  }
}
```

- Throwing away an uncaught exception (uncaught exception must be specified in the signature):

```
public class C{
  public void t(B b) throws ExceptieNoua {
    try{
      int rez=b.g(8);
    }catch(Exceptie1 e){         //only Exceptie1 is caught
      System.err.println(" Exceptie "+e);
    }
  }
}
```

Exception specification

- The subclass constructor must specify all the base class constructor (explicitly or implicitly called) in its signature.
- The subclass constructor may add new exceptions to its signature.

```
public class A{
  public A() throws Exceptie1{
  }
  public A(int i){ }
  //...
}

public class B extends A{
  public B() throws Exceptie1{  }
  public B(int i){
    super(i);
  }
  public B(char c) throws Exceptie1, ExceptieNoua{
  }
  //...
}
```

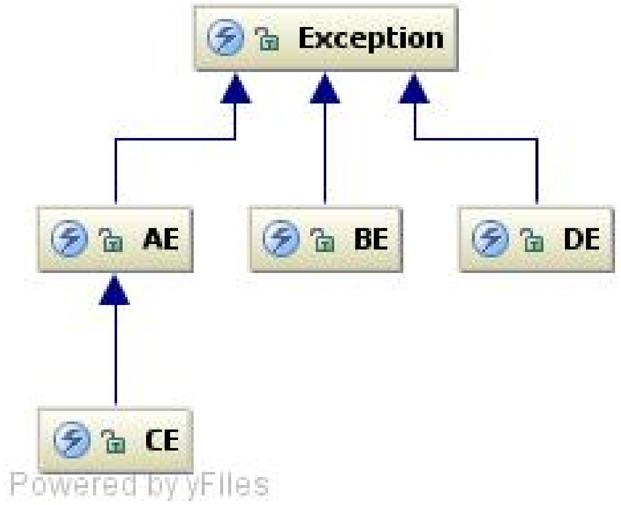
Exceptions and method overriding

- An overriding method may declare a part of the exceptions of the overridden method.
- An overriding method may add only new exceptions which are inherited from the overridden method exceptions
- The same rules are applied for the interfaces.

```
public class AA {
    public void f() throws Exceptie1, Exceptie2{ }
    public void g(){ }
    public void h() throws Exceptie1{  }
}

public class BB extends AA{
    public void f() throws Exceptie1{ }  //Exceptie2 is not declared
    public void g() throws Exceptie2{ }  //not allowed
    public void h() throws Exceptie3{ }
}
public class Exceptie3 extends Exceptie1{...}
```

Exceptions and method overriding

```
public class A {
    public void f() throws AE, BE {}
    public void g() throws AE{}
}

public class B extends A{
    public void g(){}
    public void f() throws AE, BE, CE{}
    public void f() throws AE, BE, DE{}
    public void g() throws DE{}
}
//?
```



Exceptions order in catch clauses

- The order of catch clauses is important since the JVM selects the first catch clause on which the try block thrown exception matches.
- An exception A matches an exception B if A and B have the same class or A is a subclass of B.

```
public class C {
  public void g(B b){
    try{
        b.f();
    } catch(Exception e){...

    } catch (CE ce) { ...
    } catch (AE ae) {...
    } catch (BE be) {...
    }
  }
}
```

```
public class C {
  public void g(B b){
    try{
        b.f();
    } catch (CE ce) { ...
    } catch (AE ae) {...
    } catch (BE be) {...
    } catch(Exception e){...
    }
  }
}
```

Lost exceptions

```
public class C {
    public void g(B b){
        try{
            b.f();
        }
        catch (CE ce) { }  //At least an error message must be printed
        catch (AE ae) { }
        catch (BE be) { }
    }
}
```

Re-throwing an exception

- A caught exception can be re-thrown

```
public class C {
 public int h(A a) throws Exceptie4, BE {
        try {
            a.f();
        } catch (BE be) {
            System.out.println("Exceptie rearuncata "+be);
            throw be;
        } catch (AE ae) {
            throw new Exceptie4("mesaj", ae);
        }
        return 0;
    }
}
public class Exceptie4 extends Exception {
    public Exceptie4() { }
    public Exceptie4(String message) {
        super(message);
    }
    public Exceptie4(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Exception class

- Constructors:
  - `Exception()`
  - `Exception(String message)`
  - `Exception(String message, Throwable cause)`
  - `Exception(Throwable cause)`
- Methods:
  - `getCause(): Throwable`
  - `getMessage(): String`
  - `printStackTrace()`
  - `printStackTrace(PrintStream s)`

```
public class C {
 public int h(A a) throws BE {
        try {
            a.f();
        } catch (BE be) {
            System.out.println("Exceptie rearuncata "+be.getMessage());
            throw be;
        } catch (AE ae) {
            ae.printStackTrace();
        }
        return 0;
    }
}
```

Unchecked Exceptions

- Checked exceptions are those which are derived from class `Exception`
- Exceptions may be derived from class `RuntimeException`. They are named *unchecked exceptions*.

```
public class ExceptieNV extends RuntimeException{
    public ExceptieNV() { }
    public ExceptieNV(String message) {
     super(message);
    }
}
```

- Uncheked Exceptions must not be declared in the method signature.

- Unchecked Exceptions are used only for the abnormal situations that can not be solved (the recovering cannot be done).

JAVA GENERICS

Generics

- Parameterized types
- Started with Java 1.5
- Different than C++ templates
  - It does not generate a new class for each parameterized type
  - The constraints can be imposed on the type variables of the parameterized types.
- Motivation:

```
Stiva s=new Stiva();     //stack of Object
s.push("Ana");
s.push(new Persoana("Ana", 23));
Persoana p1 =(Persoana)s.pop();
Persoana p2 =(Persoana)s.pop();
  //correct at compile-time, error at execution time
```

Generic Class declaration

```
[access_mode] class ClassName <TypeVar1[, TypeVar2[, ...]] >{
   TypeVar1 field1;
  [declarations of fields]
  [declarations and definitions of methods]
}
```

Obs:

Type variables must be upper letters(for example E for element, K for key, V for value, T, U, S ...).

```
public class Stiva<E>{
  private class Nod<T>{
    T info;
    Nod<T> next;
    Nod(){info=null; next=null;}
    Nod(T info, Nod next){
         this.info=info;
         this.next=next;
    }
  }//class Nod
  Nod<E> top;
  //...
}
```

Object creation

```
public class Test{
  public static void main(String[] args){
    Stiva<String> ss=new Stiva<String>();
    ss.push("Ana");
    ss.push("Maria");
    ss.push(new Persoana("Ana", 23));    //error at compile-time
    String elem=ss.pop();                //NO CAST

   Stiva<Persoana> sp=new Stiva<Persoana>();
   sp.push(new Persoana("Ana", 23));
   sp.push(new Persoana("Maria", 10));

   Dictionar<String, String> dic=new Dictionar<String, String>();
   dic.add("abc", "ABC");
   dic.add(23, "acc");  //error at compile-time
   dic.add("acc", 23);  //error la compile-time
  }
}
```

Object creation

Type variables can be instantiated only with reference types. Primitive types: int, byte, char, float, double,.... are not allowed. Therefore the corresponding reference types are used.

```
  Stiva<int> si=new Stiva<int>();
//error at compile-time


 Stiva<Integer> si=new Stiva<Integer>();
```

primitive types	**Corresponding reference types**
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

- Java 1.5
- Autoboxing: automatic conversion of a value of a primitive type to an object instance of a corresponding reference type when an object is expected, and vice-versa when a primitive value is expected.

```
Stiva<Integer> si=new Stiva<Integer>();
si.push(23);          //autoboxing
si.push(new Integer(23));
int val=si.pop();


Character ch = 'x';
char c = ch;
```

Generic methods

- Methods with type variables

```
   class ClassName[<TypeVar ...>]{
[access_mod] <TypeVar1[, TypeVar2[,...]]> TypeR nameMethod([list_param]){
  }
  //...
}
```

Obs:

- Static methods cannot use the type variables of the class.
- A generic method can contain type variables different than those used by the generic class.
- A generic method can be defined in a non-generic class.

Generic methods

```
public class GenericMethods {
   public <T> void f(T x) {
      System.out.println(x.toString());
  }

   public static <T> void copy(T[] elems, Stiva<T> st) {
         for(T e:elems)
             st.push(e);
  }

}
```

Calling a generic method

- The compiler automatically infers the types which instantiate the type variables when a generic method is called.

```
public class A {
   public <T> void print(T x) {
      System.out.println(x);
  }

  public static void main(String[] args) {
    A a=new A();
    a.print(23);
    a.print(“ana”);
    a.print(new Persoana(“ana”,23));
  }
}
```

Calling a generic method

- The instantiations of the type vars are explicitly given:
  - Instance method:

```
 a.<Integer>print(3);
 a.<Persoana>print(new Persoana(“Ana”,23));
```

  - Static method :

```
NameClass.<Typ>nameMethod([parameters]);
//...
Integer[] ielem={2,3,4};
Stiva<Integer> st=new Stiva<Integer>();
GenericMethods.<Integer>copy(ielem, st);
//
```

  - Non-static method in a class:

```
this.<Typ>nameMethod([parameters]);
class A{
   public <T> void print(T x){...}
   public void g(Complex x){
     this.<Complex>print(x);
   }
}
```

Generic arrays

- Cannot be created using new:

```
      T[] elem=new T[dim];  //error at compile time
```

but we can use:

```
   T[] elem=(T[])new Object[dim];   //warning at compile-time
```

- Alternatives:
  - Using `Array.newInstance`

```
import java.lang.reflect.Array;
public class Stiva <E>{
    private E[] elems;
    private int top;
    @SuppressWarnings("unchecked")
    public Stiva(Class<E> tip) {
       elems= (E[])Array.newInstance(tip, 10);
       top=0;
    }
   //...
}
Stiva<Integer> si=new Stiva<Integer>(Integer.class);
```

  - Using `ArrayList` instead of array.

Generic arrays

- Use an array of Object, but read operation requires an explicit cast:

```
public class Stiva <E>{
    private Object[] elems;
    private int top;
    public Stiva() {
      elems=new Object[10];
        top=0;
    }
    public void push(E elem){
        elems[top++]=elem;
    }
    @SuppressWarnings("unchecked")
    public E pop(){
        if (top>0)
            return (E)elems[--top];
        return null;
    }
  //...
}
```

Erasure

- Java does not create a new class for each new instantiation of the type variables in case of the generic classes.
- The compiler erases all type variables and replaces them with their upper bounds (usually Object) and explicit casts are inserted when it is necessary

```
public class A {
  public String f (Integer ix){
    Stiva<String> st=new Stiva<String>();
    Stiva sts=st;
    sts.push(ix);
    return st.top();
  }
}
```

```
public class A {
  public String f (Integer ix){
    Stiva st=new Stiva();
    Stiva sts=st;
    sts.push(ix);
    return (String)st.top();
  }
}
```

compilation

- Reason: backward compatibility with the non-generic Java versions
- The generic class is not recompiled for each new instantiation of the type variables like in C++.

Bounds

```
public class ListOrd<E> {
    private class Nod<E>{
        E info;
        Nod<E> nxt;
        public Nod(){ info=null; nxt=null; }
        private Nod(E info, Nod<E> nxt) { this.info = info; this.nxt = nxt; }
        private Nod(E info) { this.info = info; nxt=null; }
    }
    private Nod<E> head;
    public ListOrd(){ head=null;}
    public void add(E elem){
        if (head==null){
            head=new Nod<E>(elem);
            return;
        }
        if (/*compare elem to head.info*/){
            head=new Nod<E>(elem,head);
        }else {...}
    }
}
```

Bounds

- Type variables can have constraints (namely bounds) using `extends`.

```
T extends E    //T is the type E or is a subtype of E.
```

- General form of the constraint:

  `T extends [C &] I`$_1$ `[& I`$_2$ `&...& I`$_n$`]`

  T inherits the class C and implements the interfaces I_1, ... I_n.

- At compile-time T is replaced by the first element from the constraint expression:

  `T extends C` `//T is replaced by C`

  `T extends C & I`$_1$ `& I`$_2$ `//T is replaced by C`

  `T extends I`$_1$ `& I`$_2$ `//T is replaced by I1`

  `T extends I`$_1$ `//T is replaced by I1`

  `T` `//T is replaced by Object`

- If T has constraints then through T we can call any method from the class and interfaces specified as bounds.

Bounds

```
public interface Comparable<E>{
   int compareTo(E e);
}
public class ListOrd<E extends Comparable<E>> {
    private class Nod<E>{...}
    private Nod<E> head;
    public ListOrd(){ head=null;}
    public void add(E elem){
        if (head==null){
            head=new Nod<E>(elem);
            return;
        }
        if (elem.compareTo(head.info)<0){
            head=new Nod<E>(elem,head);
        }else {...}
    }
     public E retElemPoz(int poz){
        //...
     }
}
```

Wildcards

```
    ListOrd<String> ls=new ListOrd<String>();
    ListOrd<Object> lo=ls; //ASSUME this is CORRECT
    lo.add(23);
    String s=ls.retElemPoz(0);  //ERROR
```

Obs:

If `SB` is a subtype of `T` and `G` is a generic container class then `G<SB>` is not a subtype of `G<T>`.

```
void printLista(ListOrd<Object> lo){
   for(Object o:lo)
    System.out.println(o);
}
...
  ListOrd<String> ls=new ListOrd<String>();
  ls.add("mere");
  ls.add("pere");
  printLista(ls);        //error at compile-time
```

Wildcards

`We use ? to denote any type (or unknown type)`

```
void printLista(ListOrd<?> lo){
   for(Object o:lo)
    System.out.println(o);
}
```

Obs:

1. When we use `?`, the elements can be considered to be of type `Object` (upper bound).
2. When we use `?` to declare an instance, the instance elements cannot be read or write, the only allowed operations are to read Object and to write null.

```
ListOrd<String> ls=new ListOrd<String>();
ls.add("mere");
ls.add("pere");
ListOrd<?> ll=ls;
ll.add("portocale");     //error
ll.update(1, "struguri");//error
Object el=ll.retElemPoz(0);
```

Bounded Wildcards

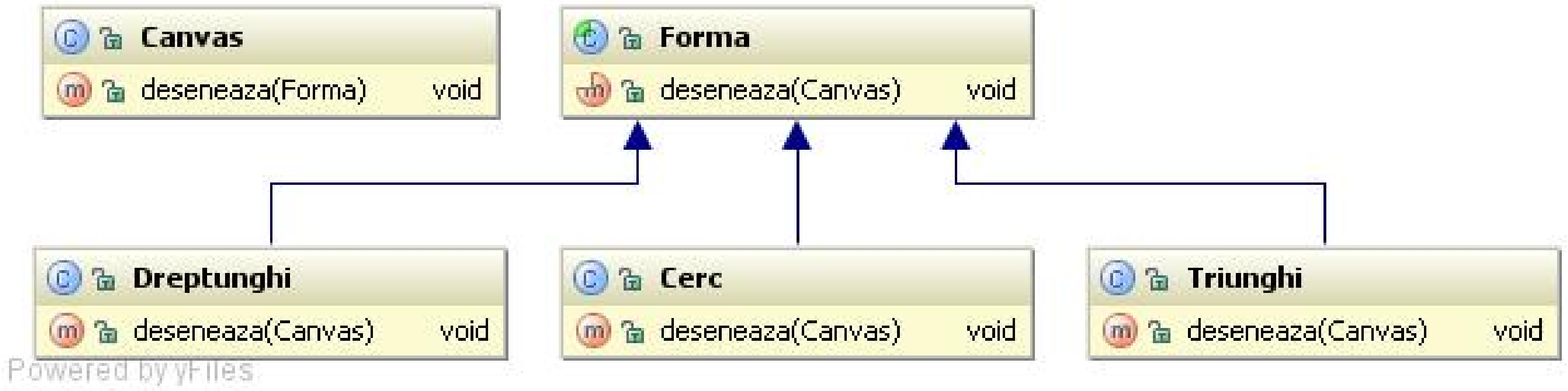
We can specify bounds for ?:

- Upper bound by `extends`: `? extends C` or `? extends I`
- Lower bound by `super`: `? super C (any superclass of C)`

1. Upper bound means that we can read elements of the type (or of superclass of the type) given by the upper bound.
2. Lower bound means that we can write elements of type (or of subclasses of the type) given by the lower bound.

```
ListOrd<Angajat> la=new ListOrd<Angajat>();
la.add(new Angajat(...));
ListOrd<? extends Persoana> lp=la;
lp.add(new Angajat(...));  //error at compile time
Persoana p=lp.retElemPoz(0);
lp.retElemPoz(0).getNume();
ListOrd<? super Angajat> linf=la;
linf.add(new Angajat(...));  //correct
```

Bounded Wildcards



```
public class Canvas {
   public void deseneaza(Forma f){ f.deseneaza(this); }
  public void deseneaza(ListOrd<Forma> lf){
     for(Forma f: lf)
        f.deseneaza(this);
   }
}
//...
Canvas c=new Canvas();
ListOrd<Cerc> lc=new ListOrd<Cerc>();
c.deseneaza(lc);     //error at compile time
```

Bounded Wildcards

```
public class Canvas {
    public void deseneaza(Forma f){ f.deseneaza(this); }
   public void deseneaza(ListOrd<? extends Forma> lf){
     for(Forma f: lf)
         f.deseneaza(this);
    }
}
//...
Canvas c=new Canvas();
ListOrd<Cerc> lc=new ListOrd<Cerc>();
c.deseneaza(lc);  //correct
ListOrd<? extends Forma> ll=lc;
ll.add(new Cerc());       //error at compile time;
```

JAVA COLLECTIONS

Java Collections Framework (JCF)

A *collection* is an object that maintains references to others objects

JCF is part of the `java.util` package and provides:

Interfaces

- Each defines the operations and contracts for a particular type of collection (List, Set, Queue, etc)
- Idea: when using a collection object, it's sufficient to know its interface

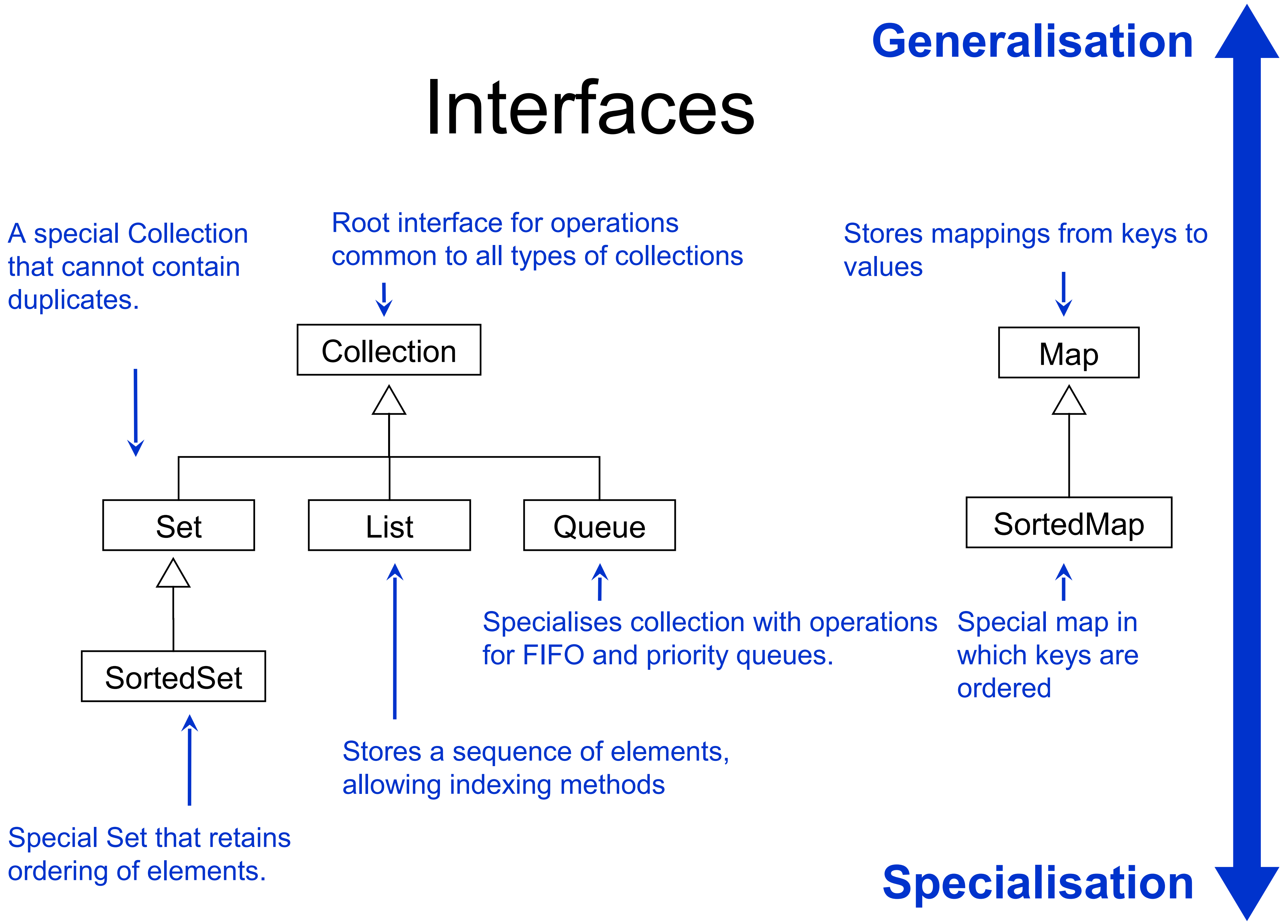
Implementations

- Reusable classes that implement above interfaces (e.g. LinkedList, HashSet)

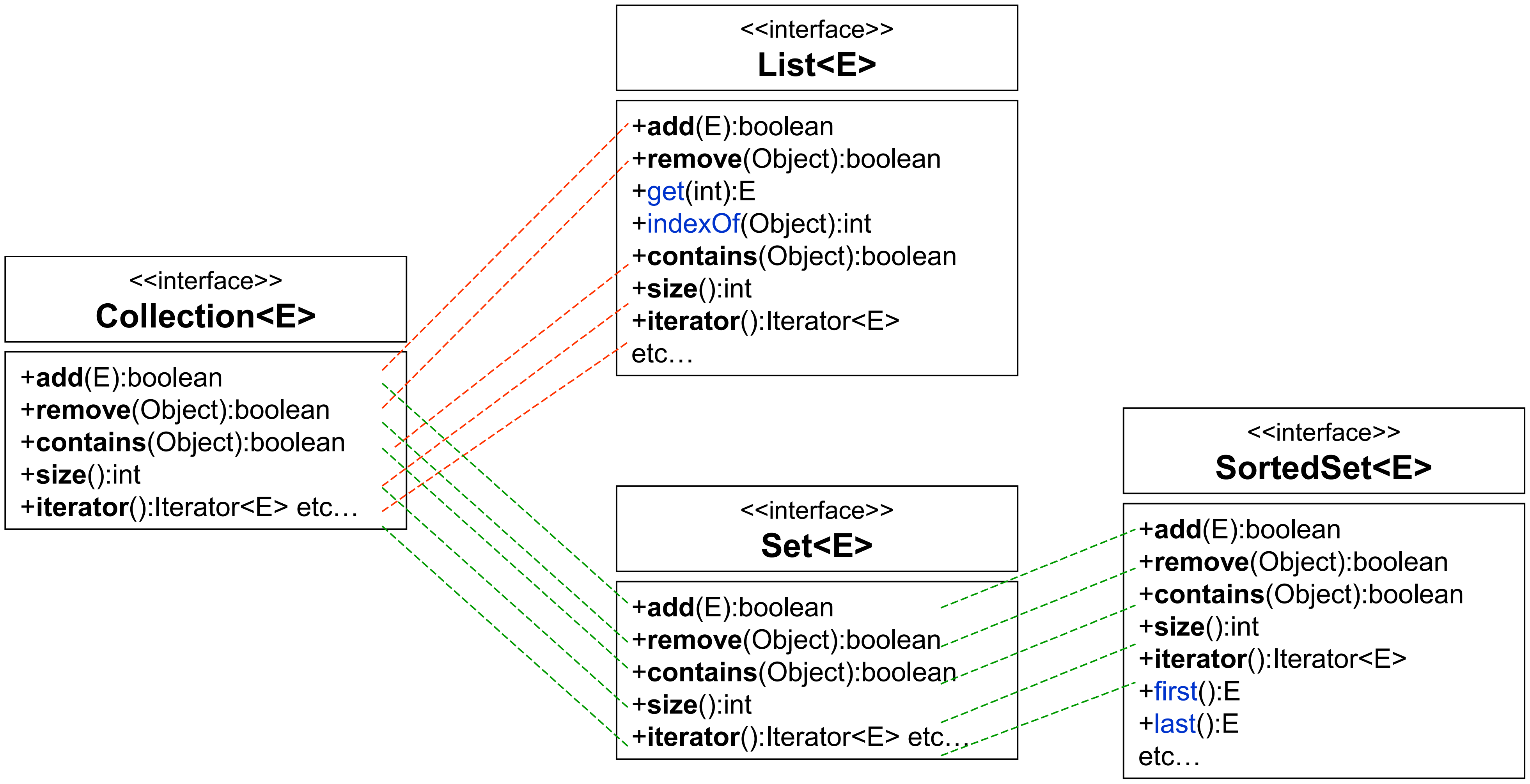
Algorithms

- Useful polymorphic methods for manipulating and creating objects whose classes implement collection interfaces
- Sorting, index searching, reversing, replacing etc.

Interfaces


Generalisation
A special Collection that cannot contain duplicates.
Root interface for operations common to all types of collections
Stores mappings from keys to values
Collection
Map
Set
List
Queue
SortedMap
SortedSet
Specialises collection with operations for FIFO and priority queues.
Special map in which keys are ordered
Stores a sequence of elements, allowing indexing methods
Special Set that retains ordering of elements.
Specialisation

Expansion of contracts



The Collection Interface

- The Collection interface provides the basis for List-like collections in Java. The interface includes:

```
boolean add(Object)
boolean addAll(Collection)
void clear()
boolean contains(Object)
boolean containsAll(Collection)
boolean equals(Object)
boolean isEmpty()
Iterator iterator()
boolean remove(Object)
boolean removeAll(Collection)
boolean retainAll(Collection)
int size()
Object[] toArray()
Object[] toArray(Object[])
```

List Interface

- Lists allow duplicate entries within the collection
- Lists are an ordered collection much like an array
  - Lists grow automatically when needed
  - The list interface provides accessor methods based on index
- The List interface extends the Collections interface and add the following method definitions:

```
void add(int index, Object)
boolean addAll(int index, Collection)
Object get(int index)
int indexOf(Object)
int lastIndexOf(Object)
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
Object set(int index, Object)
List subList(int fromIndex, int toIndex)
```

Set Interface

- The Set interface also extends the Collection interface but does not add any methods to it.
- Collection classes which implement the Set interface have the add stipulation that Sets CANNOT contain duplicate elements
- Elements are compared using the equals method
- NOTE: exercise caution when placing mutable objects within a set. Objects are tested for equality upon addition to the set. If the object is changed after being added to the set, the rules of duplication may be violated.

SortedSet Interface

- SortedSet provides the same mechanisms as the Set interface, except that SortedSets maintain the elements in ascending order.
- Ordering is based on natural ordering (Comparable) or by using a Comparator.

java.util.**Iterator<E>**

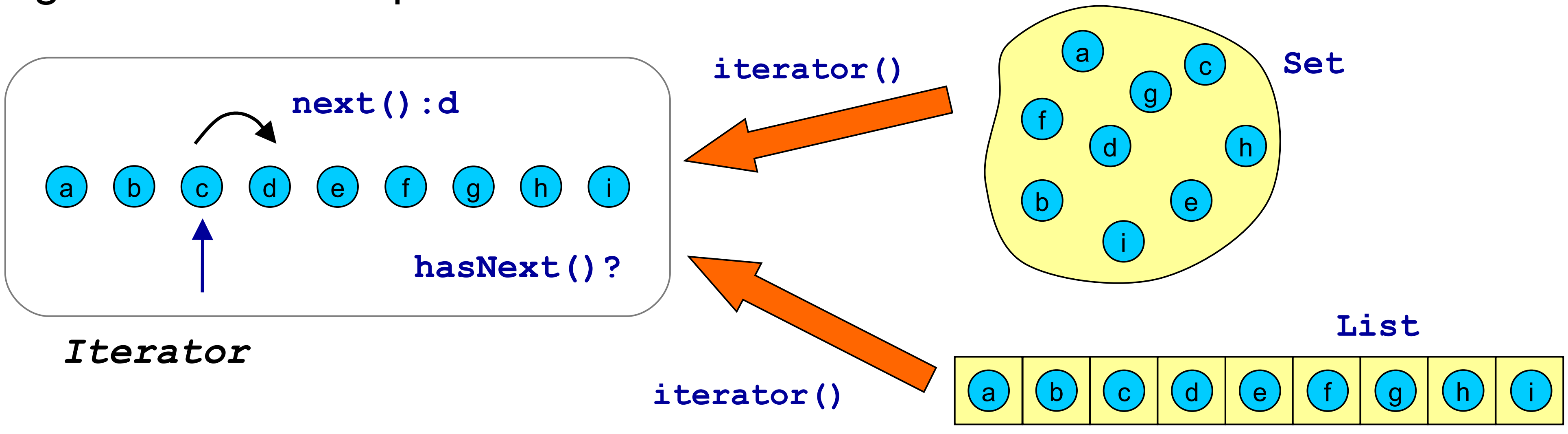
Think about typical usage scenarios for Collections

Retrieve the list of all patients

Search for the lowest priced item

More often than not you would have to traverse every element in the collection – be it a List, Set, or your own datastructure

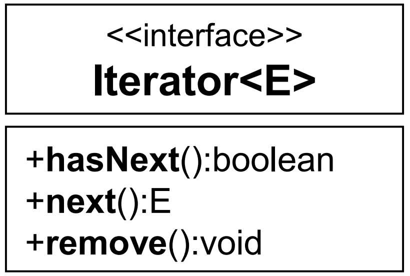
Iterators provide a generic way to traverse through a collection regardless of its implementation



Using an `Iterator`

Quintessential code snippet for collection iteration:

```
public void list(Collection<T> items) {
  Iterator<T> it = items.iterator();
  while(it.hasNext()) {
    Item item = it.next();
    System.out.println(item.toString());
  }
}
```



Design notes:

- Above method takes in an object whose class implements Collection
  - List, ArrayList, LinkedList, Set, HashSet, TreeSet, Queue, MyOwnCollection, etc
- We know any such object can return an Iterator through method iterator()
- We don’t know the exact implementation of Iterator we are getting, but **we don’t care**, as long as it provides the methods next() and hasNext()
- Good practice: **Program to an interface!**

java.lang.**Iterable<T>**

```
for (Item item : items) {
  System.out.println(item);
}
```

=

```
Iterator<Item> it = items.iterator();
while(it.hasNext()) {
  Item item = it.next();
  System.out.println(item);
}
```

This is called a "**for-each**" statement

For each **`item`** in **`items`**

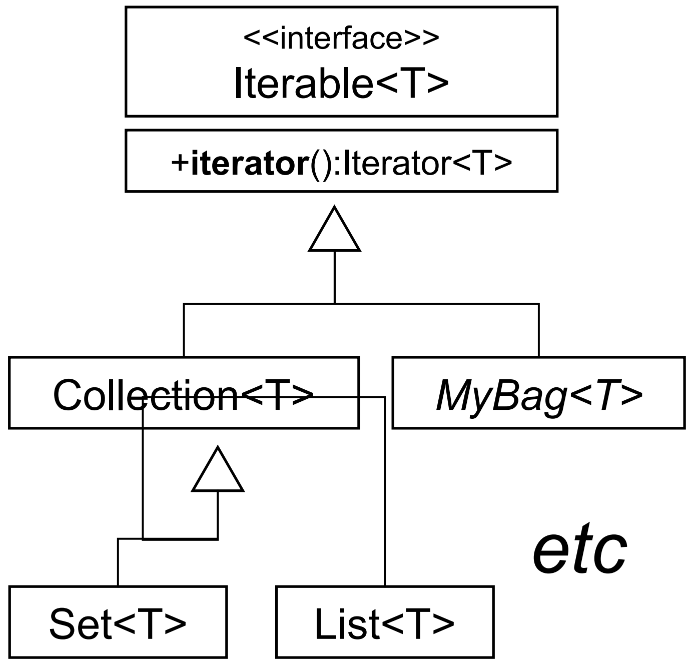
This is possible as long as items is of type **`Iterable`**

Defines single method **`iterator()`**

`Collection` (and hence all its subinterfaces) implements **`Iterable`**

You can do this to your own implementation of **`Iterable`** too!

To do this you may need to return your own implementation of **`Iterator`**



`java.util.`**`Collections`**

Offers many very useful utilities and algorithms for manipulating and creating collections

Sorting lists

Index searching

Finding min/max

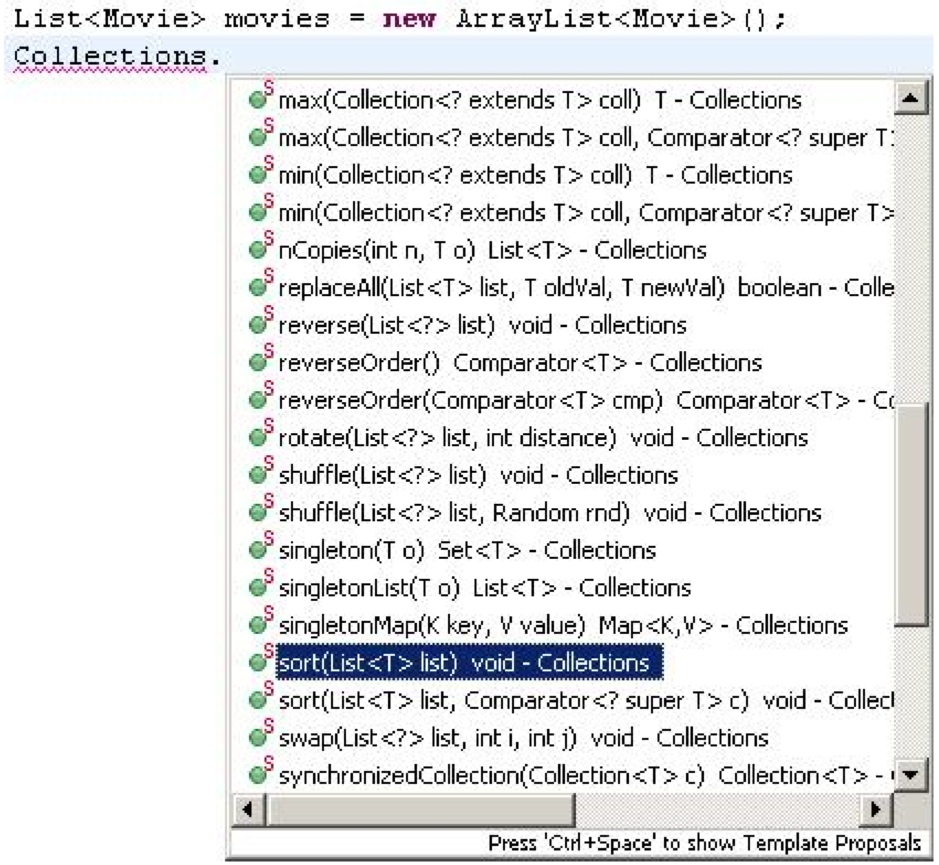
Reversing elements of a list

Swapping elements of a list

Replacing elements in a list

Other nifty tricks

Saves you having to implement them yourself → **reuse**



```
List<Movie> movies = new ArrayList<Movie>();
Collections.
```

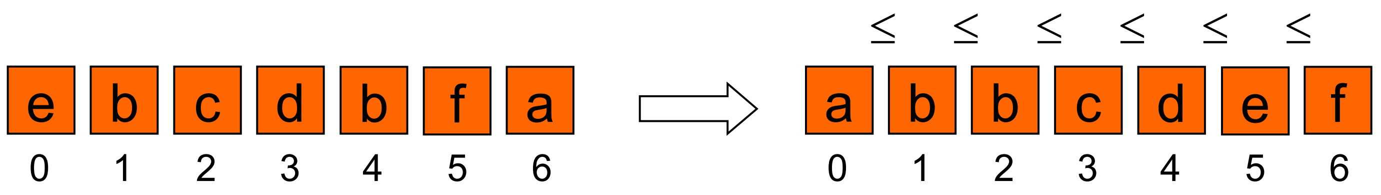
```
max(Collection<? extends T> coll)  T - Collections
max(Collection<? extends T> coll, Comparator<? super T:
min(Collection<? extends T> coll)  T - Collections
min(Collection<? extends T> coll, Comparator<? super T>
nCopies(int n, T o)  List<T> - Collections
replaceAll(List<T> list, T oldVal, T newVal)  boolean - Colle
reverse(List<?> list)  void - Collections
reverseOrder()  Comparator<T> - Collections
reverseOrder(Comparator<T> cmp)  Comparator<T> - Cc
rotate(List<?> list, int distance)  void - Collections
shuffle(List<?> list)  void - Collections
shuffle(List<?> list, Random rnd)  void - Collections
singleton(T o)  Set<T> - Collections
singletonList(T o)  List<T> - Collections
singletonMap(K key, V value)  Map<K,V> - Collections
sort(List<T> list)  void - Collections
sort(List<T> list, Comparator<? super T> c)  void - Collect
swap(List<?> list, int i, int j)  void - Collections
synchronizedCollection(Collection<T> c)  Collection<T> -
Press 'Ctrl+Space' to show Template Proposals
```

Comparable and Comparators

- You will have noted that some classes provide the ability to sort elements.
  - How is this possible when the collection is supposed to be de-coupled from the data?
- Java defines two ways of comparing objects:
  - The objects implement the Comparable interface
  - A Comparator object is used to compare the two objects
- If the objects in question are Comparable, they are said to be sorted by their "natural" order.
- Comparable object can only offer one form of sorting. To provide multiple forms of sorting, Comparators must be used.

`Collections.sort()`

Java's implementation of merge sort – ascending order



- What types of objects can you sort? Anything that has an **ordering**
- Two sort() methods: sort a given List according to either 1) *natural ordering* of elements or an 2) externally defined ordering.

1)
```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

2)
```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

- Translation:
  1. Only accepts a List parameterised with type implementing Comparable
  2. Accepts a List parameterised with any type as long as you also give it a Comparator implementation that defines the ordering for that type

java.lang.**Comparable<T>**

A **generic interface** with a single method: **`int compareTo(T)`**

Return 0 if this = other

Return **any +'ve integer** if this > other

Return **any –'ve integer** if this < other

Implement this interface to define **natural ordering** on objects of type T

```
public class Money implements Comparable<Money> {
  ...
  public int compareTo( Money other ) {
    if( this.cents == other.cents ) {
      return 0;
    }
    else if( this.cents < other.cents ) {
      return -1;
    }
    else {
      return 1;
    }
  }
```

m1 = new Money(100,0);
m2 = new Money(50,0);
m1.compareTo(m2) returns 1;

A more concise way of doing this? (hint: 1 line)

```
return this.cents – other.cents;
```

Natural-order sorting

```
List<Money> funds = new ArrayList<Money>();
funds.add(new Money(100,0));
funds.add(new Money(5,50));
funds.add(new Money(-40,0));
funds.add(new Money(5,50));
funds.add(new Money(30,0));

Collections.sort(funds);
System.out.println(funds);

List<CD> albums = new ArrayList<CD>();
albums.add(new CD("Street Signs","Ozomatli",2.80));
//etc...
Collections.sort(albums);
```

What's the output?
[-40.0,
5.50,
5.50,
30.0,
100.0]

CD does not implement a Comparable interface

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

java.util.**Comparator<T>**

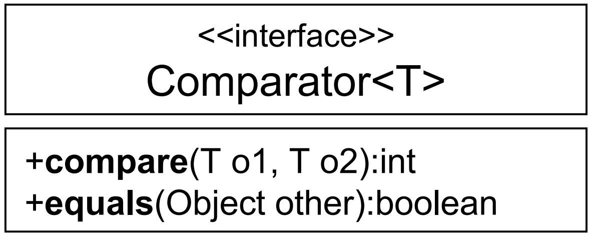
Useful if the type of elements to be sorted is not Comparable, or you want to define an alternative ordering

Also a generic interface that defines methods **`compare(T,T)`** and **`equals(Object)`**

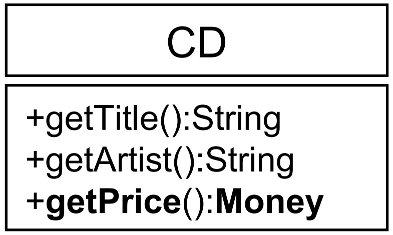
Usually only need to define **`compare(T,T)`**

Define ordering by CD's getPrice() → **Money**

Note: PriceComparator implements a Comparator para-meterised with **`CD`** → **`T`** "becomes" **`CD`**



<<interface>>
Comparator<T>
+**compare**(T o1, T o2):int
+**equals**(Object other):boolean



```
public class PriceComparator
implements Comparator<CD> {
  public int compare(CD c1, CD c2) {
    return c1.getPrice().compareTo(c2.getPrice());
  }
}
```

Comparator and Comparable going hand in hand ☺

Comparator sorting

```
List<CD> albums = new ArrayList<CD>();
albums.add(new CD("Street Signs","Ozomatli",new Money(3,50)));
albums.add(new CD("Jazzinho","Jazzinho",new Money(2,80)));
albums.add(new CD("Space Cowboy","Jamiroquai",new Money(5,00)));
albums.add(new CD("Maiden Voyage","Herbie Hancock",new Money(4,00)));
albums.add(new CD("Here's the Deal","Liquid Soul",new Money(1,00)));

Collections.sort(albums, new PriceComparator());
System.out.println(albums);
```

```
implements Comparator<CD>
```

Note, in sort(), Comparator overrides natural ordering

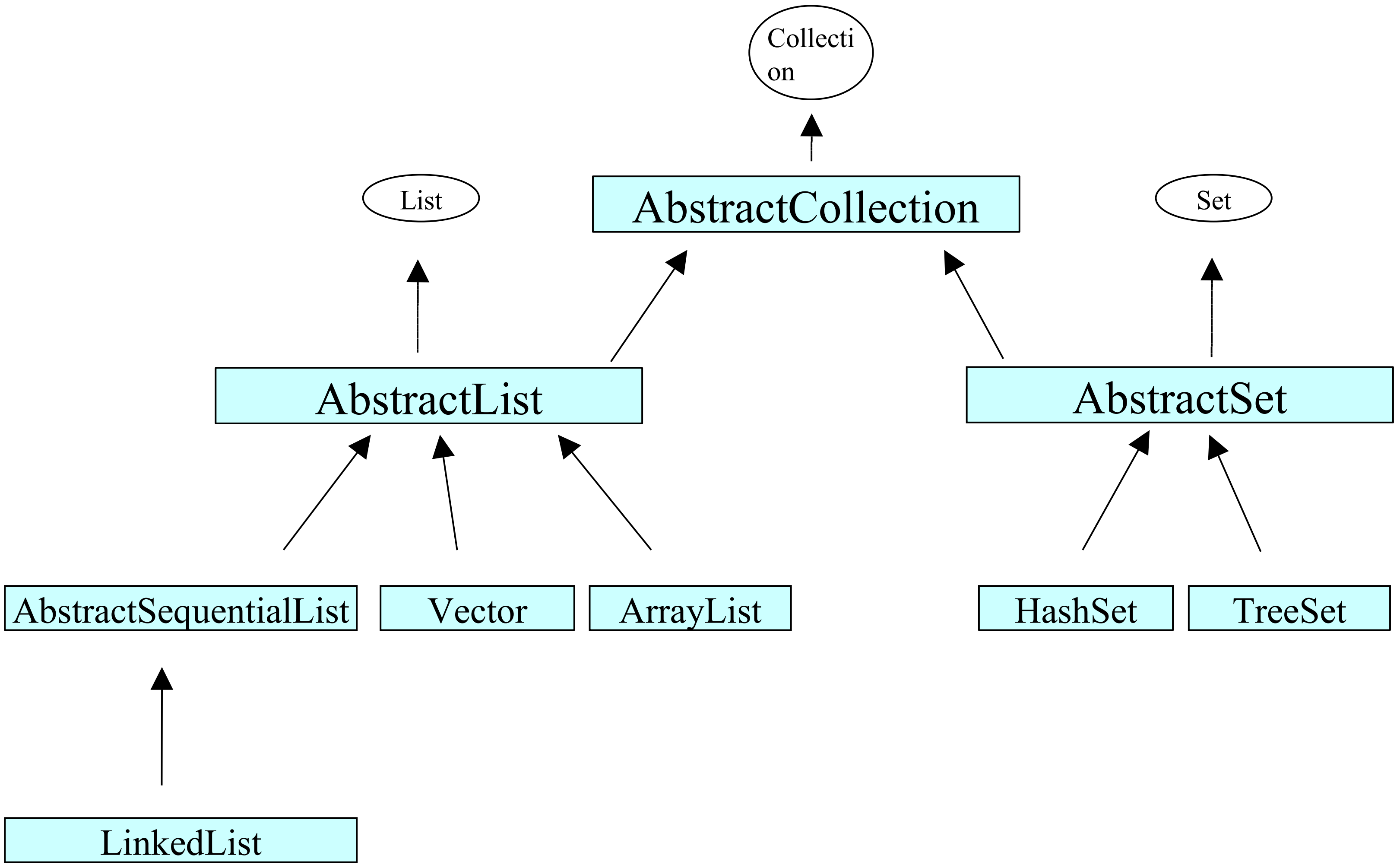
i.e. Even if we define natural ordering for CD, the given comparator is still going to be used instead

(On the other hand, if you give `null` as Comparator, then natural ordering is used)

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

The Class Structure

- The Collection interface is implemented by a class called AbstractCollection. Most collections inherit from this class.



Lists

- Java provides 3 concrete classes which implement the list interface
  - Vector
  - ArrayList
  - LinkedList
- Vectors try to optimize storage requirements by growing and shrinking as required
  - Methods are synchronized (used for Multi threading)
- ArrayList is roughly equivalent to Vector except that its methods are not synchronized
- LinkedList implements a doubly linked list of elements
  - Methods are not synchronized

Sets

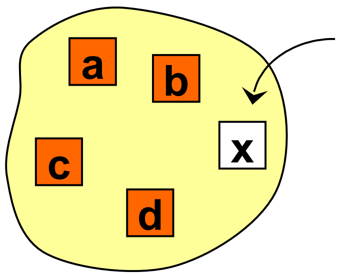
- Java provides 2 concrete classes which implement the Set interface
  - HashSet
  - TreeSet
- HashSet behaves like a HashMap except that the elements cannot be duplicated.
- TreeSet behaves like TreeMap except that the elements cannot be duplicated.
- Note: Sets are not as commonly used as Lists

Set<E>

Mathematical Set abstraction – contains **no duplicate** elements
i.e. no two elements e1 and e2 such that e1.equals(e2)

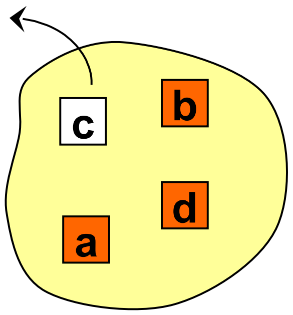
add(x)
→*true*

add(b)
→*false*



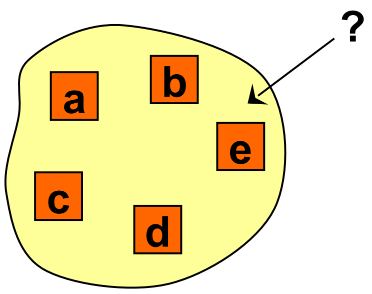
remove(c)
→*true*

remove(x)
→*false*



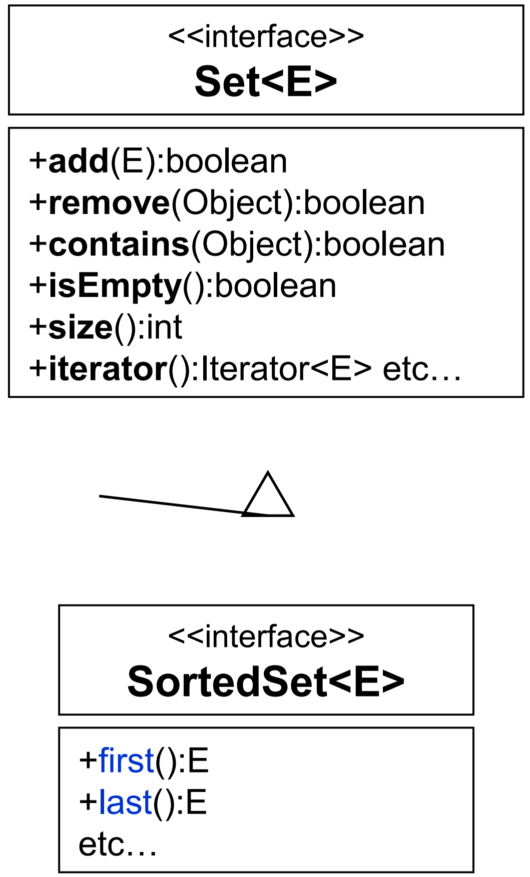
contains(e)
→true

contains(x)
→*false*



isEmpty()
→*false*

size()
→*5*



<<interface>>
Set<E>

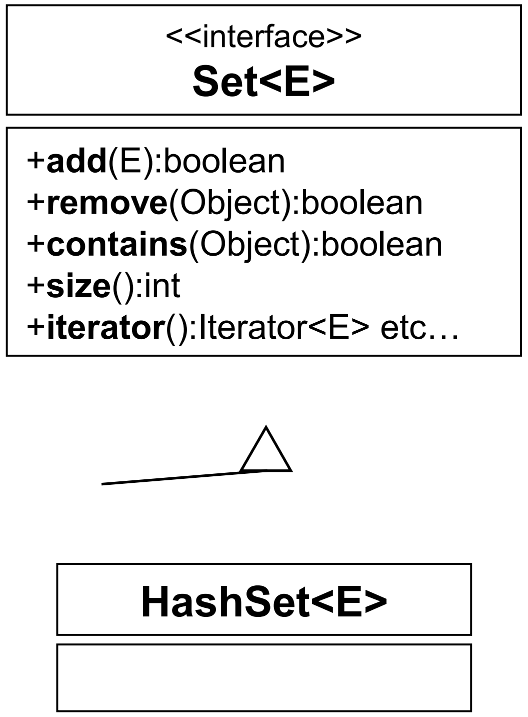
+**add**(E):boolean
+**remove**(Object):boolean
+**contains**(Object):boolean
+**isEmpty**():boolean
+**size**():int
+**iterator**():Iterator<E> etc…

<<interface>>
SortedSet<E>

+first():E
+last():E
etc…

HashSet<E>

- Typically used implementation of Set.
- Parameterise Sets just as you parameterise Lists
- Efficient (constant time) insert, removal and contains check – all done through hashing
- x and y are duplicates if x.equals(y)
- How are elements ordered? Quiz:



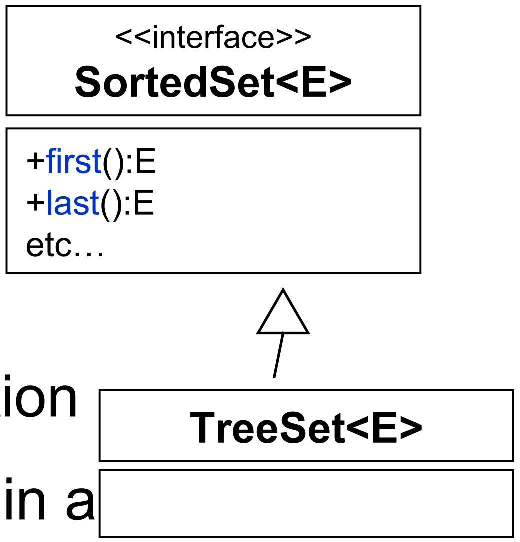
```
Set<String> words = new HashSet<String>();
words.add("Bats");
words.add("Ants");
words.add("Crabs");
words.add("Ants");
System.out.println(words.size());
for (String word : words) {
   System.out.println(word);
}
```

3

?

a) Bats, Ants, Crabs
b) Ants, Bats, Crabs
c) Crabs, Bats, Ants
d) Nondeterministic

TreeSet<E> (SortedSet<E>)



- If you want an ordered set, use an implementation of a SortedSet: TreeSet
- What's up with "Tree"? Red-black tree
- Guarantees that all elements are ordered (sorted) at all times
  - » **`add()`** and **`remove()`** preserve this condition
  - » **`iterator()`** always returns the elements in a specified order
- Two ways of specifying ordering
  - » Ensuring elements have natural ordering (**`Comparable`**)
  - » Giving a **`Comparator<E>`** to the constructor
- **Caution:** TreeSet considers x and y are duplicates if x.compareTo(y) == 0 (or compare(x,y) == 0)

TreeSet construction

```
Set<String> words = new TreeSet<String>();
words.add("Bats");
words.add("Ants");
words.add("Crabs");
for (String word : words) {
   System.out.println(word);
}
```

String has a **natural ordering**, so empty constructor

What's the output?

```
Ants; Bats; Crabs
```

- But CD doesn't, so you must pass in a Comparator to the constructor

```
Set<CD> albums = new TreeSet<CD>(new PriceComparator());
albums.add(new CD("Street Signs","O",new Money(3,50)));
albums.add(new CD("Jazzinho","J",new Money(2,80)));
albums.add(new CD("Space Cowboy","J",new Money(5,00)));
albums.add(new CD("Maiden Voyage","HH",new Money(4,00)));
albums.add(new CD("Here's the Deal","LS",new Money(2,80)));
System.out.println(albums.size());
for (CD album : albums) {
   System.out.println(album);
}
```

What's the output?

```
4
Jazzinho; Street; Maiden; Space
```

The Map Interface

- The Map interface provides the basis for dictionary or key-based collections in Java. The interface includes:

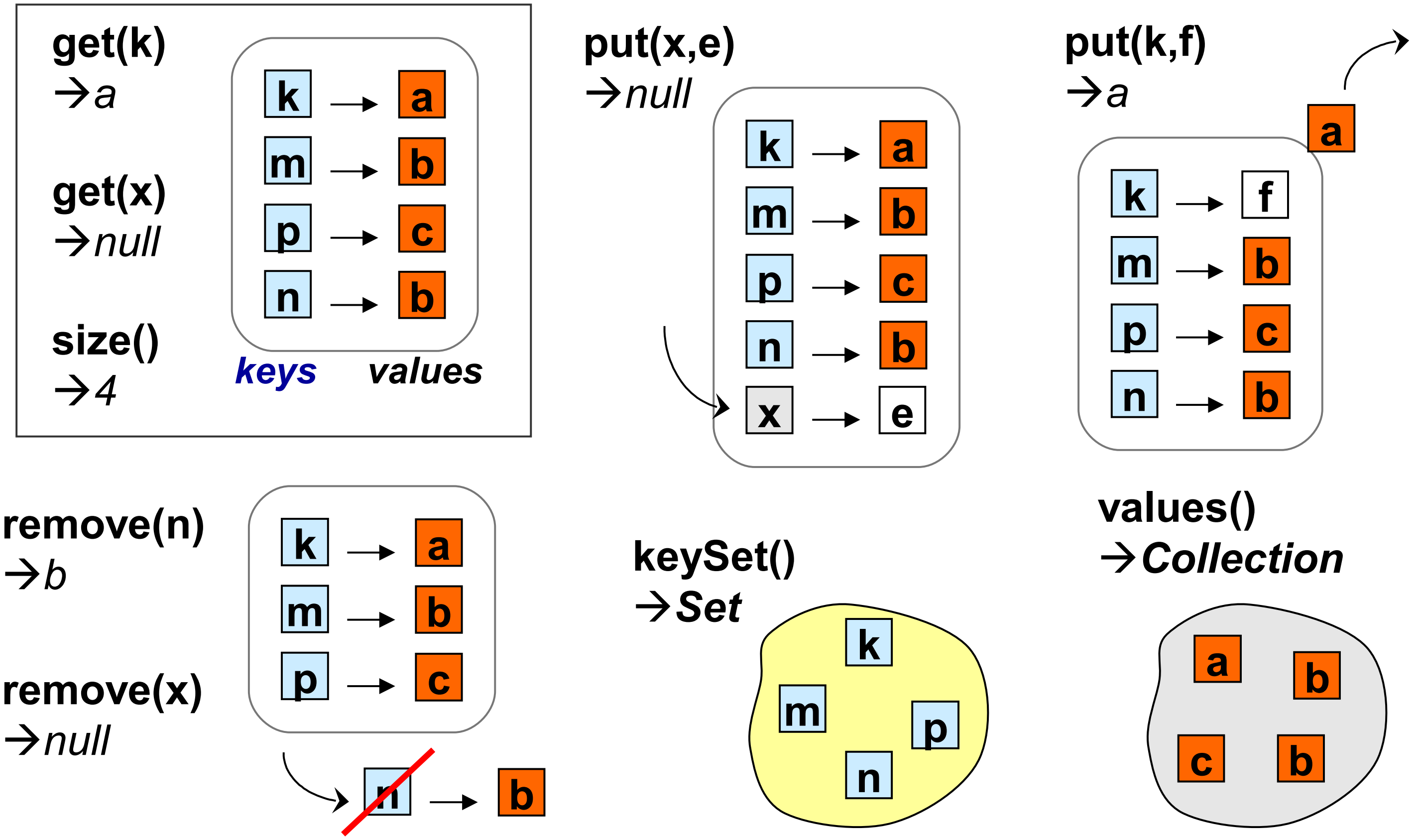
```
void clear()
boolean containsKey(Object)
boolean containsValue(Object)
Set entrySet()
boolean equals(Object)
Object get(Object)
boolean isEmpty()
Set keySet()
Object put(Object key, Object value)
void putAll(Map)
boolean remove(Object key)
int size()
Collection values()
```

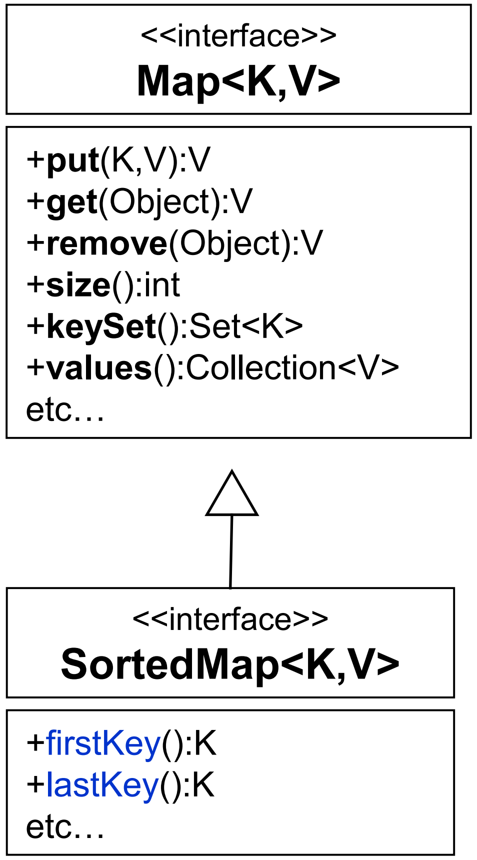
Maps

- Java provides 3 concrete classes which implement the map interface
  - HashMap
  - WeakHashMap
  - TreeMap
- HashMap is the most commonly used Map.
  - Provides access to elements through a key.
  - The keys can be iterated if they are not known.
- WeakHashMap provides the same functionality as Map except that if the key object is no longer used, the key and it's value will be removed from the Map.
- A Red-Black implementation of the Map interface

`Map<K,V>`

- Stores mappings from (unique) keys (type `K`) to values (type `V`)
  - » See, you can have more than one type parameters!
- Think of them as “arrays” but with objects (keys) as indexes
  - » Or as “directories”: e.g. `"Bob"` → `021999887`





HashMap<K,V>

- keys are hashed using **`Object.hashCode()`**
  - » i.e. no guaranteed ordering of keys
- **`keySet()`** returns a **`HashSet`**
- **`values()`** returns a Collection

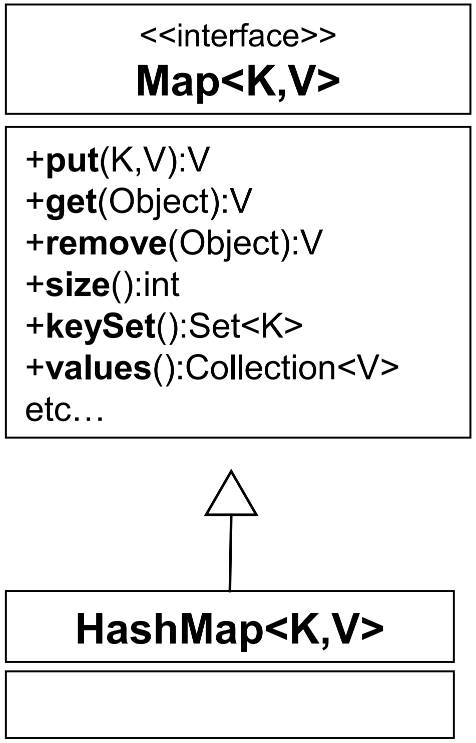
```
Map<String, Integer> directory
                  = new HashMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
  System.out.print(key+"'s number: ");
  System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

"autoboxing"

4 or 5?

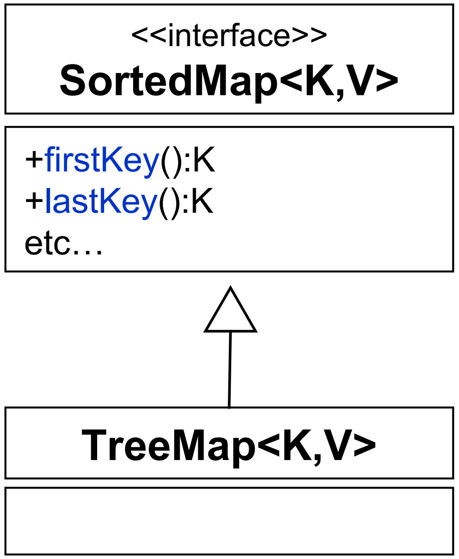
`Set<String>`

What's Bob's number?



TreeMap<K,V>

- Guaranteed ordering of keys (like TreeSet)
  - » In fact, TreeSet is implemented using TreeMap ☺
  - » Hence **`keySet()`** returns a **`TreeSet`**
- **`values()`** returns a Collection – ordering depends on ordering of **keys**



Empty constructor
→ natural ordering

```
Map<String, Integer> directory
                  = new TreeMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
  System.out.print(key+"'s #: ");
  System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

4

Loop output?

```
Bob's #: 1000000
Dad's #: 9998888
Edward's #: 5553535
Mum's #: 9998888
```

?

TreeMap with Comparator

As with TreeSet, another way of constructing TreeMap is to give a Comparator → necessary for non-Comparable keys

```
Map<CD, Double> ratings
              = new TreeMap<CD, Double>(new PriceComparator());
ratings.put(new CD("Street Signs","O",new Money(3,50)), 8.5);
ratings.put(new CD("Jazzinho","J",new Money(2,80)), 8.0);
ratings.put(new CD("Space Cowboy","J",new Money(5,00)), 9.0);
ratings.put(new CD("Maiden Voyage","H",new Money(4,00)), 9.5);
ratings.put(new CD("Here's the Deal","LS",new Money(2,80)), 9.0);

System.out.println(ratings.size());
for (CD key : ratings.keySet()) {
  System.out.print("Rating for "+key+": ");
  System.out.println(ratings.get(key));
}
System.out.println("Ratings: "+ratings.values());
```

4

Ordered by key's price

Depends on key ordering

Most Commonly Use Methods

- While it is a good idea to learn and understand all of the methods defined within this infrastructure, here are some of the most commonly used methods.
- For Lists:
  - add(Object), add(index, Object)
  - get(index)
  - set(index, Object)
  - remove(Object)
- For Maps:
  - put(Object key, Object value)
  - get(Object key)
  - remove(Object key)
  - keySet()

Which class should I use?

- You'll notice that collection classes all provide the same or similar functionality. The difference between the different classes is how the structure is implemented.
  - This generally has an impact on performance.
- Use Vector
  - Fast access to elements using index
  - Optimized for storage space
  - Not optimized for inserts and deletes
- Use ArrayList
  - Same as Vector except the methods are not synchronized. Better performance
- Use linked list
  - Fast inserts and deletes
  - Stacks and Queues (accessing elements near the beginning or end)
  - Not optimized for random access

Which class should I use?

- Use Sets
  - When you need a collection which does not allow duplicate entries
- Use Maps
  - Very Fast access to elements using keys
  - Fast addition and removal of elements
  - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

Collections and Fundamental Data Types

- Note that collections can only hold Objects.
  - One cannot put a fundamental data type into a Collection
- Java has defined "wrapper" classes which hold fundamental data type values within an Object
  - These classes are defined in java.lang
  - Each fundamental data type is represented by a wrapper class
- The wrapper classes are:

  Boolean
  Byte
  Character
  Double
  Float
  Short
  Integer
  Long

Wrapper Classes

- The wrapper classes are usually used so that fundamental data values can be placed within a collection
- The wrapper classes have useful class variables.
  - Integer.MAX_VALUE, Integer.MIN_VALUE
  - Double.MAX_VALUE, Double.MIN_VALUE, Double.NaN, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY
- They also have useful class methods

  Double.parseDouble(String) - converts a String to a double

  Integer.parseInt(String) - converts a String to an integer

Advanced Programming Methods

Lecture 3 - Java IO Operations

Overview

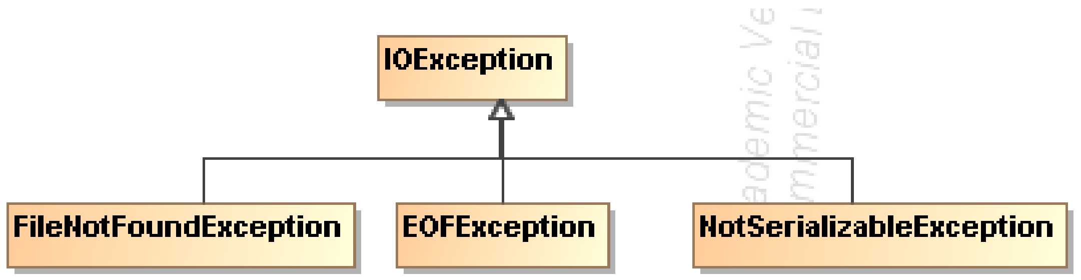
1. Java IO
2. Java NIO
3. Try-with-resources

Java IO

Java.IO

- Package java.io
  - classes working on bytes (InputStream, OutputStream)
  - Classes working on chars (Reader, Writer)
  - Byte-char conversion (InputStreamReader, OutputStreamWriter)
  - Random access (RandomAccessFile)
  - Scanner

- Exceptions:

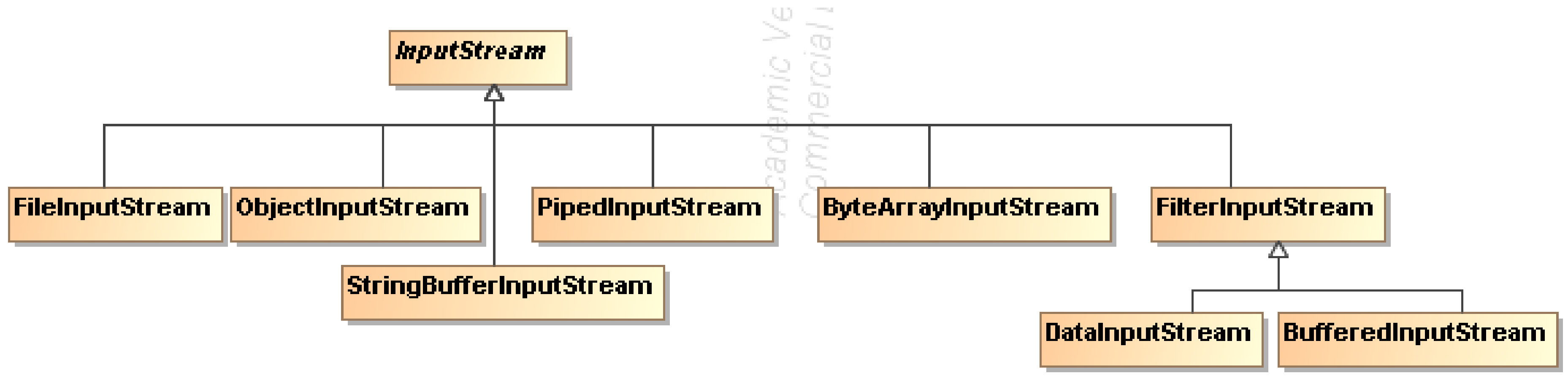


IO Stream

- represents an input source or an output destination
- is a sequence of data
- supports many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- simply passes on data; others manipulate and transform the data in useful ways.

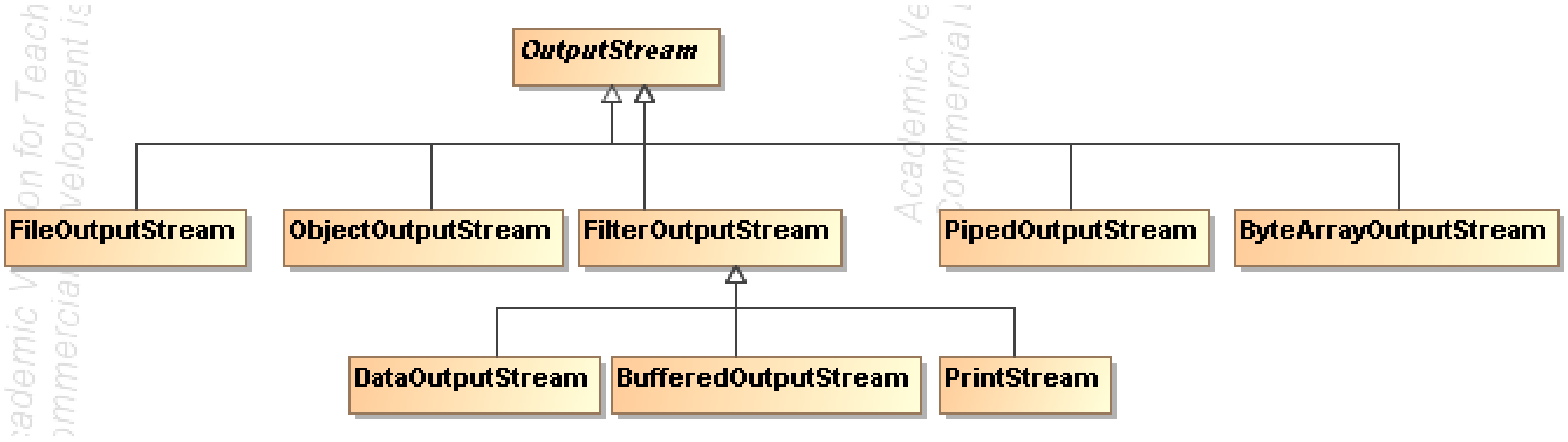
InputStream

- Abstract class that contains methods for reading bytes from a stream (file, memory, pipe, etc.)
  - `read():int` //read a byte, return –1 if no more bytes
  - `read(cbuff:byte[]): int` //read max `cbuff.length bytes`, return the nr of bytes that has been read, or -1
  - `read(buff:byte[], offset:int, length:int):int` //read max `length` bytes and write to `buff` starting with position `offset`, return the number of bytes that has been read, or -1
  - `available(): int` //number of bytes available for reading
  - `close()` //close the stream



OutputStream

- Abstract class that contains methods for writing bytes into a stream (file, memory, pipe, etc.)
  - `write(int)` //write a byte
  - `write(b:byte[])` //write `b.length` bytes from array `b` into the stream
  - `write(b:byte[], offset:int, len:int):int` //write `len` bytes from array `b` starting with the position `offset`
  - `flush()` // force the effective writing into the stream
  - `close()` //close the stream

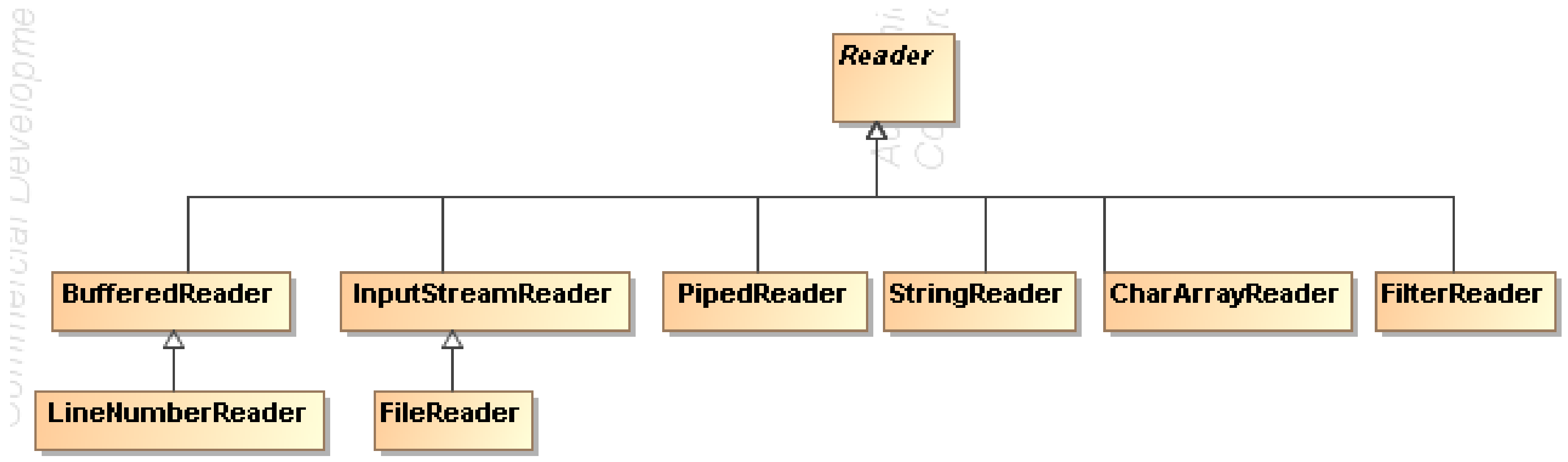


Example

```
 FileInputStream in = null;
FileOutputStream out = null;
try {
   in = new FileInputStream(“fisier.txt");
   out = new FileOutputStream(“fisier2.txt");
   int c;
   while ((c = in.read()) != -1) {
          out.write(c);
   }
 } catch(IOException e){
    System.err.println(“Eroare ”+e);
 }finally {
   if (in != null)
      try {
          in.close();
      } catch (IOException e){ System.err.println("eroare "+e);}
   if (out != null)
       try {
           out.close();
       } catch (IOException e) { System.err.println("eroare "+e);}
}
```

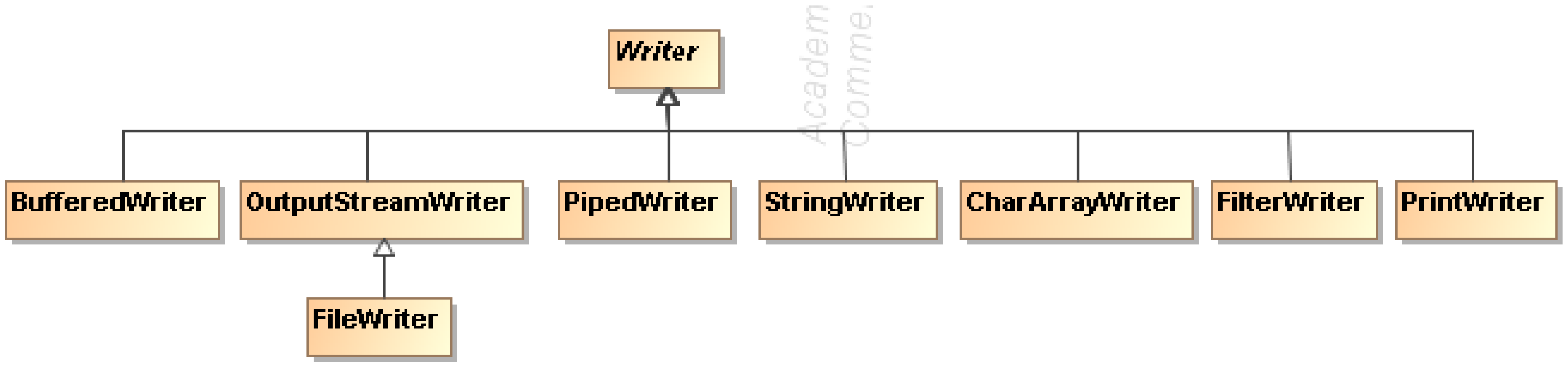
Reader

- Abstract class that contains methods for chars reading (1 char = 2 bytes) from a stream (file, memory, pipe, etc.)
  - `read():int` //read a char, return –1 for the end of the stream
  - `read(cbuff:char[]): int` //read max `cbuff.length` chars, return nr of read chars or -1
  - `read(buff:char[], offset:int, length:int):int` //read max `length` chars into array `buff` starting with offset `offset`, return the number of read chars or -1
  - `close()` //close the stream



Writer

- Abstract class that contains the methods for writing chars into a stream
  - `write(int)` //write a char
  - `write(b:char[])` //write `b.length` chars from array `b` into the stream
  - `write(b:char[], offset:int, len:int):int` //write `len` chars from array `b` starting with offset
  - `write(s:String)` //write a `String`
  - `write(s:String, off:int, len:int)` //write a part of a `String`
  - `flush()` // force the effective writing
  - `close()` //close the stream



Example

```
FileReader input = null;
FileWriter output = null;
try {
   input = new FileReader("Fisier.txt");
   output = new FileWriter(“Fisier2out.txt");
   int c;
   while ((c = input.read()) != -1) output.write(c);
} catch (IOException e) {
   System.err.println("Eroare la citire/scriere"+e);
} finally {
    if (input != null)
        try {
           input.close();
        } catch (IOException e) {System.err.println("eroare "+e);}
    if (output != null)
         try {
             output.close();
         } catch (IOException e) {System.err.println("Eroare "+e);}
}
```

Classes

Operations	**Byte**	**Char**
Files	FileInputStream, FileOutputStream	FileReader, FileWriter
Memory	ByteArrayInputStream, ByteArrayOutputStream	CharArrayReader CharArrayWriter
Buffered Operations	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Format	PrintStream	PrintWriter
Conversion Byte ↔ Char	InputStreamReader (byte -> char) OutputStreamWriter (char -> byte)	

Example

- Read a list of students (from a file, from keyboard)
- Saving the list of students in ascending order based on their average (into a file)

//Studenti.txt

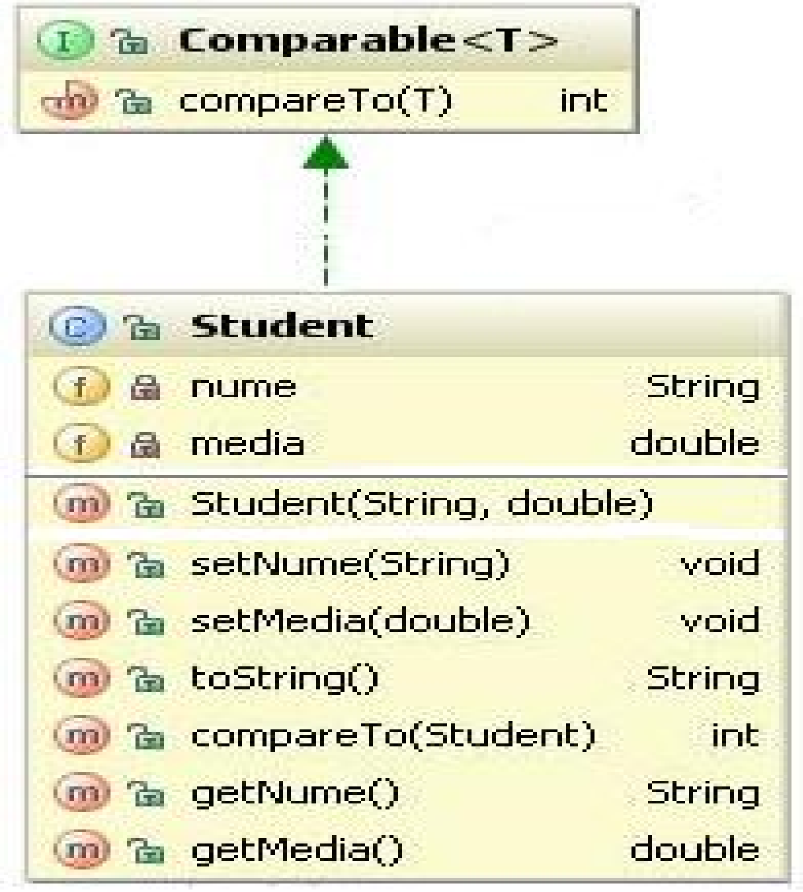
Vasilescu Maria|8.9

Popescu Ion|6.7

Marinescu Ana|9.6

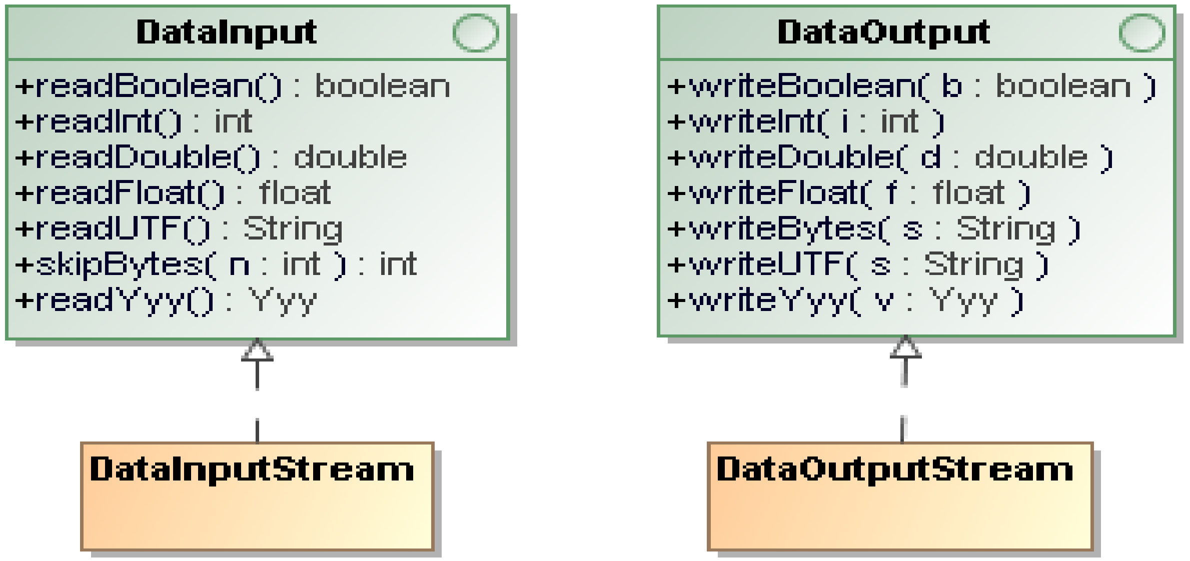
Ionescu George|7.53

Pop Vasile|9.3



Writing and reading data of primitive types

- Interfaces `DataInput` and `DataOutput`



- `Yyy` can be primitive types (`byte, short, char,` ...): `readByte(), readShort(), readChar(), writeByte(byte), writeShort(short), ...`

Obs:

1. In order to read data of primitive types using methods `readYyy,` the data must be saved before using the methods `writeYyy.`
2. When there are no more data it throws the exception `EOFException.`

Example DataOutput

```
 void printStudentiDataOutput(List<Student> studs, String numefis){
        DataOutputStream output=null;
        try{
            output=new DataOutputStream(new FileOutputStream(numefis));
            for(Student stud: studs){
                output.writeUTF(stud.getNume());
                output.writeDouble(stud.getMedia());
            }
        } catch (FileNotFoundException e) {
            System.err.println("Eroare scriere DO "+e);
        } catch (IOException e) {
            System.err.println("Eroare scriere DO "+e);
        }finally {
            if (output!=null)
                try {
                    output.close();
                } catch (IOException e) {
                    System.err.println("Eroare scriere DO "+e);
                }
        }
}
```

Example DataInput

```
List<Student> citesteStudentiDataInput(String numefis){
   List<Student> studs=new ArrayList<Student>();
   DataInputStream input=null;
   try{
      input=new DataInputStream(new FileInputStream(numefis));
      while(true){
          String nume=input.readUTF();
          double media=input.readDouble();
          studs.add(new Student(nume,media));
      }
   }catch(EOFException e){ }
   catch (FileNotFoundException e) { System.err.println("Eroare citire"+e);}
   catch (IOException e) { System.err.println("Eroare citire DI "+e);}
   finally {
       if (input!=null)
         try { input.close();}
         catch (IOException e) {
                   System.err.println("Eroare inchidere fisier "+e);
          }}
   return studs;
}
```

Standard streams

- `System.in` of type `InputStream`
- `System.out` of type `PrintStream`
- `System.err` of type `PrintStream`

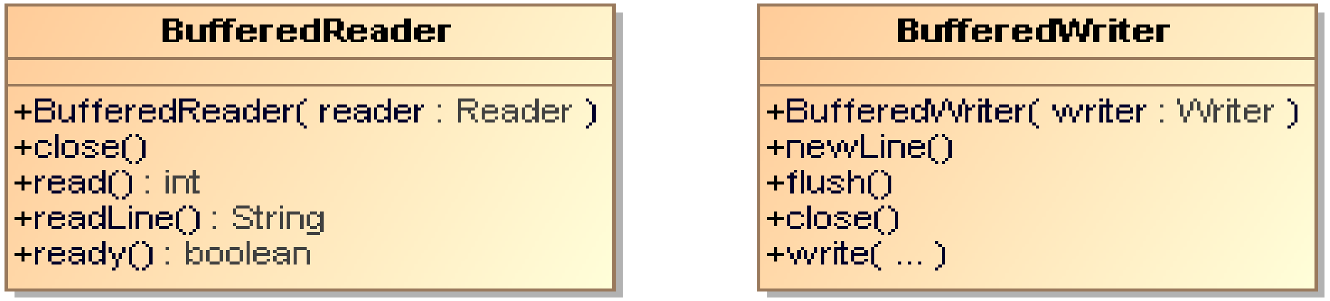
The associated streams can be modified using the methods:

`System.setIn(), System.setOut(), System.setErr(),`

```
Example:
System.setOut(new PrintStream(new File("Output.txt")));
System.setErr(new PrintStream(new File("Erori.txt")));
```

BufferedReader/BufferedWriter

- Use a buffer to keep the data which are going to be read/write from/to a stream.
- Read/Write operations are more efficient since the reading/writing is effectively done only when the buffer is empty/full.



```
BufferedReader br=new BufferedReader(new FileReader(numefisier));
BufferedWriter bw=new BufferedWriter(new FileWriter(numefisier));
// rewrite the existing data in the file

//add at the end of the file
BufferedWriter bw=new BufferedWriter(new FileWriter(numefisier, true));
```

Example BufferedReader

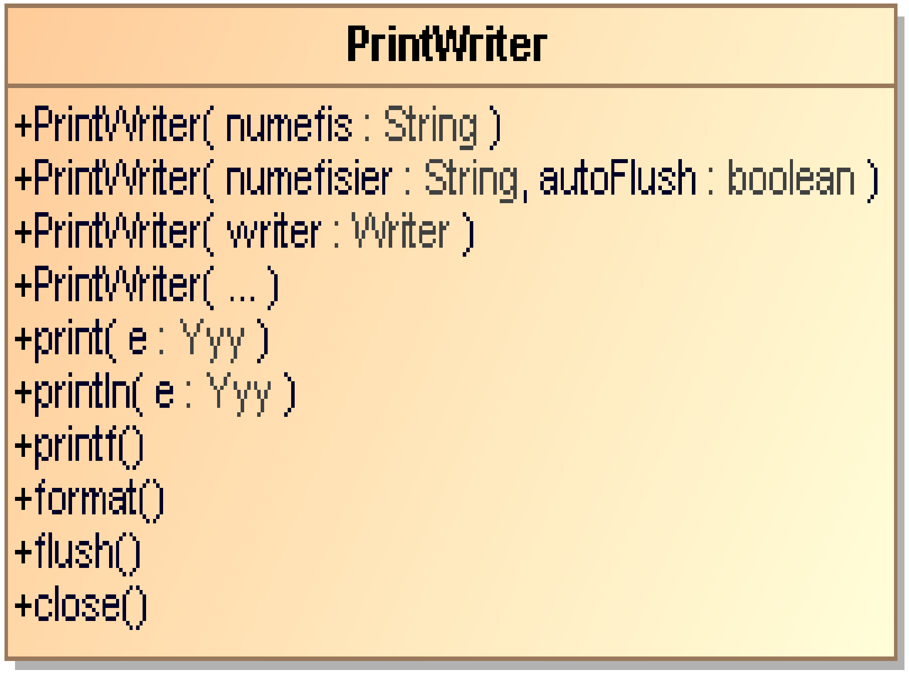
```
List<Student> citesteStudenti(String numefis){
   List<Student> ls=new ArrayList<Student>();
   BufferedReader br=null;
   try{
       br=new BufferedReader(new FileReader(numefis));
       String linie;
       while((linie=br.readLine())!=null){
           String[] elems=linie.split("[|]");
           if (elems.length<2){
               System.err.println("Linie invalida "+linie);
               continue;}
         Student stud=new Student(elems[0], Double.parseDouble(elems[1]));
         ls.add(stud);
         }
    }catch (FileNotFoundException e) {System.err.println("Eroare citire "+e);}
     catch (IOException e) { System.err.println("Eroare citire "+e);}
     finally{
        if (br!=null)
            try { br.close();}
            catch (IOException e) { System.err.println("Eroare inchidere fisier:
  "+e); }
    }
   return ls;
}
```

Example BufferedWriter

```
void printStudentiBW(List<Student> studs, String numefis){
        BufferedWriter bw=null;
        try{
            bw=new BufferedWriter(new FileWriter(numefis));
            //bw=new BufferedWriter(new FileWriter(numefis,true));
            for(Student stud: studs){
                bw.write(stud.getNume()+'|'+stud.getMedia());
                bw.newLine();    //scrie sfarsitul de linie
            }
        } catch (IOException e) {
            System.err.println("Eroare scriere BW "+e);
        } finally {
            if (bw!=null)
                try {
                    bw.close();
                } catch (IOException e) {
                    System.err.println("Eroare inchidere fisier "+e);
                }
        }
    }
```

PrintWriter

- Contains methods to save any type of data in text format.
- Contains methods to format the data .


PrintWriter
+PrintWriter(numefis : String)
+PrintWriter(numefisier : String, autoFlush : boolean)
+PrintWriter(writer : Writer)
+PrintWriter(...)
+print(e : Yyy)
+println(e : Yyy)
+printf()
+format()
+flush()
+close()

- `Yyy` is any primitive or reference type. If `Yyy` is a reference type it is called the method `toString` corresponding to `e`.

Example 1 PrintWriter

```
void printStudentiPrintWriter(List<Student> studs, String numefis){
        PrintWriter pw=null;
        try{
            pw=new PrintWriter(numefis);
            for(Student stud: studs){
                pw.println(stud.getNume()+'|'+stud.getMedia());
            }

        } catch (FileNotFoundException e) {
            System.err.println("Eroare scriere PW "+e);
        }finally {
            if (pw!=null)
                pw.close();
        }
}
```

Example 2 PrintWriter

```
void printStudentiPWTabel(List<Student> studs, String numefis){
   PrintWriter pw=null;
   try{
      pw=new PrintWriter(numefis);
      String linie=getLinie('-',48);
      int crt=0;
      for(Student stud:studs){
         pw.println(linie);
//pw.printf("| %3d | %-30s | %5.2f |%n",(++crt),stud.getNume(),stud.getMedia());
  pw.format("| %3d | %-30s | %5.2f |%n",(++crt),stud.getNume(),stud.getMedia());
       }
       if (crt>0)
          pw.println(linie);
    } catch (FileNotFoundException e) {
          System.err.println("Eroare scriere PWTabel "+e);
    } finally {
          if (pw!=null)
              pw.close();
    }
 }
```

Example 2 PrintWriter

```
String getLinie(char c, int length){
        char[] tmp=new char[length];
        Arrays.fill(tmp,c);
        return String.valueOf(tmp);
}
//file result
------------------------------------------------
|   1 | Popescu Ion                    |  6.70 |
------------------------------------------------
|   2 | Ionescu George                 |  7.53 |
------------------------------------------------
|   3 | Vasilescu Maria                |  8.90 |
------------------------------------------------
|   4 | Pop Vasile                     |  9.30 |
------------------------------------------------
|   5 | Marinescu Ana                  |  9.60 |
------------------------------------------------
```

Reading from the keyboard

- Class `BufferedReader`

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

- Class `Scanner` (package `java.util)`

  ```
  Scanner input=new Scanner(System.in);
  ```

- Class `Scanner` contains methods to read data of primitive types from the keyboard (or other stream):
  - `nextInt():int`
  - `nextDouble():double`
  - `nextFloat():Float`
  - `nextLine():String`
  - `...`
  - `hasNextInt():boolean`
  - `hasNextDouble():boolean`
  - `hasNextFloat():boolean`
  - `...`

Example BufferedReader (keyboard)

```
List<Student> citesteStudenti(){   //from the keyboard
  List<Student> ls=new ArrayList<Student>();
  BufferedReader br=null;
  try{
    br=new BufferedReader(new InputStreamReader(System.in));
    System.out.println("La terminare introduceti cuvantul \"gata\"");
    boolean gata=false;
    while(!gata){
            System.out.println("Introduceti numele: ");
            String snume=br.readLine();
            if ("gata".equalsIgnoreCase(snume)){gata=true; continue;}
            System.out.println("Introduceti media: ");
            String smedia=br.readLine();
            if ("gata".equalsIgnoreCase(smedia)){gata=true; continue;}
            try{
               double media=Double.parseDouble(smedia);
               lista.add(new Student(snume,media));
            }catch(NumberFormatException nfe){
               System.err.println("Eroare: "+nfe);
            }
     } }/*catch, finally, ...*/ }//citesteStudenti
```

Example Scanner (keyboard)

```
List<Student> citesteStudentiScanner(){
  List<Student> lista=new ArrayList<Student>();
  Scanner scanner=null;
  try{
    scanner=new Scanner(System.in);
   System.out.println("La terminare introduceti cuvantul \"gata\"");
    boolean gata=false;
    while(!gata){
       System.out.println("Introduceti numele: ");
       String snume=scanner.nextLine();
       if ("gata".equalsIgnoreCase(snume)){gata=true; continue;  }
       System.out.println("Introduceti media: ");
       if (scanner.hasNextDouble()) {
           double media=scanner.nextDouble();
           lista.add(new Student(snume,media));
            scanner.nextLine();  //to read <Enter>
            continue;
        }else{
        //next slide
```

Example Scanner (keyboard) cont.

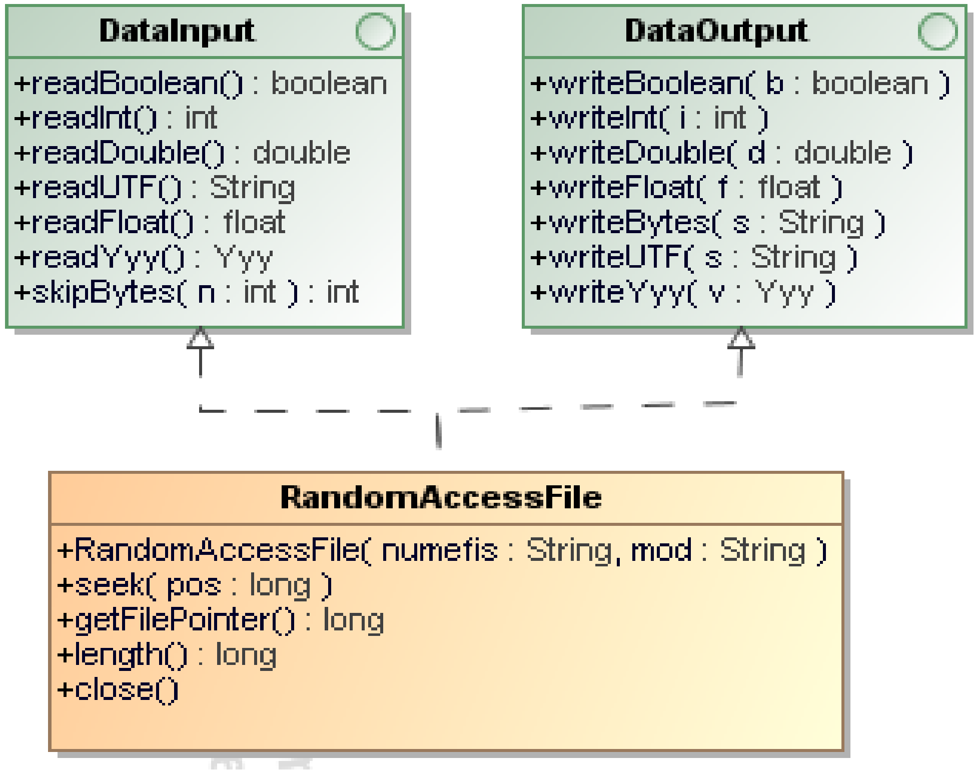
```
List<Student> citesteStudentiScanner(){
  //...
  if (scanner.hasNextDouble()){
    //...
  } else{
        String msj=scanner.nextLine();
        if ("gata".equalsIgnoreCase(msj)){
                        gata=true;
                        continue;
         }else
            System.out.println("Trebuie sa introduceti media studentului");
    }//else
   }//while
 } finally {
    if (scanner!=null)
         scanner.close();
 }
 return lista;
}
```

Example Scanner file

```
Scanner inscan=null;
try{
  inscan=new Scanner(new BufferedReader(new FileReader("intregi.txt")));
  //inscan.useDelimiter(‟,");
  while(inscan.hasNextInt()){
     int nr=inscan.nextInt();
     System.out.println("nr = " + nr);
  }
} catch (FileNotFoundException e) {
    System.err.println("Eroare "+e);
}finally {
   if (inscan!=null)
    inscan.close();
}
```

RandomAccessFile

- Allow random access to a file.
- Can be used for either reading or writing.
- Class uses the notion of cursor to denote the current position in the file. Initially the cursor is on the position 0 at the beginning of the file.
- Operations of reading/writing move the cursor according the number of bytes read/written.



Example writing RandomAccessFile

```
void printStudentiRAF(List<Student> studs, String numefis){
  RandomAccessFile out=null;
  try{
    out=new RandomAccessFile(numefis,"rw");
    for(Student stud: studs){
      out.writeUTF(stud.getNume());
      out.writeDouble(stud.getMedia());
    }
  }catch (FileNotFoundException e){System.err.println("Eroare RAF "+e);}
  catch (IOException e) { System.err.println("Eroare scriere RAF "+e);}
  finally{
     if (out!=null)
        try {
           out.close();
        } catch (IOException e) {
            System.err.println("Eroare inchidere fisier "+e);
        }
    }
 }
```

Example reading RandomAccessFile

```
List<Student> citesteStudentiRAF(String numefis){
  List<Student> studs=new ArrayList<Student>();
  RandomAccessFile in=null;
  try{
            in=new RandomAccessFile(numefis, "r");
            while(true){
                String nume=in.readUTF();
                double media=in.readDouble();
                studs.add(new Student(nume,media));
            }
  }catch(EOFException e){ }
  catch (FileNotFoundException e){System.err.println("Eroare la citire: "+e);}
  catch (IOException e) { System.err.println("Eroare la citire "+e);}
  finally {
      if (in!=null)
         try {
             in.close();
         } catch (IOException e) {System.err.println("Eroare RAF "+e);}
}
 return studs;     }
```

Example appending RandomAccessFile

```
void adaugaStudent(Student stud, String numefis){
        RandomAccessFile out=null;
        try{
            out=new RandomAccessFile(numefis, "rw");
            out.seek(out.length());
            out.writeUTF(stud.getNume());
            out.writeDouble(stud.getMedia());
        } catch (FileNotFoundException e) {
            System.err.println("Eroare RAF "+e);
        } catch (IOException e) {
            System.err.println("Eroare RAF "+e);
        }finally {
            if (out!=null)
                try {
                    out.close();
                } catch (IOException e) {
                    System.err.println("Eroare RAF "+e);
                }
        }
    }
```

Class File

- Represent the name of a file (not its content).
- Allow platform-independent operations on the files (create, delete, rename, etc.) .
  - `File(name:String) //name  is the path to a file or a directory`
  - `getName():String`
  - `getAbsolutePath():String`
  - `isFile():boolean`
  - `isDirectory():boolean`
  - `exists():boolean`
  - `delete():boolean`
  - `deleteOnExit()    //Directory/File is removed at the exit of JVM`
  - `mkdir()`
  - `list():String[]`
  - `list(filtru:FilenameFilter):String[]`
  - ...

Class File: examples

- Printing the current directory

```
File dirCurent=new File(".");
System.out.println("Directory:" + dirCurent.getAbsolutePath());
```

- Creating an OS-independent path

```
 String namefis=".."+File.separator+"data"+File.separator+"intregi.txt";
File f1=new File(namefis);
System.out.println("F1 "+f1.getName());
System.out.println("Exists f1? "+f1.exists());
```

- Selecting the .txt files from a directory

```
 File dir=new File(".");
String[] files=dir.list(new FilenameFilter(){
   public boolean accept(File dir, String name) {
      return name.toLowerCase().endsWith(".txt");
}  });
 System.out.println("Files "+ Arrays.toString(files));
```

- Removing a file

```
File namef=new File("erori.txt");
if (namef.exists())
   boolean ok=namef.delete();
```

Java NIO

Java NIO (New IO)

- From Java 1.4
- is an alternative IO API for Java (to the standard Java IO and Java Networking API's)
- consist of the following core components:
    - Channels
    - Buffers
    - Selectors

Channels and Buffers

- In the standard IO API you work with byte streams and character streams.
- In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.
- all IO in NIO starts with a Channel.

Channels and Buffers

- Channels are similar to streams with a few differences:

  - You can both read and write to a Channels. Streams are typically one-way (read or write).
  - Channels can be read and written asynchronously.
  - Channels always read to, or write from, a Buffer.

Channels and Buffers

the most important Channel implementations in Java NIO:

- FileChannel: reads data from and to files.
- DatagramChannel: can read and write data over the network via UDP.
- SocketChannel: can read and write data over the network via TCP.
- ServerSocketChannel: allows you to listen for incoming TCP connections, like a web server does.

Channels and Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again.
- Buffer types let you work with the bytes in the buffer as char, short, int, long, float or double:
  - ByteBuffer
  - MappedByteBuffer
  - CharBuffer
  - DoubleBuffer
  - FloatBuffer
  - IntBuffer
  - LongBuffer
  - ShortBuffer

Channels and Buffers

- Using a Buffer to read and write data typically follows this little 4-step process:
  - Write data into the Buffer: The buffer keeps track of how much data you have written.
  - Call buffer.flip(): in order to switch the buffer from writing mode into reading mode
  - Read data out of the Buffer: In reading mode the buffer lets you read all the data written into the buffer.
  - Call buffer.clear() or buffer.compact(): to make buffer ready for writing again

Channels and Buffers

- The clear() method clears the whole buffer.
- The compact() method only clears the data which you have already read. Any unread data is moved to the beginning of the buffer, and data will now be written into the buffer after the unread data.

Channels and Buffers

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();

//create buffer with capacity of 48 bytes
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); //read into buffer.
while (bytesRead != -1) {
  buf.flip();  //make buffer ready for read
  while(buf.hasRemaining()){
      System.out.print((char) buf.get()); // read 1 byte at a time
  }
  buf.clear(); //make buffer ready for writing
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

Channels and Buffers

- A Buffer has three properties:
  1. Capacity: a certain fixed size. Once the Buffer is full, you need to empty it (read the data, or clear it) before you can write more data into it.
  2. Position:
     - Write mode:Initially the position is 0. When a byte, long etc. has been written into the Buffer the position is advanced to point to the next cell in the buffer to insert data into. Position can maximally become capacity – 1
     - Read mode:When you flip a Buffer from writing mode to reading mode, the position is reset back to 0. As you read data from the Buffer you do so from position, and position is advanced to next position to read.

Channels and Buffers

3. Limit:

- Write mode: is the limit of how much data you can write into the buffer and it is equal to the capacity of the Buffer
- Read mode: is the limit of how much data you can read from the data. Therefore, when flipping a Buffer into read mode, limit is set to write position of the write mode. In other words, you can read as many bytes as were written (limit is set to the number of bytes written, which is marked by position).

Scattering Reads

- reads data from a single channel into multiple buffers

```
ByteBuffer buf1 = ByteBuffer.allocate(128);
ByteBuffer buf2   = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { buf1,buf2  };
channel.read(bufferArray);
```

Gathering Writes

- writes data from multiple buffers into a single channel

```
ByteBuffer buf1 = ByteBuffer.allocate(128);
ByteBuffer buf2   = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { buf1,buf2  };
channel.write(bufferArray);
```

Java NIO FileChannel

- writes data from multiple buffers into a single channel
- is a channel that is connected to a file.
- you can read data from a file, and write data to a file.
- is an alternative to reading files with the standard Java IO API.

Channel to Channel Transfer

- you can transfer data directly from one channel to another

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count    = fromChannel.size();

toChannel.transferFrom(fromChannel, position, count);
```

Java NIO Path

- an interface that is similar to the java.io.File class
- An instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

Java NIO Files

- provides several methods for manipulating files in the file system

Java NIO AsynchronousFileChannel

- makes it possible to read data from, and write data to files asynchronously
- Read and write operations can be done either via a Future or via a CompletionHandler

Reading via a Future

```
AsynchronousFileChannel fileChannel =
                AsynchronousFileChannel.open(path, StandardOpenOption.READ);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

Future<Integer> operation = fileChannel.read(buffer, position);

while(!operation.isDone());

buffer.flip();
byte[] data = new byte[buffer.limit()];
buffer.get(data);
System.out.println(new String(data));
buffer.clear();
```

Writing via a CompletionHandler

```
Path path = Paths.get("data/test-write.txt");

if(!Files.exists(path)){

    Files.createFile(path);}

AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);

long position = 0;

buffer.put("test data".getBytes());

buffer.flip();

fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {

    @Override

    public void completed(Integer result, ByteBuffer attachment) {

        System.out.println("bytes written: " + result);}

    @Override

    public void failed(Throwable exc, ByteBuffer attachment) {

        System.out.println("Write failed");

        exc.printStackTrace();}

});
```

Selectors

- A Selector allows a single thread to handle multiple Channel's.
- is handy if your application has many Channels open
- To use a Selector you register the Channel's with it. Then you call it's select() method. This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events. Examples of events are incoming connection, data received etc.

Try-with-resources

Old Style

```
private static void printFile() throws IOException {
    InputStream input = null;
    try {
        input = new FileInputStream("file.txt");
        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    } finally {
        if(input != null){
            input.close();
        }
    }
}
```

- The red marked code may throw exceptions

Try-with-resources

- From Java 7 the previous code can be rewritten as follows:

```
private static void printFileJava7() throws IOException {

    try(FileInputStream input = new FileInputStream("file.txt")) {

        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

Try-with-resources

- When the try block finishes the FileInputStream will be closed automatically. This is possible because FileInputStream implements the Java interface java.lang.AutoCloseable.

```
public interface AutoClosable {

    public void close() throws Exception;
}
```

- All classes implementing this interface can be used inside the try-with-resources construct.

Advanced Programming Methods

Lecture 4 - Functional Programming in Java

Important Announcement:

At Seminar 6
(7-13 November 2017)
you will have a closed-book test
(based on your laboratory work).

Overview

1. Anonymous inner classes in Java
2. Lambda expressions in Java 8
3. Processing Data with Java 8 Streams

Note: Lecture notes are based on Oracle tutorials.

Anonymous Inner classes

- provide a way to implement classes that may occur only once in an application.

```
JButton testButton = new JButton("Test Button");
    testButton.addActionListener(new ActionListener(){
          @Override public void actionPerformed(ActionEvent ae){
               System.out.println("Click Detected by Anon Class");
          }
    });
```

Functional Interfaces

- are interfaces with only one method
- Using functional interfaces with anonymous inner classes are a common pattern in Java

```
public interface ActionListener extends EventListener {
   public void actionPerformed(ActionEvent e);
}
```

Lambda Expressions

- are Java's first step into functional programming
- can be created without belonging to any class
- can be passed around as if they were objects and executed on demand.

```
(int x, int y) -> x + y


() -> 42


(String s) -> { System.out.println(s); }


testButton.addActionListener(e -> System.out.println("Click Detected by
    Lambda Listner"));
```

Lambda Expressions

- Lambda function body

```
(oldState, newState) -> System.out.println("State changed")


(oldState, newState) -> {
    System.out.println("Old state: " + oldState);
    System.out.println("New state: " + newState);
  }
```

- Returning a value

```
(param) -> {System.out.println("param: " + param; return "return value";}


 (a1, a2) -> { return a1 > a2; }


 (a1, a2) -> a1 > a2;
```

Lambdas as Objects

- A Java lambda expression is essentially an object.
- You can assign a lambda expression to a variable and pass it around, like you do with any other object.

```
public interface MyComparator {

    public boolean compare(int a1, int a2);

}


MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);
```

Runnable Lambda

```
// Anonymous Runnable
Runnable r1 = new Runnable(){
@Override
public void run(){ System.out.println("Hello world one!"); } };

// Lambda Runnable
 Runnable r2 = () -> System.out.println("Hello world two!");

// Run em!
 r1.run();
 r2.run();
```

Comparator Lambda

```
List<Person> personList = Person.createShortList();
    // Sort with Inner Class
    Collections.sort(personList, new Comparator<Person>(){
      public int compare(Person p1, Person p2){
        return p1.getSurName().compareTo(p2.getSurName());
      }});


    // Use Lambda instead
  Collections.sort(personList, (Person p1, Person p2) →
    p1.getSurName().compareTo(p2.getSurName()));
  Collections.sort(personList, (p1,  p2) ->
    p2.getSurName().compareTo(p1.getSurName()));
```

Lambda Expressions

- can improve your code
- provide a means to better support the Don't Repeat Yourself (DRY) principle
- make your code simpler and more readable.
- Motivational example: Given a list of people, various criteria are used to send messages to matching persons:
  - Drivers(persons over the age of 16) get phone calls
  - Draftees(male persons between the ages of 18 and 25) get emails
  - Pilots(persons between the ages of 23 and 65) get mails

First Attempt

```
public class RoboContactMethods {
 public void callDrivers(List<Person> pl){
    for(Person p:pl){
        if (p.getAge() >= 16){ roboCall(p);}
    } }
 public void emailDraftees(List<Person> pl){
    for(Person p:pl){
       if (p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE){
         roboEmail(p);
     } }}
 public void mailPilots(List<Person> pl){
     for(Person p:pl){
       if (p.getAge() >= 23 && p.getAge() <= 65){         roboMail(p);  }
    } }
…...}
```

First Attempt

- The DRY principle is not followed.
  - Each method repeats a looping mechanism.
  - The selection criteria must be rewritten for each method
- A large number of methods are required to implement each use case.
- The code is inflexible. If the search criteria changed, it would require a number of code changes for an update. Thus, the code is not very maintainable.

Second Attempt

```
public class RoboContactMethods2 {
   public void callDrivers(List<Person> pl){
      for(Person p:pl){
         if (isDriver(p)){  roboCall(p);}}}
  public void emailDraftees(List<Person> pl){
       for(Person p:pl){
        if (isDraftee(p)){       roboEmail(p);}}}
 public void mailPilots(List<Person> pl){
     for(Person p:pl){
       if (isPilot(p)){ roboMail(p);}} }
  public boolean isDriver(Person p){ return p.getAge() >= 16; }
   public boolean isDraftee(Person p){
       return p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE; }
   public boolean isPilot(Person p){  return p.getAge() >= 23 && p.getAge() <= 65; }
```

Third Attempt

- Using a functional interface and anonymous inner classes

```
public interface Predicate<T> {
   public boolean test(T t);
}


public void phoneContacts(List<Person> pl, Predicate<Person> aTest){
    for(Person p:pl){
       if (aTest.test(p)){    roboCall(p); }
} }


robo.phoneContacts(pl,  new Predicate<Person>(){
           @Override
            public boolean test(Person p){
               return p.getAge() >=16;  } }  );
```

Fourth Attempt

- Using lambda expressions

```
public void phoneContacts(List<Person> pl, Predicate<Person> pred){
   for(Person p:pl){
      if (pred.test(p)){    roboCall(p); }
} }


Predicate<Person> allDrivers = p -> p.getAge() >= 16;
 Predicate<Person> allDraftees = p -> p.getAge() >= 18 && p.getAge() <= 25 &&
     p.getGender() == Gender.MALE;
 Predicate<Person> allPilots = p -> p.getAge() >= 23 && p.getAge() <= 65;


robo.phoneContacts(pl, allDrivers);
```

java.util.function

- standard interfaces are designed as a starter set for developers:
  - Predicate: A property of the object passed as argument
  - Consumer: An action to be performed with the object passed as argument
  - Function: Transform a T to a U
  - Supplier: Provide an instance of a T (such as a factory)
  - UnaryOperator: A unary operator from T -> T
  - BinaryOperator: A binary operator from (T, T) -> T

Function Interface

- It has only one method apply with the following signature:

  public R apply(T t)

- Example for class Person:

```
  public String printCustom(Function <Person, String> f){

      return f.apply(this);}
```

```
Function<Person, String> westernStyle = p -> {return "\nName: " +
    p.getGivenName() + " " + p.getSurName() + "\n"};

Function<Person, String> easternStyle =  p -> "\nName: " + p.getSurName() + " " +
    p.getGivenName() + "\n"};

person.printCustom(westernStyle);

person.printCustom(easternStyle);

person.printCustom(p -> "Name: " + p.getGivenName() + " EMail: " + p.getEmail());
```

Java 8 Streams

- is a new addition to the Java Collections API, which brings a new way to process collections of objects.
- declarative way
- Stream: a sequence of elements from a source that supports aggregate operations.

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
myList.stream()
 .filter(s -> s.startsWith("c"))
 .map(String::toUpperCase)
 .sorted()
 .forEach(System.out::println);
```

- Output:

```
C1
C2
```

Java 8 Streams

- Sequence of elements: A stream provides an interface to a sequenced set of values of a specific element type. However, streams don’t actually store elements; they are computed on demand.

- Source: Streams consume from a data-providing source such as collections, arrays, or I/O resources.

- Aggregate operations: Streams support SQL-like operations and common operations from functional programing languages, such as filter, map, reduce, find, match, sorted, and so on.

Streams vs Collections

Two fundamental characteristics that make stream operations very different from collection operations:

- Pipelining: Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as laziness and short-circuiting
- Internal iteration: In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you.

Streams vs Collections

- collections are about data
- **streams are about computations**.
- A collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.
- In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.

Obtaining a Stream From a Collection

```
List<String> items = new ArrayList<String>();


items.add("one");
items.add("two");
items.add("three");


Stream<String> stream = items.stream();
```

- is similar to how you obtain an Iterator by calling the items.iterator() method, but a Stream is different than an Iterator.

Stream Processing Phases

1. **Configuration**-- intermediate operations:
    - filters, mappings
    - can be connected together to form a pipeline
    - return a stream
    - Are lazy: do not perform any processing
2. **Processing**—terminal operations:
    - operations that close a stream pipeline
    - produce a result from a pipeline such as a List, an Integer, or even void (any non-Stream type).

Filtering

- **stream.filter(item -> item.startsWith("o"));**

- filter(Predicate): Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
- distinct: Returns a stream with unique elements (according to the implementation of equals for a stream element)
- limit(n): Returns a stream that is no longer than the given size n
- skip(n): Returns a stream with the first n number of elements discarded

Filtering

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

List<Integer> twoEvenSquares =

    numbers.stream()

           .filter(n -> {System.out.println("filtering " + n); return n % 2 == 0;})

           .map(n -> { System.out.println("mapping " + n); return n * n;})

           .limit(2)

           .collect(toList());
```

```
filtering 1

filtering 2

mapping 2

filtering 3

filtering 4

mapping 4
```

- limit(2) uses short-circuiting; we need to process only part of the stream, not all of it, to return a result.

Mapping

- Streams support the method map, which takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form. The function is applied to each element, “mapping” it into a new element.

```
items.stream()
    .map( item -> item.toUpperCase() )
```

- maps all strings in the items collection to their uppercase equivalents.

NOTE: it doesn't actually perform the mapping. It only configures the stream for mapping. Once one of the stream processing methods are invoked, the mapping (and filtering) will be performed.

Mapping

```
List<String> words = Arrays.asList("Oracle", "Java",
    "Magazine");
 List<Integer> wordLengths =
    words.stream()
         .map(String::length)
         .collect(toList());
```

Stream.collect()

- is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map .

- Collect accepts a Collector which consists of four different operations: a *supplier*, an *accumulator*, a *combiner* and a *finisher*.

- Java 8 supports various builtin collectors via the Collectors class. So for the most common operations you don't have to implement a collector yourself.

Stream.collect()

```
List<Person> filtered =
persons
 .stream()
 .filter(p -> p.name.startsWith("P"))
 .collect(Collectors.toList());



Double averageAge =
persons
 .stream()
 .collect(Collectors.averagingInt(p -> p.age));
```

Stream.collect()

```
String phrase =
persons
 .stream()
 .filter(p -> p.age >= 18)
 .map(p -> p.name)
 .collect(Collectors.joining(" and ", "In Germany ", " are of legal
    age."));
```

- The join collector accepts a delimiter as well as an optional prefix and suffix.

Stream.collect()

- In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped.
- the mapped keys must be unique, otherwise an IllegalStateException is thrown.
- You can optionally pass a merge function as an additional parameter to bypass the exception:

```
Map<Integer, String> map = persons
.stream()
.collect(Collectors.toMap(
  p -> p.age,
  p -> p.name,
  (name1, name2) -> name1 + ";" + name2));
```

Stream.min() and Stream.max()

- Are terminal operations
- return an Optional instance which has a get() method on, which you use to obtain the value. In case the stream has no elements the get() method will return null
- take a Comparator as parameter. The Comparator.comparing() method creates a Comparator based on the lambda expression passed to it. In fact, the comparing() method takes a Function which is a functional interface suited for lambda expressions

```
String shortest = items.stream()

        .min(Comparator.comparing(item -> item.length()))

        .get();
```

Stream.min() and Stream.max()

- The Optional<T> class (java.util .Optional) is a container class to represent the existence or absence of a value
- we can choose to apply an operation on the optional object by using the ifPresent method

```
Stream.of("a1", "a2", "a3")
 .map(s -> s.substring(1))
 .mapToInt(Integer::parseInt)
 .max()
 .ifPresent(System.out::println);
```

- Stream.of() creates a stream from a bunch of object references

Stream.count()

- Returns the number of elements in the stream

```
long count = items.stream()
    .filter( item -> item.startsWith("t"))
    .count();
```

Stream.reduce()

- can reduce the elements of a stream to a single value
- takes a BinaryOperator as parameter, which can easily be implemented using a lambda expression.
- Returns an Optional
- The BinaryOperator.apply() method:
  - takes two parameters. The acc which is the accumulated value, and item which is an element from the stream.

```
String reduced2 = items.stream()
      .reduce((acc, item) -> acc + " " + item)
      .get();
```

Stream.reduce()

- There is another reduce() method which takes two parameters: an initial value for the accumulated value, and then a BinaryOperator.

```
String reduced = items.stream()
    .filter( item -> item.startsWith("o"))
    .reduce("", (acc, item) -> acc + " " + item);
```

Stream.reduce()

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
int sum = numbers.stream().reduce(0, (a, b) -> a + b);

int product = numbers.stream().reduce(1, (a, b) -> a * b);
int max = numbers.stream().reduce(1, Integer::max);
```

Numerical Streams

- IntStream, DoubleStream, and LongStream—that respectively specialize the elements of a stream to be int, double, and long.
- to convert a stream to a specialized version: mapToInt, mapToDouble, and mapToLong.
- to help generate ranges: range and rangeClosed.

```
IntStream oddNumbers =
    IntStream.rangeClosed(10, 30)
        .filter(n -> n % 2 == 1);
```

Building Streams

- InStream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);

- int[] numbers = {1, 2, 3, 4};

  IntStream numbersFromArray = Arrays.stream(numbers);

- Converting a file into a stream of lines:

  ```
  long numberOfLines =
   Files.lines(Paths.get("yourFile.txt"),Charset.defaultCharset())
       .count();
  ```

Infinite Streams

- There are two static methods—Stream.iterate and Stream .generate—that let you create a stream from a function.
- because elements are calculated on demand, these two operations can produce elements “forever.”

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);
```

- The iterate method takes an initial value (here, 0) and a lambda (of type UnaryOperator<T>) to apply successively on each new value produced.

Infinite Streams

- We can turn an infinite stream into a fixed-size stream using the limit operation:

```
numbers.limit(5).forEach(System.out::println);
// 0, 10, 20, 30, 40.
```

Finding and Matching

- A common data processing pattern is determining whether some elements match a given property. You can use the **anyMatch, allMatch, and noneMatch** operations to help you do this. They all take a predicate as an argument and return a boolean as the result (they are, therefore, terminal operations)

- Stream interface provides the operations **findFirst and findAny** for retrieving arbitrary elements from a stream. Both findFirst and findAny return an Optional object

Processing Order

```
Stream.of("d2", "a2", "b1", "b3", "c")
 .map(s -> {System.out.println("map: " + s);return s.toUpperCase();})
 .filter(s -> {System.out.println("filter: " + s);return s.startsWith("A");})
 .forEach(s -> System.out.println("forEach: " + s));
```

```
// map: d2
// filter: D2
// map: a2
// filter: A2
// forEach: A2
// map: b1
// filter: B1
// map: b3
// filter: B3
// map: c
// filter: C
```

Processing Order

```
Stream.of("d2", "a2", "b1", "b3", "c")
 .filter(s -> {System.out.println("filter: " + s);return s.startsWith("a");})
 .map(s -> {System.out.println("map: " + s);return s.toUpperCase();})
 .forEach(s -> System.out.println("forEach: " + s));


// filter: d2

// filter: a2

// map: a2

// forEach: A2

// filter: b1

// filter: b3

// filter: c
```

Reusing Streams

- Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed

```
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true); // ok

stream.noneMatch(s -> true);
 // exception since stream has been consumed
```

Reusing Streams

```
Supplier<Stream<String>> streamSupplier =
() -> Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> s.startsWith("a"));



streamSupplier.get().anyMatch(s -> true); // ok
streamSupplier.get().noneMatch(s -> true); // ok
```

- *Each call to get() constructs a new stream on which we can call the desired terminal operation.*