Șiruri

Există trei clase referitoare la şiruri: `String`, `StringBuffer` si `StringBuilder`. Diferenţa constă dans faptul că obiectele de tipul String sunt **imutabile**, iar cele de tip StringBuffer sau `StringBuilder` sunt mutabile (pot fi modificate).
Clasa `StringBuffer` este sigura in aplicatiile concurente in timp ce `StringBuilder` nu. In general vom utiliza `StringBuilder` in aplicatii cu un singur Thread.
Comparativ cu `StringBuffer` clasa `StringBuilder` va fi utilizata cu precadere datorita vitezei mai mari.

Clasa String

Orice constantă şir este transformată într-o instanţă a clasei `String`, fiind echivalentă cu `new String(constanta_sir)`.

```
String s=”Exemplu”;
```

În această declaraţie este construit implicit un obiect String, fără a fi nevoie de a apela `new String(”Exemplu”)`.

Operaţii
- Operatorii + şi += realizează concatenarea şirurilor
- `int length()` – numărul de caractere din şir
- `char charAt(i)` – caracterul cu indicele i din sir
- `int indexOf(c)` – indicele caracterului c
- `String substring(i,j)` – returnează subşirul care începe cu caracterul de indice `i` al şirului şi se termină cu carcaterul de indice `j-1`
- `String replace(char c1, char c2)` – returnează un şir în care c1 este înlocuit cu c2
- `valueOf()` realizează conversia unei valori într-un şir

Exemplu

```
boolean p = true;
int n = 10;
double x = 6.023;
 String sp = String.valueOf(p); //”true”
 String sn = String.valueOf(n); //”10”
 String sx = String.valueOf(x); //”6.023”
```

Compararea a două şiruri poate fi efectuată cu una din metodele
- `boolean` **`equals()`**
- `boolean equalsIgnoreCase()`
- `int compareTo()` similar cu `strcmp()` din biblioteca C.

Atenţie: operatorul == verifică egalitatea referinţelor nu şi a conţinutului

Exemplu

```
class siruri {
  public static void main(String [] args)
  {
    String s1="ABCdefgh", so=s1;

    System.out.println("String s1=\""+ s1 +"\", so=s1");
```

```
    System.out.println("s1.length :"+ s1.length() );
    System.out.println("s1.charAt(1) :"+ s1.charAt(1) );
    System.out.println("s1.substring(1,3) :"+
                        s1.substring(1,3) );
    System.out.println("s1.indexOf('d') :"+
                        s1.indexOf('d') );
    System.out.println("\nInlocuire s1.replace('A','a') -> \""+
s1.replace('A','a') +"\"");
    System.out.println("s1=s1.toUpperCase() : "+
                       (s1=s1.toUpperCase()) );
// comparatii
    String s2="ABCDEFGH";
    System.out.println("\nString s1=\""+ s1+", s2=\""+s2+"\"" );
    System.out.println("Comparatii:\ns1==s2 :"+(s1==s2));
    System.out.println("s1.equals(s2) :"+ s1.equals(s2));
    System.out.println("s1.compareTo(s2) :"+
                        s1.compareTo(s2) );
    System.out.println("s1.equalsIgnoreCase(s2) :"+
                        s1.equalsIgnoreCase(s2) );
    System.out.println("Comparatii:\ns1==so :"+ (s1==so) + "( so=\""+so+"\"
)" );
// referinta
    String sx=s1;
    System.out.println("\nComparatii referinta:\ns1==sx :"+ (s1==sx) );
    System.out.println("s1.equals(sx) :"+ s1.equals(sx));
  }
}
```

Programul produce urmatoarea iesire.

```
String s1="ABCdefgh", so=s1
s1.length :8
s1.charAt(1) :B
s1.substring(1,3) :BC
s1.indexOf('d') :3

Inlocuire s1.replace('A','a') -> "aBCdefgh"
s1=s1.toUpperCase() : ABCDEFGH

String s1="ABCDEFGH, s2="ABCDEFGH"
Comparatii:
s1==s2 :false
s1.equals(s2) :true
s1.compareTo(s2) :0
s1.equalsIgnoreCase(s2) :true
Comparatii:
s1==so :false( so="ABCdefgh" )

Comparatii referinta:
s1==sx :true
s1.equals(sx) :true
```

Se observă că `s1==s2` are valoarea `false`, deoarece reprezintă referinţe distincte, în schimb `s1.equals(s2)` este `true` deoarece conţinutul este identic.
Mai trebuie observat şi că după atribuirea

```
s1=s1.toUpperCase()
```

referinţa `s1` pierde vechea valoare (memorată în `so`) către obiectul şir iniţial `("ABCdefgh")` primind valoarea referinţei unui nou şir obţinut din cel iniţial prin transformarea tuturor literelor în majuscule.
În clasa `String` mai şi există metodele `toCharArray()`, `getChars()` şi `getBytes()` care transformă un şir într-un tablou de caractere, respectiv octeţi.

Clasa StringBuffer

Principalele metode definitorii pentru această clasă şi care nu se regăsesc în clasa `String` sunt următoarele

- `s1.append(s2)` – s2 este adăugat la s1; există mai multe supradefiniri ale funcţiei pentru `Object, String, char[]` (în cazul lui `char[]` se precizează offset şi lungime), `boolean, char, int, long, float, double`)
- `s.insert(i,valoare)` – începând cu indicele `i` se inserează în şirul `s` `valoare` (care poate fi `Object, String, char[], boolean, char, int, long, float, double`)
- `s.reverse()` – inversează ordinea
- `s.setCharAt(i,c)` – stabileşte caracterul aflat la indicele `i` în şirul `s` ca fiind `c`

Exemplu cu clasa StringBuffer

```
class sirbuf {
  public static void main(String [] args)
  {
    StringBuffer s=new StringBuffer();
    StringBuffer s1=new StringBuffer("Caci unde-ajunge nu-i hotar,-
\tNici");
    String s2="\n\tDin goluri a se naste.";

    s1.setCharAt(28 , '\n');  // unde este '-'
    s1.append(" ochi spre a cunoaste,\n ");
    System.out.println("StringBuffer s1=\n\""+ s1 +"\"");

    // adaugare la un sir vid
    s.append(s1).append("Si vremea-ncearca in zadar, ").append(s2);
    System.out.println("\nStringBuffer s=\n\""+ s +"\"");

    StringBuffer c= new StringBuffer("c= km/s");
    c.insert(2, 300000.);
    System.out.println("\nViteza luminii incidente : \""+ c +"\"");
    c.reverse();
    System.out.println(  "Viteza luminii reflectate: \""+ c +"\"");


 }
}
```

In urma executiei programul produce iesirea urmatoare:

```
StringBuffer s1=
"Caci unde-ajunge nu-i hotar,
        Nici ochi spre a cunoaste,
 "

StringBuffer s=
"Caci unde-ajunge nu-i hotar,
        Nici ochi spre a cunoaste,
 Si vremea-ncearca in zadar,
        Din goluri a se naste."

Viteza luminii incidente : "c=300000.0 km/s"
Viteza luminii reflectate: "s/mk 0.000003=c"
```

Clasa StringBuilder

Este compatibila cu clasa **StringBuffer** dar fara garantii de siguranta in aplicatii concurente.

Compararea obiectelor

Clasa **Object** are metodele :

`clone`, **`equals`**, `finalize`, **`getClass`**, `hashCode`, `notify`, `notifyAll`, **`toString`**, `wait`, `wait`, `wait`

Metoda **equals()** din **Object** compara referintele. Daca se doreste mai mult trebuie rescrisa.

```
public boolean equals(Object o) {
   if(o==null)
          return false;
   if(o==this)
          return true;
   if(!(o instanceof Punct))
          return false;
   Punct p = (Punct)o;
   return x==p.x && y==p.y;
}
```

EXEMPLU COMPLET (discutat la curs, pe baza sugestiilor si intrebarilor studentilor)

```
package lab.comparpuncte;

/**
 * @author Pentiuc_St_Gh + anul IIIC, oct.2019
 */
```

```
public class Punct {
    private int x,y;

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

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

    public Punct setX(int x) { // special pt. exercitiu
        this.x = x;
        return this;
    }
    @Override
    public boolean equals(Object o) {
        if(o==null)
            return false;
        if(o==this)
            return true;
        if(!(o instanceof Punct))
            return false;
        Punct p = (Punct)o;
        return x==p.x && y==p.y;
   }

}
```

```
package lab.comparpuncte;
public class DemoComparPuncte {
    public static void main (String[] a){
        Punct   p=new Punct(1,2),
            q=new Punct(1,2),
            z = new Punct(-10,2);
        System.out.println((p==q)+" "+ p.equals(q));
        System.out.println(q.equals(z.setX(1))  +" "+
                           p.equals(p.setX(20)));
        System.out.println("p="+p+"  q="+q+"  z="+z);
    }
}
```

Rezultatul executiei:

```
false true
true true
p=(20, 2}  q=(1, 2}  z=(1, 2}
```

Explicati !

Compararea obiectelor potrivit unei **relatii de ordine** este posibilă prin implementarea uneia din interfeţele

- **java.lang.Comparable**
- **java.util.Comparator**

Interfaţa java.util.Comparator

```
public interface Comparator<T> {
    public int compare(T o1,  T o2);
}
```

In versiunile mai vechi (formatul brut) :

```
public interface Comparator {
    public int compare(Object o1,  Object o2);
}
```

Metode care accepta obiecte `Comparator ca argument`

```
Arrays.binarySearch(Object [] tablou, Object val, Comparator
compar)
Arrays.sort(Object [] tablou, Comparator compar)

Collections.binarySearch(List lista, Comparator compar)
Collections.max(Collection colectie, Comparator compar)
Collections.min(Collection colectie, Comparator compar)
Collections.sort(List lista, Comparator compar)
```

Exemplu format brut (versiuni anterioare)

```
import java.util.Comparator;

public class ComparatieSiruri implements Comparator{

  public int compare(Object x, Object y) {
    String s1 = (String)x;
    String s2 = (String)y;
    return s1.toLowerCase().compareTo(s2.toLowerCase());
  }
}
```

Exemplu Java 8

```
import java.util.Comparator;

class ComparatieSiruri implements Comparator<String>{
  @Override
  public int compare(String s1, String s2) {
    return s1.toLowerCase().compareTo(s2.toLowerCase());
```

```
   }
}

class Aplicatie {
//...

    String nume[]=
                 {"Maria","Paula","Stefan","Ioana","George","Alina"};
    Arrays.sort(nume, new ComparatieSiruri());
    for(String s: nume)
        System.out.print(s+" ");
//...
}
```

Interfaţa `java.lang.Comparable`

```
public interface Comparable<T> {
    int compareTo(T o) ;
}
```

Versiuni anterioare, formatul brut:

```
public interface Comparable {
    int compareTo(Object obj) ;
}
```

Este necesară atunci când se doreşte implementarea unei relaţii de **ordine naturală** peste mulţimea obiectelor unei clase. Obiectele claselor care implementează această interfaţă pot fi sortate fără a se defini un alt comparator (elementele din `List` cu `Collection.sort`, din `Arrays` cu `Arrays.sort`). Obiectele unei astfel de clase pot constitui elementele unui obiect al unei clase ce implementează `SortedSet` sau cheile pentru o clasă ce implementează `SortedMap`.

Interfaţa `Comparable` conţine o singură metodă

```
public int compareTo(T o)
```

Această metodă trebuie să compare obiectul curent cu cel specificat prin argument şi să returnrze un întreg negativ dacă obiectul curent este mai mic, zero dacă sunt egale sau un întreg pozitiv dacă obiectul curent este mai mare. Metoda care va implementa comparaţia trebuie să respecte axiomele de definiţie ale relaţiei de ordine.icare ar fi obiectele x şi y valoarea returnată de **`x.compareTo(y)`** **trebuie să fie de semn contrar valorii returnate de** **`y.compareTo(x)`**.

2. Metoda implementată trebuie să fie tranzitivă:

`x.compareTo(y)>0 && y.compareTo(z)>0)` implică `x.compareTo(z)>0`.

3. Dacă `x.compareTo(y)==0` atunci oricare ar fi `z` semnul valorii returnate de `x.compareTo(z)` este acelaşi cu semnul valorii returnate de `y.compareTo(z)`.

Este recomandat imperativ ca metoda `compareTo()` implementată să fie consistentă cu metoda `equals()` utilizată

`(x.compareTo(y)==0) == (x.equals(y)), oricare ar fi x şi y`

Dacă nu se respectă această recomandare, acest lucru trebuie notificat utilizatorilor ei (exemplu: "această clasă posedă o ordine naturală care nu este consistentă cu equals").

Excepţia `ClassCastException` este lansată atunci când obiectul din argument nu poate fi comparat cu obiectul curent.
Excepţia NullPointerException este lansată atunci când argumentul este null (metoda equals pentru argument null , ob.equals(null), nu lansează nicio excepţie ci returnează **false**).

Exemplu

```
package comparatii;
import java.util.*;

public class Stud2Note implements Comparable<Stud2Note> {
  private int notaSD, notaJava;

  public Stud2Note(int n1,int n2) {
     notaSD   = n1;
     notaJava = n2;
  }
  public int getNotaSD() {
     return notaSD;
  }

  @Override
  public int compareTo(Stud2Note x) {
    if(x==null) return 1;
    if(equals(x)) return 0;
    float media1 = (notaSD + notaJava) /2,
          media2 = (x.notaSD + x.notaJava)/2;
    return media1 < media2 ? -1 : 1;
  }

  @Override
  public boolean equals(Object x) {
    return
      x!=null
      && (x instanceof Stud2Note)
      && (notaSD == ((Stud2Note)x).notaSD)
      && (notaJava == ((Stud2Note)x).notaJava);
  }

  @Override
  public String toString() {
     return "(SD =" + notaSD + " notaJava =" +notaJava+")";
  }
```

```
@Override
    public int hashCode() {
        int hash = 5;
        hash += 37 * hash + this.notaSD;
        hash += 41 * hash + this.notaJava;
        return hash;
    }

  public static void main(String args[])
  {
      Stud2Note e=new Stud2Note(9,5),
              f=new Stud2Note(7,10),
              g=new Stud2Note(9,5),
              t[]= new Stud2Note[]{e,f,g},
              p[]=t;

      System.out.println("e: "+e+"\nf: "+f+"\ng: "+g);
      System.out.println("e.compareTo(f)="+e.compareTo(f));
      System.out.println("e.compareTo(g)="+e.compareTo(g));
      System.out.println("f.compareTo(g)="+f.compareTo(g));
      System.out.println("t="+t[0]+" "+t[1]+" "+t[2]);
      Arrays.sort(t);
      System.out.println("Dupa sort(t)="+t[0]+" "+t[1]+" "+t[2]);
      System.out.println("p="+p[0]+" "+p[1]+" "+p[2]);
      Arrays.sort(p, new Comparatie());
      System.out.println("Dupa sort(p,new Comparatie())="+
                                  p[0]+" "+p[1]+" "+p[2]);
  }
}
//
// Acest comparator realizeaza comparatia a doi Stud2Notei
// doar pe baza notei de la SD
//
class Comparatie implements Comparator<Stud2Note> {
   @Override
   public int compare(Stud2Note a, Stud2Note b)
   {
      int n1 = a.getNotaSD(),
          n2 = b.getNotaSD();
      return (n1 < n2 ? -1 : (n1 == n2 ? 0 : 1));
   }
   /*   public int compare(Object a, Object b) {
           int n1=((Stud2Note)a).getNotaSD(),
               n2=((Stud2Note)b).getNotaSD();
           return (n1 < n2 ? -1 : (n1 == n2 ? 0 : 1));
        }
   */
}
```

Rezultatul execuţíei

```
C:\pg\sd\Curs8>java Stud2Note
e: (SD =9 Java =5)
f: (SD =7 Java =10)
g: (SD =9 Java =5)
e.compareTo(f)=-1
e.compareTo(g)=0
f.compareTo(g)=1
t=(SD =9 Java =5) (SD =7 Java =10) (SD =9 Java =5)
Dupa sort(t)=(SD =9 Java =5) (SD =9 Java =5) (SD =7 Java =10)
p=(SD =9 Java =5) (SD =9 Java =5) (SD =7 Java =10)
Dupa sort(p,new Comparatie())=
                        (SD =7 Java =10) (SD =9 Java =5) (SD =9
Java =5)
```

Intrebari

1. Ce se intampla daca la apelul metodelor `compare()` si `compareTo()` argumentele efective nu sunt de tip Stud2Note ?
2. Ce s-ar afisa daca la sfarsitul metodei `main()` s-ar adauga `System.out.println("t="+t[0]+" "+t[1]+" "+t[2]); System.out.println(t);` **?**

Clasa Arrays

Aceasta clasa din pachetul java.util contine metode statice de prelucrare a tablourilor.
Documentatia o gasiti la adresa https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Cateva metode utile sunt prezentate in tabelul urmator.
Nota. In tabelul s-au folosit notatiile :

- **array**- poate fi un tablou de tipuri primitive (short[], int[], …, double[]) sau un tablou de obiecte (Object[], T[]),
- **value** - o valoare de acelasi tip cu elementele tabloului.

Metoda	Descriere
binarySearch(**array**, **value**)	Cauta in tabloul **array** valoarea , **value.** Daca valoarea e in tablou se intoarce indicele elemntului de tablou care o contine, altfel se intoarce <0
binarySearch(**array**, fromIndex, toIndex, **value**)	Se cauta de la indicele **from** (inclusiv) la indicele **to** (exclusiv)
binarySearch(**T[] a**, int fromIndex, int toIndex, **T key**, Comparator<? super T> c)	Cautarea utilizeaza comparatorul **c**
copyOf(**array**, **length**)	Returneaza o copie a tabloului, noua lungime fiind **length**, daca e mai mare decat lungimea initiala se completeaza cu null
copyOfRange(T[] original, int from, int to)	Se copiaza de la indicele **from** (iclusiv) la indicele **to** (exclusiv)
equals(**array1**, **array2**)	Returneaza **true** daca ambele tablouri au aceleasi elemente in aceeasi ordine
fill(**array**, **value**)	Se initializeaza elementele din **array** cu **value**
fill(Object[] a, int fromIndex, int toIndex, Object val)	Se initializeaza elementeel din a de la indicele from (iclusiv) la indicele to (exclusiv)

sort(**array**)	Se sorteaza elementele din **array**. Daca **array** este un tablou de obiecte atunci pentru sortare se utilizeaza ordinea naturala (obiectele trebuie sa apartina unei clase care implementeaza interfata java.lang.Comparable, adica au definita metoda compareTo())
sort(Object[] a, int fromIndex, int toIndex)	Se sorteaza incepand cu fromIndex inclusiv, pana la toIndex **exclusiv.**Oobiectele din tabloul a trebuie sa apartina unei clase care implementeaza interfata java.lang.Comparable
sort(T[] a, Comparator<? super T> c)	Se utilizeaza comparatorul c pentru sortare
sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)	Se sorteaza incepand cu fromIndex inclusiv, pana la toIndex exclusiv. Se utilizeaza comparatorul c pentru sortare
toString(**array**)	Converteste tabloul intr-un sir utilizand metoda toString() a fiecarui elment din tablou; exemplu[12, ,' abc' 3,14]

Clase anonime

Clasele anonime sunt clase fara nume, interioare care dat fiind ca sunt anonime nu pot fi instantiate decat o singura data.

```
abstract  class Student {
    String nume;
    public Student(String nume) {
        this.nume = nume;
    }
    abstract void studiaza();

}
public class TestClasaAnonima {

   public static void main(String[] args) {

      Student s = new Student("Ionescu"){
                        void studiaza(){
                            System.out.println(nume +
                                  " a terminat tema la SDA");
                        }
      };
      s.studiaza();
  }
}
```

In acest caz clasa anonima a fost o extindere a clasei abstracte Student prin implementarea metodei abstracte studiaza(). Secventa marcata cu violet executa doua lucruri:

- defineste o clasa anonima care extinde clasa Student
- creeaza o instanta a acestei clase anonime

Iesirea produsa este urmatoarea:

```
Ionescu a terminat tema la SDA
```

Putem avea si 2 sau mai multe implementari anonime ale aceleeasi clase abstracte

```
public class TestClasaAnonima {
    public static void main(String[] args) {

        Student stud1 = new Student("Ionescu"){
            void studiaza(){
                System.out.println(nume + " a terminat tema la SDA");
            }
        };

        Student stud2 = new Student("Vasilescu"){
            void studiaza(){
                System.out.println(nume + " a configurat un Access Point");
            }
        };

        stud1.studiaza();
        stud2.studiaza();
    }
}
```

Iesirea produsa este urmatoarea:

```
Ionescu a terminat tema la SDA
Vasilescu a configurat un Access Point
```

Putem avea clase anonime interioare care sa extinda in acelasi mod o clasa parinte.

Clase interioare anonime care implementeaza o interfata

Fie interfata **Voluntar**

```
public interface Voluntar {
    public void actiune();
}
```

In clasa de mai jos se defineste o clasa interioara anonima care implementeaza interfata si creeaza un obiect din aceasta clasa anonima.

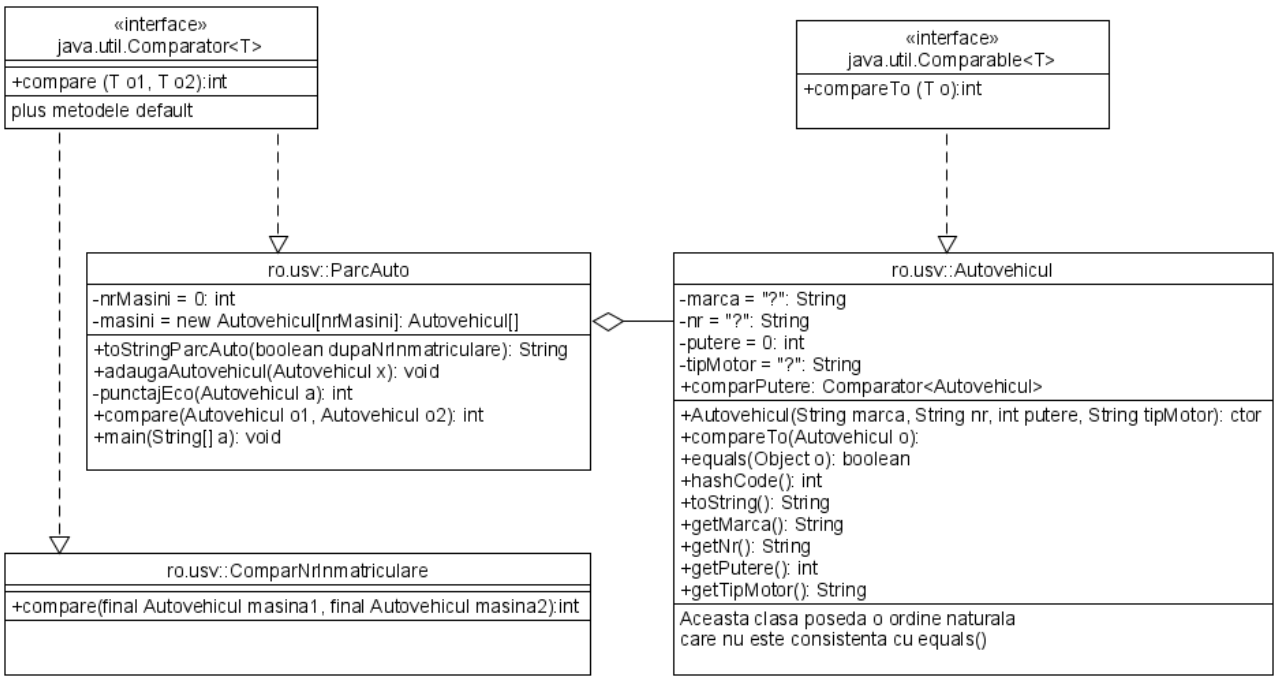
```
public class TestClasaAnonima2 {

  public static void main(String[] args) {

    Voluntar v = new Voluntar(){
        @Override
        public void actiune() {
            System.out.println("Tutoriat cu studentii din ani mai mici");
        }

    };
    v.actiune();
  }
}
```

Pentru mai multe informatii accesati
https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Clase anonime utilizate ca si comparatori

Utilizarea comparatorilor si a claselor anonmie va fi prezentata printr-un exemplu care utilizeaza sortarea bazata pe ordinea naturala sau cu un comparator. Acesta poate fi o instanta a unei clase externe (ca in exemplul precedent cu *Stud2Note*) sau o instanta a unei clase anonime ca in exemplul de mai jos in care constructorul clasei anonime este marcat cu galben.

Aplicatia din exemplu este compusa din trei clase: ParcAuto (main), Autovehicul si ComparNumarInmatriculare care implementeaza interfetele comparable si Comparator din java,util si utilizeaza clase anonime cionstruite cu interfata Comparator.



Nota. *Referitor la interfata **java.util.Comparator** se reaminteste faptul ca intr-o interfata Java pot exista metode abstracte, dar si metode concrete, declarate cu* `default`*, care constituie o implementare implicita si pot fi utilizate in clasele care implementeaza interfata sau optional suprascrise daca se doreste acest lucru.*

Clasa Autovehicul

Pentru a demonstra cât mai multe posibilități de realizare a comparației obiectelor în această clasă sunt doi comparatori

- comparatorul care realizează ordinea naturală (după tipul motorului: C(onventional), E(lectric), H(ibrid)) și implementează interfața **Comparable<Autovehicul>**

```
public int compareTo(Autovehicul o)
```

- un obiect comparator realizat cu o clasă anonimă care extinde clasa `Comparator<Autovehicul>`. Acest obiect comparator are referința memorată în variabila statică *`comparPutere`* și prin modul în care este scris determină sortarea în ordine descrescătoare, după valoarea câmpului putere din fiecare obiect de tip Autovehicul.

```
package ro.usv;
import java.util.Comparator;
import java.util.Objects;
```

```java
public class Autovehicul implements Comparable<Autovehicul> {
    private String marca="?";
    private String nr="?";
    private int putere=0;
    private String tipMotor="?";

    public Autovehicul(String marca, String nr, int putere, String tipMotor) {
        if(marca!=null)
            this.marca = marca;
        if(nr!=null)
            this.nr = nr;
        if(tipMotor!=null)
            this.tipMotor = tipMotor;
        if(putere>0)
            this.putere = putere;
    }

    public static Comparator<Autovehicul> comparPutere = new Comparator<>() {
        @Override
        public int compare(Autovehicul o1, Autovehicul o2) {
           if (o1.getPutere() < o2.getPutere())
                return -1;
            else
                return o1.getPutere() == o2.getPutere() ? 0 : 1;
        }
    };

    @Override
    //această clasă posedă o ordine naturală care nu este consistentă cu equals
    public int compareTo(Autovehicul o) {
        return tipMotor.compareToIgnoreCase(o.tipMotor);
    }

  @Override
  public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      Autovehicul that = (Autovehicul) o;
      return putere == that.putere &&
              marca.equals(that.marca) &&
              nr.equals(that.nr) &&
              tipMotor.equals(that.tipMotor);
  }

  @Override
  public int hashCode() {
      return Objects.hash(marca, nr, putere, tipMotor);
  }

@Override
public String toString() {
    return "{" + marca + ", " + nr +
            ", putere=" + putere +
            ", tipMotor='" + tipMotor + '\'' +
            '}';
}

public String getMarca() {
    return marca;
}

public String getNr() {
    return nr;
}
```

```
    public int getPutere() {
        return putere;
    }

    public String getTipMotor() {
        return tipMotor;
    }
}
```

Observatii

1) Cei doi comparatori din clasa **Autovehicul** sunt utilizați în metoda main() din clasa **ParcAuto**.
   a) Comparatorul de ordine naturală (prin implementarea interfaței **Comparable**)

```
System.out.println("\nSortare dupa tipul motorului");
Arrays.sort(p.masini, 0, p.nrMasini);
System.out.println(Arrays.toString(p.masini));
```

   b) Comparatorul realizat prin extinderea interfetei **Comparator**; se reamintește că referința către acest obiect comparator este memorată în variabila statică **`Autovehicul`**.*comparPutere*

Utilizarea acestor doi comparatori este in secventa urmatoare:

```
System.out.println("\nSortare dupa putere (crescator)");
Arrays.sort(p.masini, 0, p.nrMasini, Autovehicul.comparPutere);
System.out.println(Arrays.toString(p.masini));

System.out.println("\nSortare dupa putere (descrescator)");
Arrays.sort(p.masini,0,p.nrMasini, Autovehicul.comparPutere.reversed());
System.out.println(Arrays.toString(p.masini));
```

Se va observa ca pentru sortarea descrescatoare s-a utilizat un obiect comparator care sa permita obtinerea acestui mod de sortare prin aplicarea metodei *reversed()* din **Comparator** (metoda *default* in interfata)

```
Autovehicul.comparPutere.reversed()
```

2) Metodele equals() si hashCode() au fost generate automat; pentru metoda equals() s-a indcat ca valorile campurilor clasei sunt non null; de caeea au fost initializate cu "?" si inconstructor s-au pus validari pentru a nu permite crearea de obiecte cu null pentru numele marcii, tip motor sau nr. inmatriculare
3) Clasa **Autovehicul** are o ordine naturala bazata numai pe tipul motorului, in timp ce equals() verifica egalitatea tuturor celor 4 campuri; de aceea s-a indica ca aceasta clasa are o ordine naturala care nu este consistenta cu equals().

Clasa exterioară `ComparNrInmatriculare`

Ilustreaza o alta posibilitate de a crea un comparator printr-o clasa speciala care sa implementeze interfata Comparator<T>.
Comparatorul implementat aici face ca algoritmul de sortare sa 'impinga' toate valorile null din vector catre indicii mai mari (catre la sfarsitul vectorului)

```
package ro.usv;

import java.util.Comparator;
```

```
public class ComparNrInmatriculare implements Comparator<Autovehicul> {
    @Override
    // pentru sortarea tablourilor care contin null acestea vor apare la sfarsit
    public int compare(final Autovehicul masina1, final Autovehicul masina2) {
        if (masina1==masina2) return 0;
        if (masina1==null) return 1;    // pentru ca sort() sa mute null la urma
        if (masina2==null) return -1;   // idem
        return masina1.getNr().compareToIgnoreCase(masina2.getNr());
    }
}
```

Apelul acestui comparator este in metoda main() din clasa ParcAuto

```
Arrays.sort(masini, new ComparNrInmatriculare());
```

Deci fara a preciza indexul de inceput si sfarsit din tablou in care sort() executa sortarea.

Clasa ParcAuto

Este o clasa care reprezinta o multime de masini implementata printr-un vector `Autovehicul masini[]`. Metoda main() afiseaza aceasta multime in diferite ordini posibile pentru a demonstra diversitatea operatorilor folositi.
In aceasta clasa sunt implementai 2 comparatori:

1) un Comparator care compara autovehiculele dupa marca prin implementarea unei clase anonime (marcat cu galben)
2) clasa ParcAuto implementeaza interfata `Comparator<Autovehicul>` si are un Comparator care compara doua autovehicule cu ajutorul unui punctaj Eco ce favorizeaza autovehiculele cu motor electric sau hybrid si pe cele cu putere mica (marcat cu albastru) si utilizat astfel

```
Arrays.sort(p.masini, 0, p.nrMasini, p);
```

Utilizarea acestui ultim comparator este realizata cu obiectul de tip **ParcAuto** referit de varabila *p*.

Class ParcAuto

```
package ro.usv;

import java.util.Arrays;
import java.util.Comparator;

public class ParcAuto implements Comparator<Autovehicul> {
    private int nrMasini =0;
    private Autovehicul[] masini = new Autovehicul[nrMasini];

    public String toStringParcAuto(boolean dupaNrInmatriculare)
    {
        if(dupaNrInmatriculare)
           Arrays.sort(masini, new ComparNrInmatriculare());
        else
           Arrays.sort(masini, 0, nrMasini,
             new Comparator<Autovehicul>() {    // clasa anonima
               @Override
               public int compare(final Autovehicul m1, final Autovehicul m2) {
                     return m1.getMarca().compareToIgnoreCase(m2.getMarca());
               }
             } // sfarsit clasa anonima
        );
        return Arrays.toString(masini);
    }
```

```
    public void adaugaAutovehicul(Autovehicul x){
        if (nrMasini>=masini.length){
            masini = Arrays.copyOf(masini, nrMasini+10);
        }
        masini[nrMasini++] = x;
    }

    private static int punctajEco(Autovehicul a){
        switch(a.getTipMotor()){
            case "E":
                return a.getPutere();
            case "H":
                return 10000 + a.getPutere();
            case "C":
                return 20000 + a.getPutere();
            default:
                return 30000;
        }
    }
    @Override
    public int compare(Autovehicul o1, Autovehicul o2) {
        int n1 = punctajEco(o1);
        int n2 = punctajEco(o2);
        return n1<n2 ? -1: (n1==n2?0:1);
    }

    public static void main(String[] a){
        ParcAuto p = new ParcAuto();
        p.adaugaAutovehicul(new Autovehicul("Logan", "NT-17-ADE", 95, "C"));
        p.adaugaAutovehicul(new Autovehicul("WW", "CJ-07-ABC", 125, "E"));
        p.adaugaAutovehicul(new Autovehicul("Logan", "SV-07-ABC", 80, "C"));
        p.adaugaAutovehicul(new Autovehicul("Opel", "BT-01-XYX", 100, "H"));

        System.out.println("\nSortare dupa tipul motorului");
        Arrays.sort(p.masini, 0, p.nrMasini);
        System.out.println(Arrays.toString(p.masini));

        System.out.println("\nSortare dupa putere (crescator)");
        Arrays.sort(p.masini, 0, p.nrMasini, Autovehicul.comparPutere);
        System.out.println(Arrays.toString(p.masini));

        System.out.println("\nSortare dupa putere (descrescator)");
        Arrays.sort(p.masini,0,p.nrMasini, Autovehicul.comparPutere.reversed());
        System.out.println(Arrays.toString(p.masini));

        System.out.println("\nSortare dupa marca");
        System.out.println(p.toStringParcAuto(false));

        System.out.println("\nSortare dupa nr. inmatriculare");
        System.out.println(p.toStringParcAuto(true));

        System.out.println("\nSortare dupa punctaj Eco");
        Arrays.sort(p.masini, 0, p.nrMasini, p);
        System.out.println(Arrays.toString(p.masini));
    }
}
```

Nota. In acest exemplu veti mai observa si alte utilizari ale clasei Arrays.

1. Realizarea unei copii a tabloului **`masini[]`** atunci cand se depaseste capacitatea

```
    private Autovehicul[] masini = new Autovehicul[10];
    private int nrMasini =0;
```

```
public void adaugaAutovehicul(Autovehicul x){
     if (nrMasini>=masini.length){
         masini = Arrays.copyOf(masini, nrMasini+10);
     }
     masini[nrMasini++] = x;
}
```

2. Deoarece in tabloul **masini[]** exista valori null, in astfel de situatii exista 2 solutii de evitare a exceptiei nullPointer **la sortare**.
   a) Daca numai primele elemente sunt diferite de null, iar restul sunt null (ca in ex. nostru unde primele *nrMasini* elemente sunt nenule), atunci se foloseste sortarea doar a primelor elemente (de la indicele 0 inclusiv la indicele *nrMasini* exclusiv)

```
Arrays.sort(masini, 0, nrMasini, new Comparator<Autovehicul>(){ …. } );
```

   b) Daca valorile null sunt amestecate printre elementele nenule ale tabloului este necesara introducerea de cod in comparator care sa verifice daca exista valori null si sa le ‚impinga' pe acestea spre sfarsitul tabloului.

```
@Override
public int compare(final Autovehicul masina1, final Autovehicul masina2)
{
        if (masina1==masina2) return 0;
        if (masina1==null) return 1;   // pentru ca sort() sa mute null la urma
        if (masina2==null) return -1;  // idem
        return
                masina1.getNr().compareToIgnoreCase(masina2.getNr());
}
```

OBSERVATIE

In clasa **ParcAuto** putea fi utilizata o lista pentru a stoca masinile firmei si nu ar mai fi fost aceste probleme.
:

```
import java.util.ArrayList;
import java.util.List;
// . . .
   private List masini = new ArrayList<Autovehicul>();
// . . .
```

List este interfata, iar ArrayList<T> este o clasa ce implementeaza interfata List. Avantajele acestei solutii le vom prezenta ulterior.

S-a preferat sa se utilizeze tablouri atat din motive legate de momentl prezentarii cursului, dar si datorita faptului ca utilizarea tablourilor are avantajul vitezei.

3. Conversia la sir a tabloului

```
Arrays.toString(masini);
```

Iesirea produsa a fost urmatoarea:

```
Sortare dupa tipul motorului
[{Logan, NT-17-ADE, putere=95, tipMotor='C'}, {Logan, SV-07-ABC, putere=80,
tipMotor='C'}, {WW, CJ-07-ABC, putere=125, tipMotor='E'}, {Opel, BT-01-XYX,
putere=100, tipMotor='H'}, null, null, null, null, null, null]

Sortare dupa putere (crescator)
```

```
[{Logan, SV-07-ABC, putere=80, tipMotor='C'}, {Logan, NT-17-ADE, putere=95,
tipMotor='C'}, {Opel, BT-01-XYX, putere=100, tipMotor='H'}, {WW, CJ-07-ABC,
putere=125, tipMotor='E'}, null, null, null, null, null, null]

Sortare dupa putere (descrescator)
[{WW, CJ-07-ABC, putere=125, tipMotor='E'}, {Opel, BT-01-XYX, putere=100,
tipMotor='H'}, {Logan, NT-17-ADE, putere=95, tipMotor='C'}, {Logan, SV-07-ABC,
putere=80, tipMotor='C'}, null, null, null, null, null, null]

Sortare dupa marca
[{Logan, NT-17-ADE, putere=95, tipMotor='C'}, {Logan, SV-07-ABC, putere=80,
tipMotor='C'}, {Opel, BT-01-XYX, putere=100, tipMotor='H'}, {WW, CJ-07-ABC,
putere=125, tipMotor='E'}, null, null, null, null, null, null]

Sortare dupa nr. inmatriculare
[{Opel, BT-01-XYX, putere=100, tipMotor='H'}, {WW, CJ-07-ABC, putere=125,
tipMotor='E'}, {Logan, NT-17-ADE, putere=95, tipMotor='C'}, {Logan, SV-07-ABC,
putere=80, tipMotor='C'}, null, null, null, null, null, null]

Sortare dupa punctaj Eco
[{WW, CJ-07-ABC, putere=125, tipMotor='E'}, {Opel, BT-01-XYX, putere=100,
tipMotor='H'}, {Logan, SV-07-ABC, putere=80, tipMotor='C'}, {Logan, NT-17-ADE,
putere=95, tipMotor='C'}, null, null, null, null, null, null]
```

Daca am fi utilizat

```
System.out.println(p);
```

Iesirea produsa ar fi fost doar: `ParcAuto@31befd9f`
Daca utilizam **`System.out.println(Arrays.toString(p.masini));`** tabloul p.masini este afisat ca mai sus.

ALTA OBSERVATIE

Atentie la metode publice care returneaza referinta catre un tablou sau un alt obiect. Exemplu

```
private Autovehicul[] masini = new Autovehicul[nrMasini];

public Autovehicul[] getMasini() {
    return masini;
}
```

Din exterior nu se poate modifica referinta catre vectorul masini[], insa continutul acestuia poate fi mdificat printr-o atribuire

```
p.getMasini()[0]=null;
```

Chiar daca se declara final masini nu se impiedica modificarea ca mai sus, deoarece declararea

```
private final Autovehicul[] masini = new Autovehicul[nrMasini];  //are implicatii
```

impiedica doar modificarea referintei masini nu si a continutului vectorului masini[i]. Pe de alta parte nici nu poate fi declarat final deoarece valoarea referintei se modifica in functia

```
public void adaugaAutovehicul(Autovehicul x){
  if (nrMasini>=masini.length){
    masini = Arrays.copyOf(masini, nrMasini+10); //imposibil cu final masini
  }
  masini[nrMasini++] = x;
}
```

Daca ar fi final nu se poate face masini =…, insa masini[i]= … este posibil.