




# Curs practic de Java

## *Curs 12*

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică  
**Universitatea "Al. I. Cuza" Iași**





# Lucrul dinamic cu clase



# Cuprins

---

- Incarcarea claselor în memorie
- Reflection API
- Examinarea claselor și interfețelor
- Lucrul dinamic cu obiecte
- Lucrul dinamic cu vectori
- Crearea proceselor \*



# **Incărcarea claselor în memorie**



# Ciclul de viață al unei clase

1. **Incărcarea în memorie** - va fi instanțiat un obiect de tip `java.lang.Class`.
2. **Editarea de legături** - incorporarea noului tip de date în JVM.
3. **Inițializarea** - execuția blocurilor statice de inițializare și inițializarea variabilelor de clasă.
4. **Descărcarea** - Atunci când nu mai există nici o referință de tipul clasei respective, obiectul de tip `Class` creat va fi marcat pentru a fi eliminat din memorie de către *garbage collector*.

# Clasa *ClassLoader*

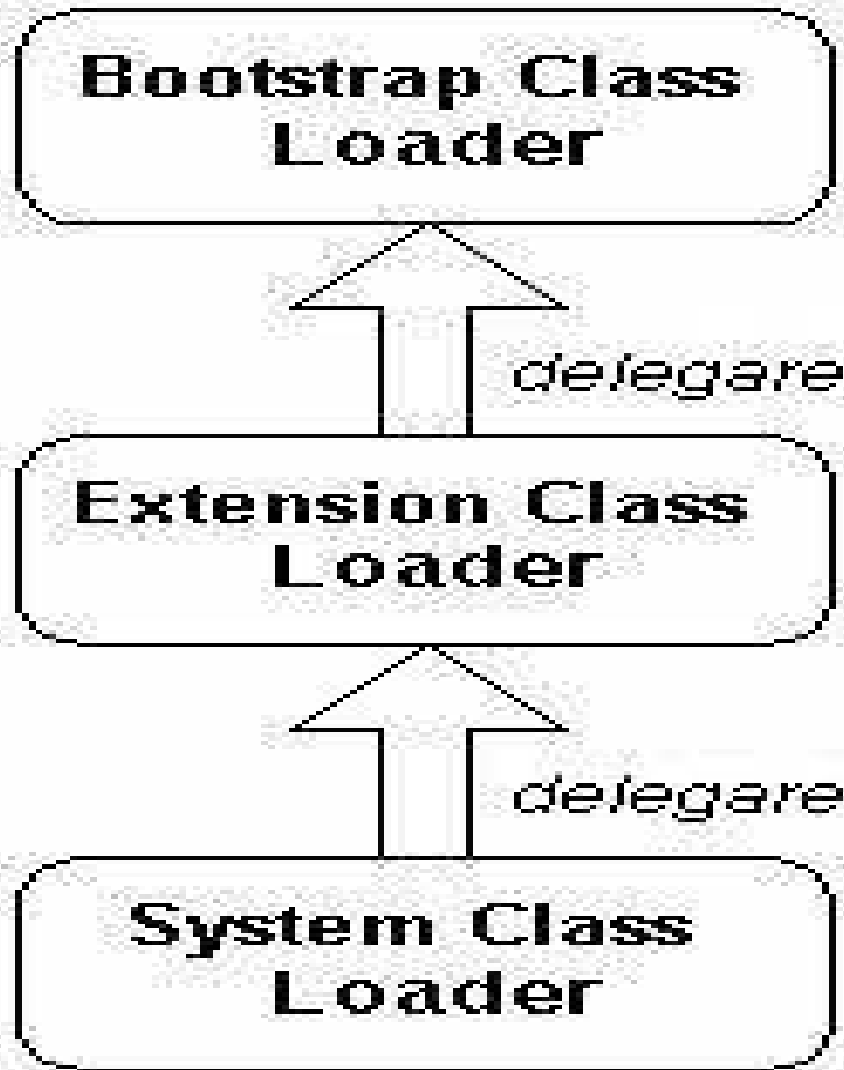
1. **Class loader-ul primordial (eng. bootstrap)** -  
Reprezintă o parte integrantă a mașinii virtuale, fiind responsabil cu încărcarea claselor standard din distribuția Java.
2. **Class loader-e proprii** - Acestea nu fac parte intrinsecă din JVM și sunt instanțe ale clasei `java.lang.ClassLoader`. Aceasta este o clasă abstractă, tipul efectiv al obiectului fiind așadar derivat din aceasta.

# Class loader-e implicite

- **Bootstrap Class Loader** - Class loader-ul primordial. Acesta este responsabil cu încărcarea claselor din distribuția Java standard (cele din pachetele `java.*`, `javax.*`, etc.).
- **Extension Class Loader** - încărcarea claselor din directoarele extensiilor JRE.
- **System Class Loader** - încărcarea claselor proprii aplicațiilor Java (cele din CLASSPATH). Tipul acestuia este `java.lang.URLClassLoader`.

Fiecare obiect `Class` va reține class loader-ul care a fost folosit pentru încărcare, acesta putând fi obținut cu metoda `getClassLoader`.

# Modelul "delegat"





# Incărcarea unei clase în memorie

- **loadClass** apelată pentru un obiect de tip `ClassLoader`

```
ClassLoader loader = new MyClassLoader();  
loader.loadClass("NumeCompletClasa");
```

- **Class.forName**

```
Class c = Class.forName("NumeCompletClasa");  
// echivalent cu  
ClassLoader loader = this.getClass().getClassLoader();  
loader.loadClass("ClasaNecunoscuta");  
  
// Clasele standard pot fi si ele incarcate astfel  
Class t = Class.forName("java.lang.Thread");
```

# Instanțierea unui obiect

Dacă dorim să instanțiem un obiect dintr-o clasă încărcată dinamic putem folosi metoda **newInstance**, cu condiția să existe constructorul fără argumente pentru clasa respectivă.

```
Class c = Class.forName("java.awt.Button");  
Button b = (Button) c.newInstance();
```

# TestFuncții.java

```
public class TestFuncții {
    public static void main(String args[]) throws IOException {
        int n=10, v[] = new int[n];
        Random rand = new Random();
        for(int i=0; i<n; i++) v[i] = rand.nextInt(100);
        // Citim numele unei funcții
        BufferedReader stdin = new BufferedReader(
                                new InputStreamReader(System.in));

        String numeFuncție = "";
        while (! numeFuncție.equals("gata")) {
            numeFuncție = stdin.readLine();
            try {
                Class c = Class.forName(numeFuncție);
                Funcție f = (Funcție) c.newInstance();
                f.setVector(v); // sau f.v = v;
                int ret = f.executa();
                System.out.println("\nCod returnat: " + ret);
            } catch (Exception e) { ... }
        }
    }
}
```

# Funcție.java

```
public abstract class Funcție {  
    public int v[] = null;  
    public void setVector(int[] v) {  
        this.v = v;  
    }  
    public abstract int executa();  
}
```

# Diverse funcții



## Sort.java

```
public class Sort extends Functie {
    public int executa() {
        if (v == null) return -1;
        Arrays.sort(v);
        for(int i=0; i<v.length; i++) System.out.print(v[i] + " ");
        return 0;
    }
}
```

## Max.java

```
public class Max extends Functie {
    public int executa() {
        if (v == null) return -1;
        int max = v[0];
        for(int i=1; i<v.length; i++)
            if (max < v[i])
                max = v[i];
        System.out.print(max);
        return 0;
    }
}
```



# Folosirea class loader-ului curent



## URLClassLoader

```
// Obtinem class loaderul curent
URLClassLoader urlLoader =
    (URLClassLoader) this.getClass().getClassLoader();

// Adaugam directorul sub forma unui URL
urlLoader.addURL(new File("c:\\\\clase").toURL());

// Incarcam clasa
urlLoader.loadClass("demo.Test");
```



# Crearea unui class loader nou

```
public class MyClassLoader extends URLClassLoader{
    public MyClassLoader(URL[] urls){
        super(urls);
    }
}

// La initializare
URLClassLoader systemLoader =
    (URLClassLoader) this.getClass().getClassLoader();
URL[] urls = systemLoader.getURLs();

// Cream class loaderul propriu
MyClassLoader myLoader = new MyClassLoader(urls);
myLoader.loadClass("Clasa");
...
// Dorim sa reincarcam clasa
myLoader.loadClass("Clasa"); // nu functioneaza !
// Cream alt class loader
MyClassLoader myLoader = new MyClassLoader(urls);
myLoader.loadClass("Clasa"); // reincarca clasa
```



# Mecanismul reflectării





# Ce înseamnă *reflection* ?

Mecanismul prin care o clasă, interfață sau obiect "reflectă" la momentul execuției structura lor internă se numește *reflectare*.

- Determinarea clasei unui obiect.
- Aflarea unor informații despre o clasă (modificatori, superclasa, constructori, metode).
- Instanțierea unor clase al căror nume este știut abia la execuție.
- Setarea sau aflarea atributelor unui obiect, chiar dacă numele acestora este știut abia la execuție.
- Invocarea metodelor unui obiect al căror nume este știut abia la execuție.
- Crearea unor vectori a căror dimensiune și tip nu este știut decât la execuție.

# Reflection API

- `java.lang.Class`
- `java.lang.Object`
- Clasele din pachetul **`java.lang.reflect`** și anume:
  - `Array`
  - `Constructor`
  - `Field`
  - `Method`
  - `Modifier`



# Examinarea claselor și interfețelor



# Aflarea instanței Class



```
Class c = obiect.getClass();  
Class c = java.awt.Button.class;  
Class c = Class.forName("NumeClasa");
```

```
Class clasa = obiect.getClass();  
String nume = clasa.getName();
```

Interfețele sunt tot de tip `Class`, diferențierea lor făcându-se cu metoda `isInterface`.

Tipurile primitive sunt descrise și ele de instanțe de tip `Class` având forma *TipPrimitiv.class*: `int.class`, `double.class`, etc., diferențierea lor făcându-se cu ajutorul metodei `isPrimitive`.



# Aflarea modifcatorilor unei clase

## Metoda getModifiers

```
Class clasa = obiect.getClass();
int m = clasa.getModifiers();
String modif = "";
if (Modifier.isPublic(m))
    modif += "public ";
if (Modifier.isAbstract(m))
    modif += "abstract ";
if (Modifier.isFinal(m))
    modif += "final ";
System.out.println(modif + "class" + c.getName());
```

# Aflarea superclasei

## Metoda `getSuperclass`

Returnează `null` pentru clasa `Object`

```
Class c = java.awt.Frame.class;  
Class s = c.getSuperclass();  
System.out.println(s); // java.awt.Window
```

```
Class c = java.awt.Object.class;  
Class s = c.getSuperclass(); // null
```

# Aflarea interfețelor



## Metoda getInterfaces

```
public void interfete(Class c) {  
    Class[] interf = c.getInterfaces();  
    for (int i = 0; i < interf.length; i++) {  
        String nume = interf[i].getName();  
        System.out.print(nume + " ");  
    }  
}  
...  
interfete(java.util.HashSet.class);  
// Va afisa interfetele implementate de HashSet:  
// Cloneable, Collection, Serializable, Set  
  
interfete(java.util.Set);  
// Va afisa interfetele extinse de Set:  
// Collection
```



# Aflarea membrilor

- **Variable:** `getFields`, `getDeclaredFields`  
→ `Field`
- **Constructori:** `getConstructors`,  
`getDeclaredConstructors`  
→ `Constructor`
- **Metode:** `getMethods`, `getDeclaredMethods`  
→ `Method`
- **Clase imbricate:** `getClasses`,  
`getDeclaredClasses`

`getDeclaringClass`: clasa căreia îi aparține un membru.





# Manipularea obiectelor



# Crearea obiectelor

## • Metoda **newInstance** din clasa **java.lang.Class**

```
Class c = Class.forName("NumeClasa");  
Object o = c.newInstance();
```

## • Metoda **newInstance** din clasa **Constructor**

```
Class clasa = java.awt.Point.class;  
// Obtinem constructorul dorit  
Class[] signatura = new Class[] {int.class, int.class};  
Constructor ctor = clasa.getConstructor(signatura);  
// Pregatim argumentele  
// Ele trebuie sa fie de tipul referinta corespunzator  
Integer x = new Integer(10);  
Integer y = new Integer(20);  
Object[] arg = new Object[] {x, y};  
// Instantiem  
Point p = (Point) ctor.newInstance(arg);
```

# Invocarea metodelor



## Metoda **invoke** din clasa **Method**

```
Class clasa = java.awt.Rectangle.class;
Rectangle obiect = new Rectangle(0, 0, 100, 100);

// Obtinem metoda dorita
Class[] signatura = new Class[] {Point.class};
Method metoda = clasa.getMethod("contains", signatura);

// Pregatim argumentele
Point p = new Point(10, 20);
Object[] arg = new Object[] {p};

// Apelam metoda
metoda.invoke(obiect, arg);
```



# Setarea și aflarea variabilelor



## Metodele **set** și **get** din clasa **Field**

```
Class clasa = java.awt.Point.class;  
Point obiect = new Point(0, 20);
```

```
// Obținem variabilele membre  
Field x, y;  
x = clasa.getField("x");  
y = clasa.getField("y");
```

```
// Setam valoarea lui x  
x.set(obiect, new Integer(10));
```

```
// Obținem valoarea lui y  
Integer val = y.get(obiect);
```



# TestFuncții2.java

```
Class c = Class.forName(numFuncție);  
Object f = c.newInstance();  
  
Field vector = c.getField("v");  
vector.set(f, v);  
  
Method m = c.getMethod("executa", null);  
Integer ret = (Integer) m.invoke(f, null);  
  
System.out.println("\nCod returnat: " + ret);
```

# Lucrul dinamic cu vectori

Vectorii sunt reprezentați ca tip de date tot prin intermediul clasei `java.lang.Class`, diferențierea făcându-se prin intermediul metodei `isArray`. Tipul de date al elementelor din care este format vectorul va fi obținut cu ajutorul metodei `getComponentType`, ce întoarce o referință de tip `Class`.

```
Point []vector = new Point[10];
Class c = vector.getClass();
System.out.println(c.getComponentType());
// Va afisa: class java.awt.Point
```

# Obiecte de tip vector



## Clasa Array

- Crearea de noi vectori: `newInstance`
- Aflarea numărului de elemente: `getLength`
- Setarea / aflarea elementelor: `set`, `get`

```
Object a = Array.newInstance(int.class, 10);  
for (int i=0; i < Array.getLength(a); i++)  
    Array.set(a, i, new Integer(i));
```

```
for (int i=0; i < Array.getLength(a); i++)  
    System.out.print(Array.get(a, i) + " ");
```





# **Crearea și comunicarea cu procese externe**





# Crearea unui proces

```
public class TestRuntime {  
    public static void main(String args[]) throws Exception{  
        Runtime runtime = Runtime.getRuntime();  
        Process child = runtime.exec("java Aplicatie");  
  
        runtime.exec("cmd.exe /c start help.html");  
  
        String[] commands = new String[]{"notepad", "calc"};  
        runtime.exec(commands);  
    }  
}
```

# Fluxul de ieşire al unui proces



```
String command = "java Aplicatie";
Process child = Runtime.getRuntime().exec(command);

// Citim de pe fluxul de iesire al procesului
InputStream in = child.getInputStream();
int c;
while ((c = in.read()) != -1) {
    // proceseaza c
}
in.close();
```



# Fluxul de intrare al unui proces



```
String command = "java Aplicatie";  
Process child = Runtime.getRuntime().exec(command);  
  
// Scriem pe fluxul de intrare al procesului  
OutputStream out = child.getOutputStream();  
  
out.write("some text".getBytes());  
out.close();
```

