Metode avansate de programare

Curs 5 - Reflexia in Java (Java Reflection)

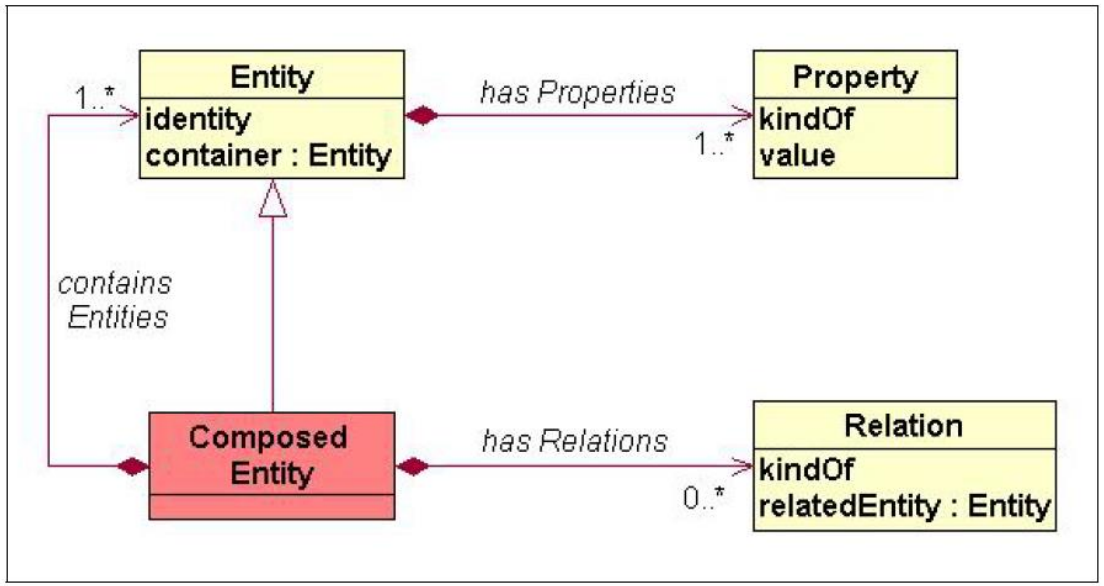
Reflexia

- Capacitatea unui program de a-si observa la executie propria structura;

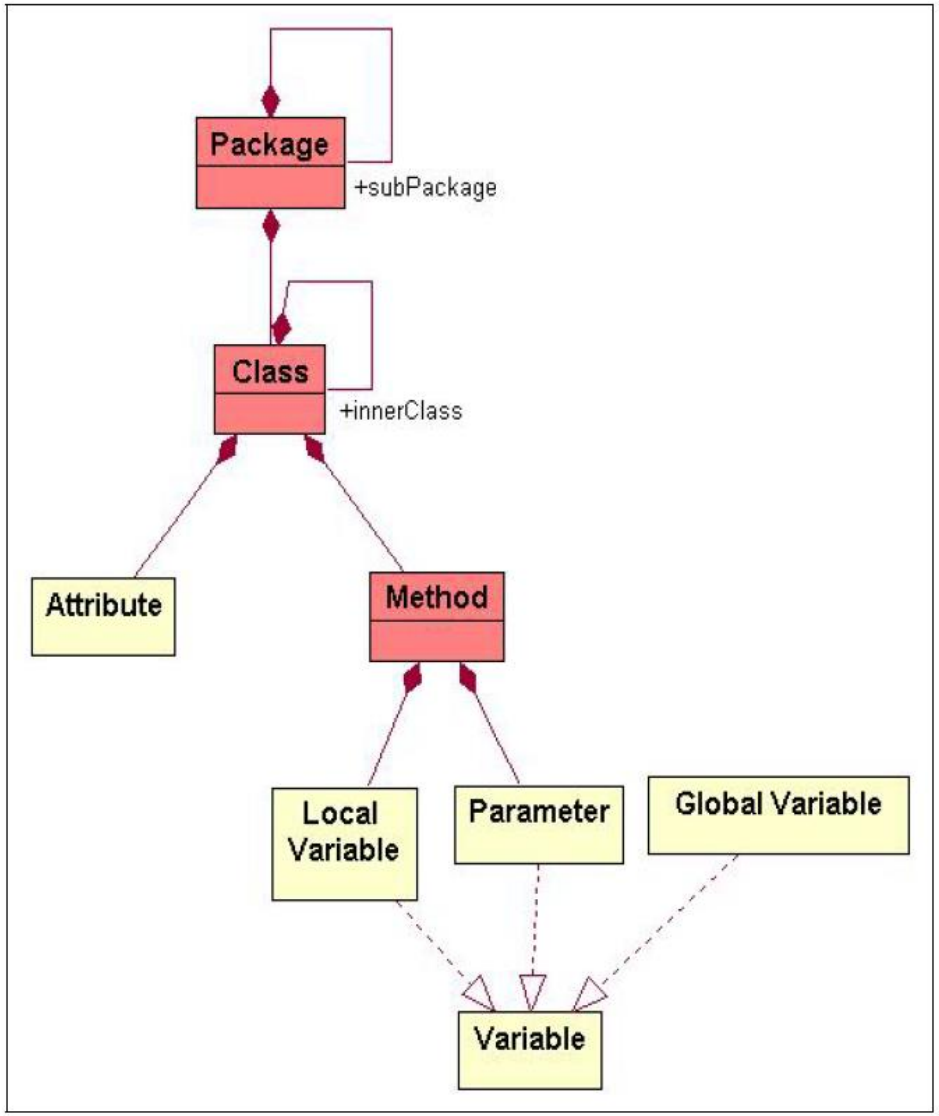
Meta-model pentru sistemele orientate obiect

- Meta-model= “Un model care descrie un model”
- Meta-Model [MarinescuR]= “A meta-model for an object-oriented system is a precise definition of the design entities and their types of interactions, used for defining and applying static analysis techniques.”
- Meta-Model pt sistemele orientate obiect:
    - Entitati de proiectare(e.g. clase, pachete)
    - Proprietati ale entitatilor de proiectare
    - Relatii intre entitatile de proiectare

Un meta-model pentru sistemele orientate obiect



[MarinescuR]

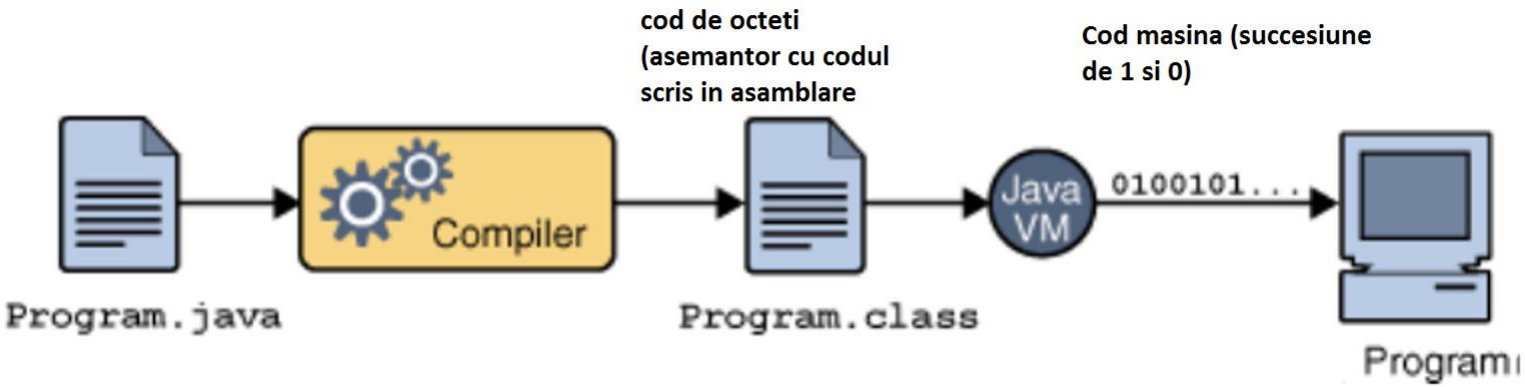


[MarinescuR]

Reamintim - Java este un limbaj compilat și interpretat

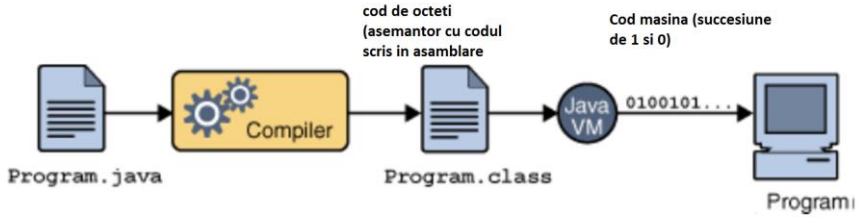
```
public class Program{
    public static void main(String[] args) {
        System.out.println("Hello");
}
```

```
javac Program.java   -> rezulta Program.class
java Program
```



Încărcarea claselor în memorie

- Execuția unei aplicatii Java este realizata de catre masina virtuala Java (JVM), aceasta fiind responsabila cu interpretarea codului de octeti (bytecode) rezultat în urma compilării.



- Un program Java compilat este descris de o mulțime de fișiere cu extensia .class corespunzătoare fiecărei clase a programului.
- Aceste clase nu sunt încarcate toate în memorie la pornirea aplicatiei, ci sunt încarcate pe parcursul execuției acestuia, atunci cand este nevoie de ele, momentul efectiv în care se realizează acest lucru depinzând de implementarea mașinii virtuale.

Etape în ciclul de viață al unei clase

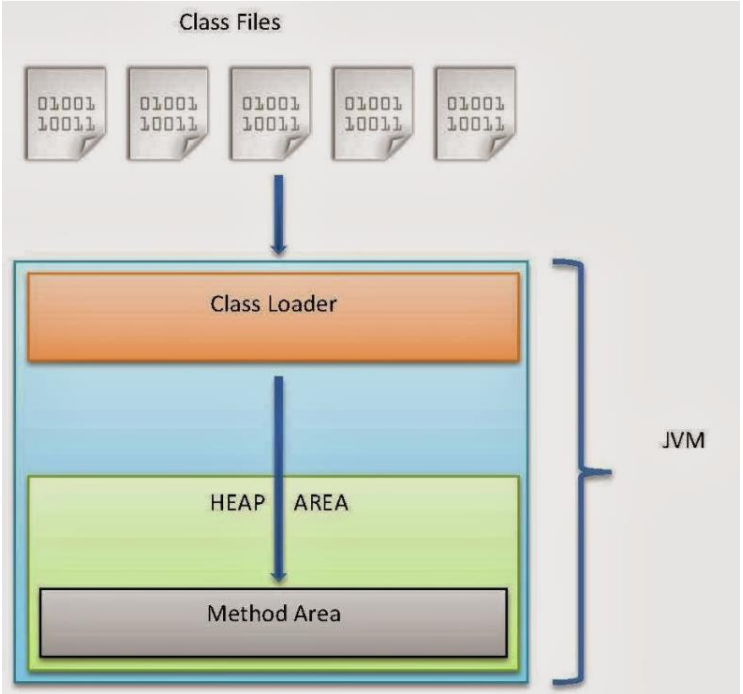
1. **Încarcarea în memorie** - va fi instanțiat un obiect de tipul `java.lang.Class`
2. **Editarea de legaturi** - încorporarea noului tip de date în JVM.
3. **Inițializarea –** execuția blocurilor statice de initializare și initializarea variabilelor de clasă.
4. **Descărcarea** - Atunci când nu mai 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*.

Încarcarea claselor în memorie

- Încarcarea claselor unei aplicatii Java în memorie este realizată prin intermediul unor obiecte, denumite generic *class loader.*

- Acestea sunt de doua tipuri:

1. Class loader-ul primordial (eng. bootstrap) - Reprezinta o parte integranta a masinii virtuale, fiind responsabil cu incarcarea claselor standard din distributia Java. (API)
2. Class loader-e proprii - Acestea nu fac parte intrinseca din JVM si sunt instante ale clasei java.lang.ClassLoader.



Încarcarea dinamică a claselor în memorie

- Se refera la faptul ca nu cunoastem tipul acesteia decat la executia programului, moment in care putem solicita încarcarea sa, specificand numele său complet prin intermediul unui șir de caractere.
- Exista mai multe modalitati:

  loadClass apelata pentru un obiect de tip ClassLoader

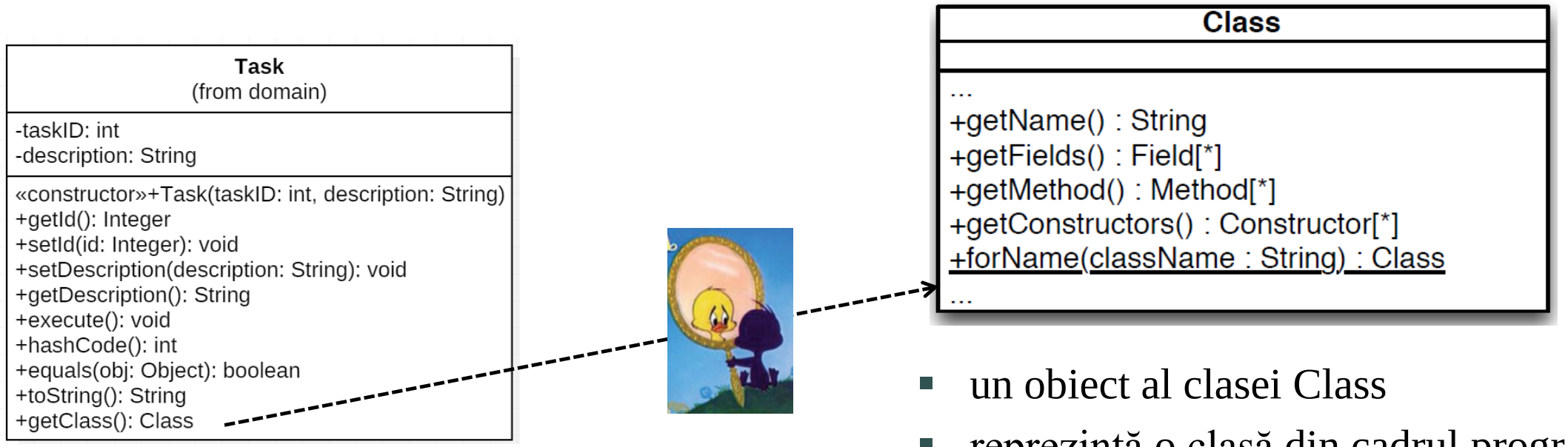
  Class.forName

Încarcarea unei clase în memorie

Meta clasa Class (java.lang.Class) - Class.forName

Exemplu: clasa **Task** este reprezentată la execuția unui program de un obiect instanță a acestei clase Class

```
Class aCls =Class.forName("domain.Task");
System.out.println(aCls.getName());
```



- un obiect al clasei Class
- reprezintă o clasă din cadrul programului

Observație

- În programarea orientată pe obiecte noțiunea de clasă și obiect sunt evident diferite
- În mecanismul de reflexie este doar o chestiune de reprezentare și nu se schimbă aceste noțiuni;
  - O clasă din program este reprezentată de un obiect și, ca orice obiect, e definit de o clasă (meta-clasă) în speță clasa Class.

Accesul la obiectele class

1. getClass():Class definită în Object

```
Task t=new Task(1,"ud florile");
Class taskClass=t.getClass();
```

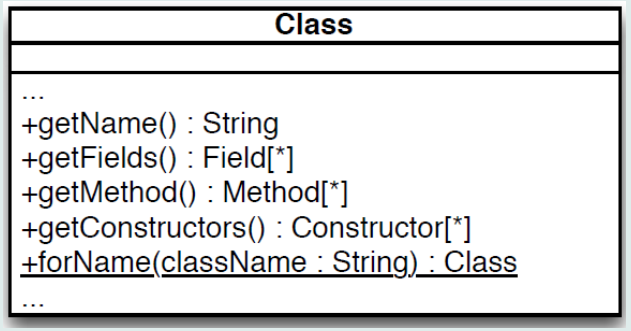
2) NumeClasa.class ex. Integer.class, int.class, Object.class, etc.

```
Class taskClass2=Task.class; //daca cunoastem numele clasei
System.out.println(taskClass2.getSimpleName());
```

3) Pt. clasele înfășurătoare există un câmp special TYPE ex. Integer.TYPE

```
Class integerClass=Integer.TYPE;
```

4) Class.forName(nume:String):Class




```
Class taskClass4=Class.forName("domain.Task");
```

bytecode-ul clasei trebuie să fie gasit la runtime în folderele specificate prin classpath altfel se arunca exceptia verificata ClassNotFoundException

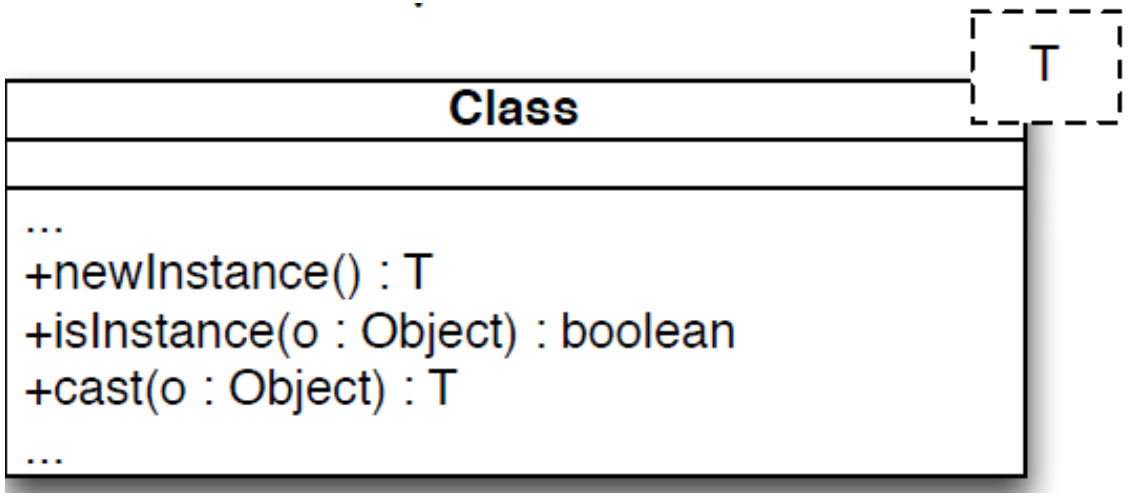
Referințe la obiectele Class

```
Class c; //poate referii orice obiect class
c = Integer.class;
c = Object.class;

Class<Integer> i; //numai obiectul class reprezentând clasa Integer
i = Integer.class;
i = int.class;
//i = Object.class; //Eroare de compilare

Class<Number> n; //numai obiectul class reprezentând clasa Number din biblioteca
//Java (superclasă pentru clasele înfășurătoare numerice)
n = Number.class;
//n = Integer.class; //Eroare de compilare
//n = Object.class; //Eroare de compilare

Class<? extends Number> nb; //obiectul class reprezentând clasa Number ori o
//subclasă a lui Number
nb = Number.class;
nb = Integer.class;    //nb = Object.class; -- Eroare de compilare
```

Metoda newInstance

newInstance

- crează o instanță a clasei respective (trebuie să aibă constructor no-arg altfel se aruncă excepție verificată)
- există variante și pentru cazul când avem numai constructori cu argumente - urmează

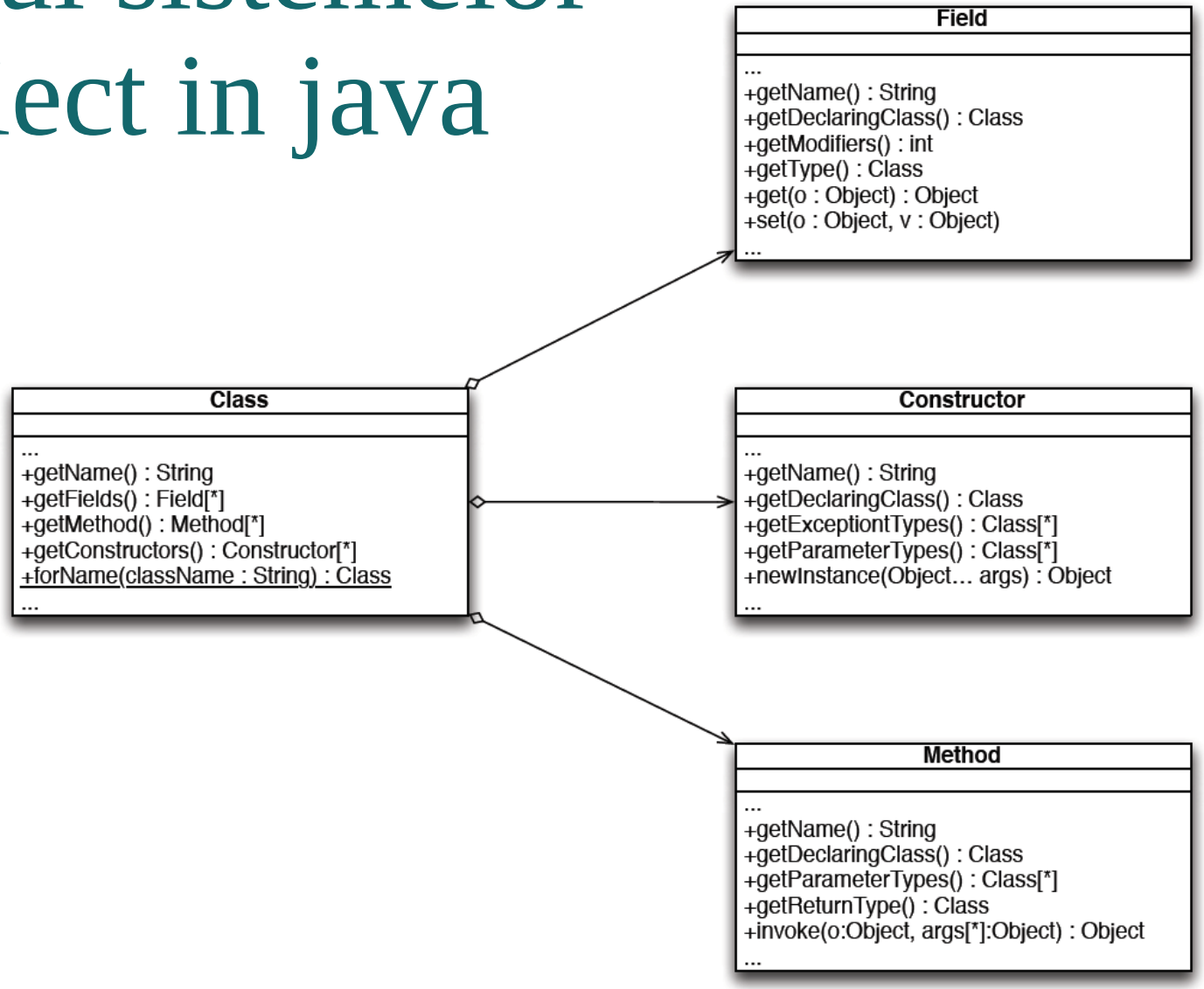



```
public Task(int taskID, String description) {
   setId(taskID);
   setDescription(description);
}
public Task() {} // ???? Daca il stergem
```

```
Class taskClass = Task.class;

Task aTask = (Task) taskClass.newInstance();

Class<Task> taskClassGeneric = Task.class;

Task atask = taskClassGeneric.newInstance();
```

Meta-modelul sistemelor orientate obiect in java

Conceptual și simplificat



API Reflection

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

Aflarea modificatorilor unei clase

```
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" + clasa.getName());
```

AccessibleObject

- Este o superclasa pe care Field, Method și Constructor o extind
- Acesta controlează accesul la variabile prin verificarea accesibilitatii unui camp (filed), a unei metode sau a unui constructor.

`isAccessible()` `setAccessible(boolean flag)`

```
Class a = Class.forName("domain.Student");

Method[] mm = a.getDeclaredMethods();
AccessibleObject.setAccessible(mm, true);
```

Aflarea superclasei

- Metoda getSuperclass
- Returneaza 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 unei clase

- **Variabile**: getFields, getDeclaredFields
- **Constructori**: getConstructors, getDeclaredConstructors
- **Metode**: getMethods, getDeclaredMethods
- **Clase imbricate**: getClasses, getDeclaredClasses

Încarcarea unei clase al carei nume se cunoaste doar la runtime

```java
/**
 * loads a class with a given name
 * @param className - name of loaded class
 * @return the corresponding Class object for the className
 */
public static Optional<Class> loadClass_forName(String className){
    Class aClass = null;
    try {
        aClass = Class.forName(className);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return Optional.ofNullable(aClass);
}
```

```java
Scanner sc=new Scanner(System.in);
String className=sc.nextLine();  // "domain.Task"
Optional<Class> cls=loadClass_forName(className);
```

Exemplu Reflectarea unei clase într-un fisier text

```
public static void reflectClass(Class aClass)
{
    String lines="";
    lines+=String.format( "Class "+ aClass.getName()+" having the following members:\n");

    for (Field aField : aClass.getDeclaredFields()) {
        lines+=String.format("\tField name -  %s :%s\n", aField.getName(),aField.getType());
    }
    for (Method aMethod : aClass.getMethods()) {
        lines+=String.format("\tMethod name -  %s (): %s\n",
         aMethod.getName(),aMethod.getReturnType().getName());
        Parameter[] param=aMethod.getParameters();
        for (int i = 0; i < param.length; i++) {
            lines+=String.format("\t\t Param  %d - %s:%s\n",i+1,param[i].getName(),param[i].getType());
        }
    }
    for (Constructor aConstructor : aClass.getConstructors())
    {
        lines+=String.format("\tConstructor name - %s:", aConstructor.getName());
        for (int i = 0; i < aConstructor.getParameters().length; i++) {
            Parameter param=aConstructor.getParameters()[i];
            lines+=String.format("\t\t Param -  %d: %s :%s\n", i+1,param.getName(),param.getType());
        }
    }
```

Reflectarea unei clase într-un fisier text

```
Path path= Paths.get("./src/data/"+aClass.getSimpleName()+"Mirror.txt");
try (BufferedWriter bufferedWriter = Files.newBufferedWriter(path,
        StandardOpenOption.CREATE)) {
    bufferedWriter.write(lines);
    bufferedWriter.newLine();
} catch (IOException e) {
    e.printStackTrace();
}


Scanner sc=new Scanner(System.in);
String className=sc.nextLine();
Optional<Class> cls=loadClass_forName(className);
if(cls.isPresent())
{
    reflectClass(cls.get());
}
```

TaskMirror.txt ☺

```
Class domain.Task having the following members:
   Field name -  taskID :int
   Field name -  description :class java.lang.String
   Method name -  equals (): boolean
       Param  1 - arg0:class java.lang.Object
   Method name -  toString (): java.lang.String
   Method name -  hashCode (): int
   Method name -  execute (): void
   Method name -  getId (): java.lang.Object
   Method name -  getId (): java.lang.Integer
   Method name -  setId (): void
       Param  1 - arg0:class java.lang.Integer
   Method name -  setId (): void
       Param  1 - arg0:class java.lang.Object
   Method name -  getDescription (): java.lang.String
   Method name -  setDescription (): void
       Param  1 - arg0:class java.lang.String
   Method name -  wait (): void
   Method name -  wait (): void
       Param  1 - arg0:long
       Param  2 - arg1:int
   Method name -  wait (): void
       Param  1 - arg0:long
   Method name -  getClass (): java.lang.Class
   Method name -  notify (): void
   Method name -  notifyAll (): void
   Constructor name - domain.Task:        Param -  1: arg0 :int
       Param -  2: arg1 :class java.lang.String
```

newInstance continuare

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

    public abstract int executa();
}
```

```
class Sort extends ArrayOperation {
    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;
    }
}
```

```
class Max extends ArrayOperation {
    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;
    }
}
```

newInstance

```
public static void runEx2ClassReflection() {
    BufferedReader br=new BufferedReader(
        (new InputStreamReader((System.in))));
    try {
      String opName=br.readLine();
      Class  operationClass=Class.forName("Examples."+opName);
      ArrayOperation op=(ArrayOperation)operationClass.newInstance();
      op.setVector(new int[] {100,200,30,40,50,60,71,80,90,91});
      op.executa();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    } catch (InstantiationException e) {
      e.printStackTrace();
    } catch (IllegalAccessException e) {
      e.printStackTrace();
    }
  }
```

newInstance și apelul unei metode

```
Class taskClass = Task.class;
try {
    Constructor<Task> cons = taskClass.getDeclaredConstructor(int.class, String.class);
    //create an instance of type task
    Task tt = null;
    try {
        tt = cons.newInstance(1, "fac sport");
        //call the toString method
        Method meth = taskClass.getDeclaredMethod("toString");
        Object result = meth.invoke(tt);
        System.out.println("Called toString, result: " + result);
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    }
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```

Apelul unei metode cont

```
Class clasa = java.awt.Rectangle.class;
Rectangle obiect = new Rectangle(0, 0, 100, 100);
Method metoda = null;
try {
    metoda = clasa.getMethod("contains", Point.class);
    System.out.println(metoda.toString());
    // Pregatim argumentele
    Point p = new Point(10, 20);
    // Apelam metoda
    Object res=metoda.invoke(obiect, p);
    System.out.println(res);
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
```

Lucru dinamic cu vectori

```
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) + " ");
```

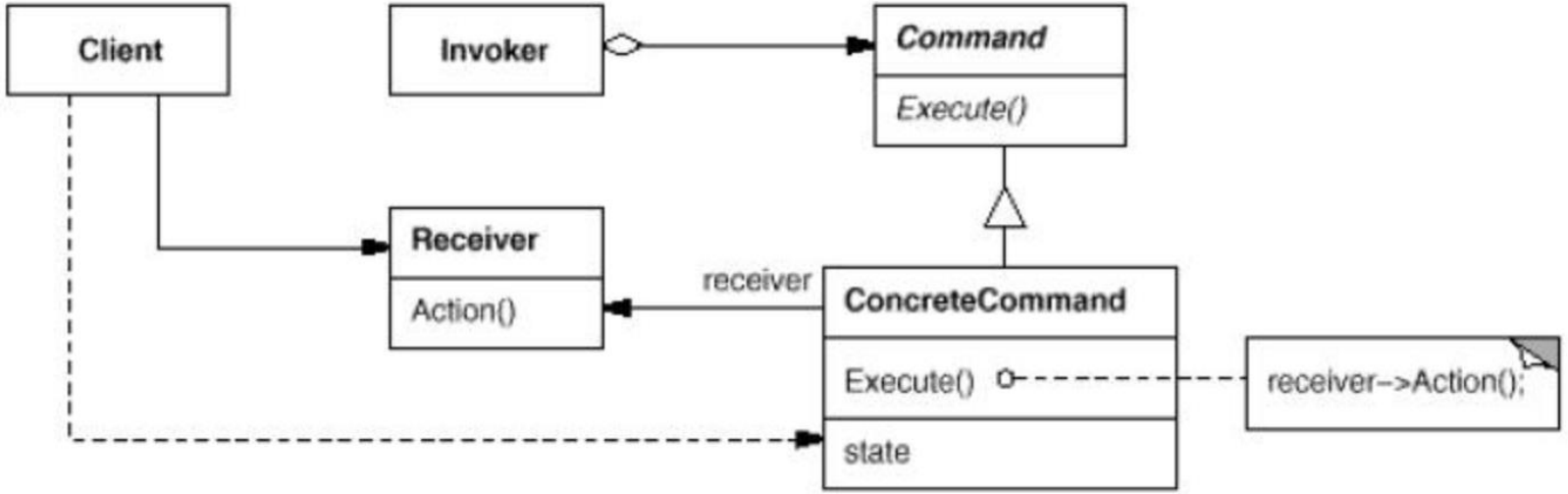
Reflexie. Avantaje și dezavantaje

- +:
- Class Browsers and Visual Development Environments
- Debuggers and Test Tools

- _:
- Performance Overhead
- Security Restrictions

Exemplu TextMenuCommand

Command Pattern

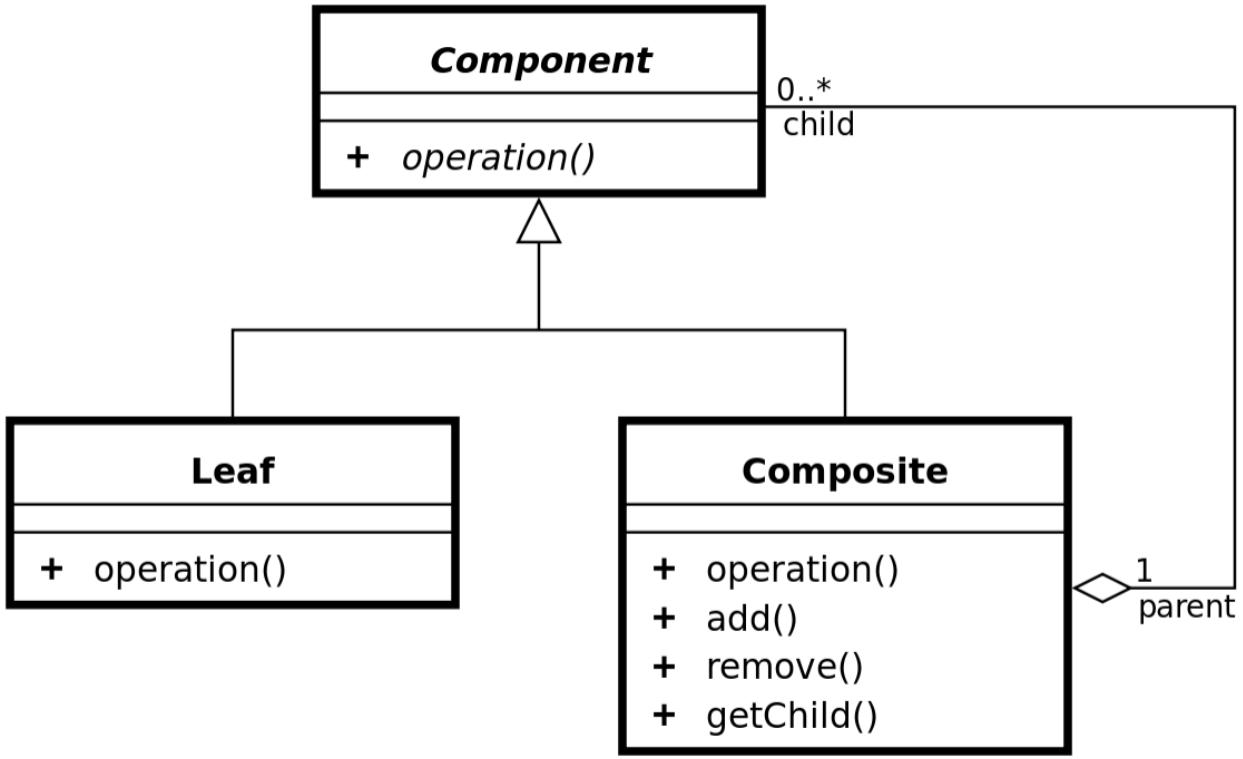


- Command
  - •obiectul comanda
- ConcreteCommand
  - •implementarea particulara a comenzii
  - •apeleaza metode ale obiectului receptor

- Invoker
  - •declanseaza comanda
- Receiver
  - •realizeaza, efectiv, operatiile aferente comenzii generate
- Client
  - •defineste obiectul comanda si efectul ei

Composite Pattern

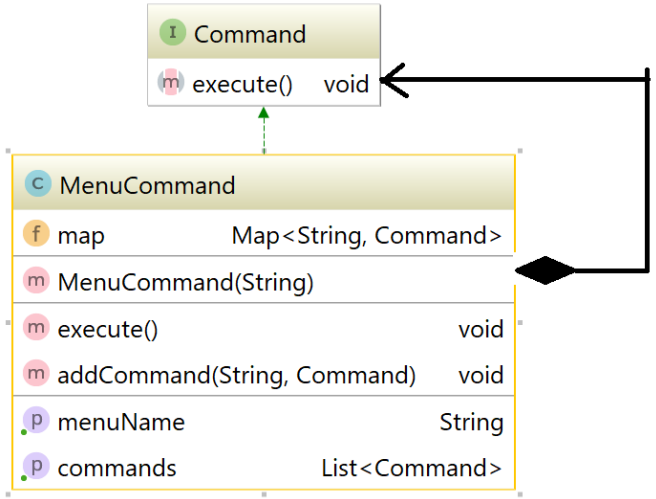
Compune mai multe obiecte similare a.i ele pot fi manipulate ca un singur obiect



TextMenuCommand

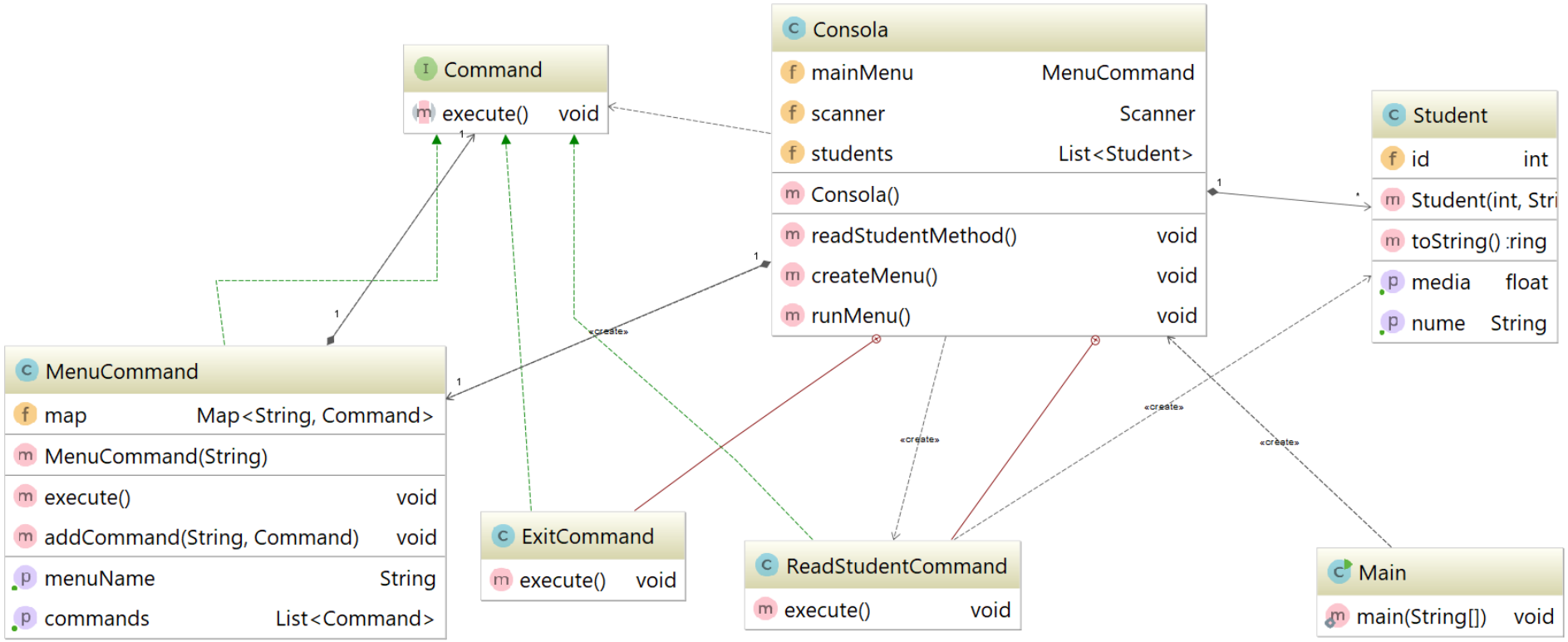
- O combinație de Command Pattern si Composite Patten

```
@FunctionalInterface
public interface Command{
    void execute();
}
```




```
public class MenuCommand implements Command {

    private String menuName;
    private Map<String, Command> map= new TreeMap<>();

    public MenuCommand(String menuName) {
        this.menuName = menuName;
    }
    @Override
    public void execute() {
        map.keySet().forEach(x-> System.out.println(x));
    }

    public void addCommand(String desc, Command c){
        map.put(desc, c);
    }

    public List<Command> getCommands(){
        return map.values().stream().collect(Collectors.toList());
    }

    public String getMenuName() {
        return menuName;
    }
}
```

TextMenuCommand



Cursul următor

- Interfețe grafice