Programare orientată pe obiecte

1. Moştenire (II)
2. Polimorfismul
2. Clasele Object şi Class

Clase abstracte

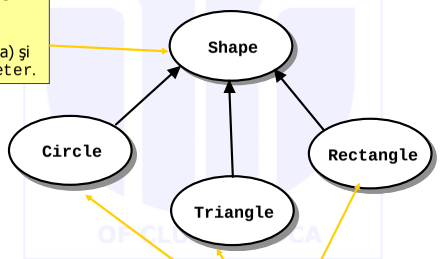
- O metodă sau o clasă abstractă se declară folosind cuvântul cheie `abstract`. De exemplu

```
public abstract double calcPay( double
hours );
```

- Dintr-o clasă abstractă nu se poate instanţia nici un obiect
- Fiecare subclasă a unei clase abstracte care va fi folosită pentru a instanţia obiecte trebuie să ofere implementări pentru toate metodele abstracte din superclasă.
- Clasele abstracte economisesc timp, deoarece nu trebuie să scriem cod "inutil" care n-ar fi executat niciodată.
- O clasa abstractă poate moşteni metode *abstracte*
  - dintr-o interfaţă sau
  - dintr-o clasă.

Exemplu: O clasă numită **Shape** (formă)

Superclasă: conţine metodele abstracte `calculateArea` (calculează suprafaţa) şi `calculatePerimeter`.



Subclase: implementează metodele concrete `calculteArea` şi `calculatePerimeter`.

Exemplu: O clasă numită **Shape**

```
/**
 * Abstract class Shape - base for inheritance for shapes
 */
public abstract class Shape {
    private static int counter;
    // Constructor
    public Shape()      {
        counter++;
    }
    // calculate area
    public abstract double calculateArea();
    // calculate perimeter
    public abstract double calculatePerimeter();
    // get number of shapes
    public int getCount() {
        return counter;
    }
    protected void finalize() throws Throwable {
        counter--;
    }
}
```

Definiţia superclasei. Observaţi că această clasă este declarată `abstract`.

Definiţii de metode abstracte. Observaţi că este declarat doar antetul. Aceste metode **trebuie suprascrise (overridden)** în toate clasele concrete.

Exemplu: subclasa `Circle`

```
/**
 * Concrete class Circle - inherits from Shape
 */
public class Circle extends Shape {
    private double r; // radius of circle
    // Constructor
    public Circle(double r)     {
        super();
        this.r = r;
    }
    // calculate area
    public double calculateArea()   {
        return  Math.PI * r * r;
    }
    // calculate perimeter
    public double calculatePerimeter()   {
        return 2.0 * Math.PI * r ;
    }
     protected void finalize() throws Throwable    {
        super.finalize();
    }
}
```

Clasă concretă. Clasa *nu trebuie* să conţină sau să moştenească metode abstracte. Metodele abstracte moştenite trebuie suprascrise.

Definiţii de metode concrete. Observaţi că aici este declarat corpul metodei.

Exemplu: subclasa `Triangle`

```
/**
 * Concrete class Triangle - inherits from Shape
 */
public class Triangle extends Shape {
    private double s; // side of Triangle
    // Constructor
    public Triangle(double s)     {
        super();
        this.s = s;
    }
    // calculate area
    public double calculateArea()   {
        return ( Math.sqrt(3.)/4 * s *s );
    }
    // calculate perimeter
    public double calculatePerimeter()   {
        return 3.0 * s ;
    }
     protected void finalize() throws Throwable {
        super.finalize();
    }
}
```

Clasă concretă. Clasa *nu trebuie* să conţină sau să moştenească metode abstracte. Metodele abstracte moştenite trebuie suprascrise.

Definiţii de metode concrete. Observaţi că corpurile metodelor sunt diferite de cele din `Circle`, dar semnăturile metodelor sunt *identice*.

Alte subclase ale lui **Shape** vor suprascrie şi ele metodele abstracte **`area`** şi **`perimeter`**

Exemplu: clasa `TestShape`

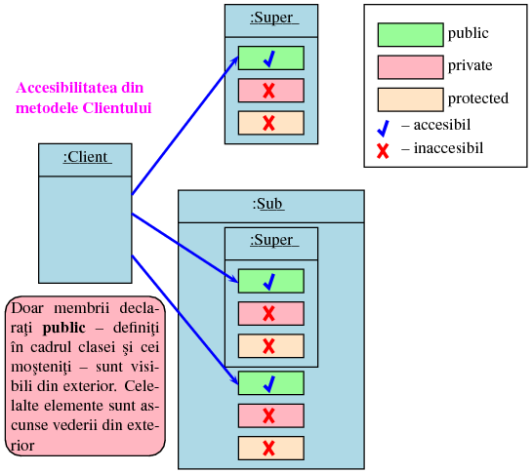
```
/**
 * Write a description of class TestShape here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class TestShape
{
    public static void main(String[] args)
    {
        // Create an array of Shapes
        Shape s[] = new Shape[2];
        // create objects
        s[0] = new Circle(2);
        s[1] = new Triangle(2);
        // Print out the number of Shapes
        System.out.println(s[0].getCount() + " shapes created");
        for (int i = 0; i < s.length; i++ ) {
            System.out.print(s[i].toString() + " ");
            System.out.print("Area = " + s[i].calculateArea());
            System.out.println(" Perimeter = " + s[i].calculatePerimeter());
        }
    }
}
```

Creează obiecte ale subclaselor folosind referinţe la superclasă.

Apelează metodele `area` şi `perimeter`. Este apelată automat versiunea corespunzătoare a fiecărei metode pentru fiecare obiect.

Variabile instanţă

- Ca şablon general, subclasele:
  - Moştenesc capabilităţile **`public`** (metode)
  - Moştenesc proprietăţile **`private`** (variabile instanţă) dar nu au acces la ele
  - Moştenesc variabilele **`protected`** şi le pot accesa
- O variabila declarată **`protected`** de o superclasă devine *parte a moştenirii*
  - variabila devine disponibilă pentru subclase, care o pot accesa *ca şi cum ar fi proprie*
  - spre deosebire de aceasta, dacă o variabilă instanţă este declarată **`private`** într-o superclasă, subclasele nu vor avea acces la ea
    - superclasa poate totuşi oferi acces protejat la variabilele instanţă private via metode *accesoare* şi *mutatoare*

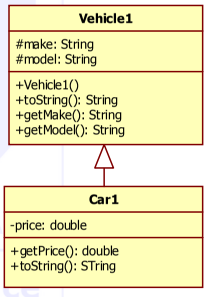


Variabile instanţă `protected` faţa de variabile instanţă `private`

- Cum putem decide între **`private`** şi **`protected`**?
  - folosiţi **`private`** dacă doriţi ca o variabilă instanţă să fie *încapsulată* de către superclasă
    - d.e., uşile, ferestrele, bujiile unei maşini
  - folosiţi **`protected`** dacă doriţi ca variabila instanţă să fie accesibilă subclaselor pentru a o modifica (şi nu doriţi să faceţi variabila mai general accesibilă prin metode accesoare/mutatoare)
    - d.e., motorul unei maşini

`protected`, Exemplu

```
public class Vehicle1 {
  protected String make;
  protected String model;
  public Vehicle1() { make = ""; model = "";}
  public String toString() {
    return "Make: " + make + " Model: " + model;
  }
  public String getMake(){ return make;}
  public String getModel() { return model;}
}
public class Car1 extends Vehicle1 {
  private double price;
  public Car1() { price = 0.0; }
  public String toString() {
    return "Make: " + make + " Model: " + model
      + " Price: " + price;
  }
  public double getPrice(){ return price; }
}
```



Suprascrierea unei definiţii de metodă

- Deşi o clasă derivată moşteneşte metode din clasa de bază, ea poate să le modifice – să le *suprascrie* dacă este necesar
  - Pentru a suprascrie o definiţie de metodă, se pune pur şi simplu o definiţie nouă în definiţia clasei, exact ca pentru orice altă metodă adăugată clasei derivate
- De obicei, tipul returnat nu poate fi schimbat la suprascrierea unei metode
- Totuşi, dacă tipul este un *tip clasă*, atunci tipul returnat poate fi schimbat la acela al oricărei *clase descendente* al tipului returnat
- Acest lucru se cunoaşte sub numele de *tip returnat covariant*
  - *Tipurile returnate covariant* sunt introduse în Java 5.0; ele nu sunt permise în versiuni anterioare de Java

Tipul returnat covariant

- Fiind dată următoarea clasă de bază:

```
public class BaseClass
{ . . .
  public Employee getSomeone(int someKey)
  . . .
```

- Este permisă următoarea modificare a tipului returnat în Java 5.0:

```
public class DerivedClass extends BaseClass
{ . . .
  public HourlyEmployee getSomeone(int someKey)
  . . .
```


Schimbarea permisiunii de acces a unei metode suprascrise

- Permisiunea de acces a unei metode suprascrise poate fi schimbată *de la private* în *clasa de bază* la *public* (sau alt *acces mai permisiv*) în *clasa derivată*
- Totuşi, permisiunea de acces a unei metode suprascrise *nu poate fi modificată* de la public în clasa de bază *la o permisiune de acces mai restrictivă* în clasa derivată
  - Adică, putem relaxa permisiunile de acces într-o clasă derivată, nu o putem restrânge

Schimbarea permisiunii de acces a unei metode suprascrise

- Fiind dat următorul antet de metodă într-o clasă de bază:
  `private void doSomething()`
- Următorul antet de metodă este valid într-o clasă derivată:
  `public void doSomething()`
- Invers (din public în privat) nu se poate
- Fiind dat următorul antet de metodă într-o clasă de bază:
  `public void doSomething()`
- Antetul de metodă următor *nu* este valid într-o clasă derivată:
  `private void doSomething()`

Capcană: Suprascriere faţa de supraîncărcare

- Nu confundaţi *suprascrierea (overriding)* unei metode într-o clasă derivată cu *supraîncărcarea (overloading)* numelui unei metode
  - Când o metodă este *suprascrisă*, noua definiţie de metodă dată în clasa derivată are *exact acelaşi număr şi tipuri de parametri ca în clasa de bază*
  - Când o metodă este într-o clasă derivată are o *semnătură diferită* în comparaţie cu metoda din clasa de bază, atunci avem de-a face cu *supraîncărcarea*
  - Observaţi că atunci când *clasa derivată suprascrie* metoda originală, *ea totuşi moşteneşte şi metoda originală* din clasa de bază

Modificatorul `final`

- Dacă se pune modificatorul `final` în faţa definiţiei unei *metode*, atunci metoda respectivă *nu poate fi redefinită* într-o clasă derivată
- Dacă modificatorul `final` este pus în faţa definiţiei unei *clase*, atunci clasa respectivă *nu mai poate fi folosită pe post de clasă de bază* pentru a deriva alte clase

Constructorul `super`

- O clasă derivată foloseşte un constructor al clasei de bază pentru a iniţializa toate datele moştenite din clasa de bază
  - Pentru a invoca un constructor al clasei de bază, se foloseşte o sintaxă specială:

```
public DerivedClass(int p1, int p2, double p3)
{
  super(p1, p2);
  instanceVariable = p3;
}
```

  - În exemplul de mai sus, `super(p1, p2);` este un apel al constructorului clasei de bază

Constructorul **super**

- Un apel al unui constructor al clasei de bază nu poate folosi numele clasei respective, ci foloseşte în schimb cuvântul cheie **super**
- Apelul lui **super** trebuie să fie întotdeauna prima acţiune efectuată în definiţia unui constructor
- Nu se pot folosi *variabile instanţă* ca argumente ale lui **super**
  - Oare de ce acest lucru nu este permis?

Constructorul **super**

- Dacă într-o clasă derivată nu este prezentă o invocare a lui **super**, atunci constructorul fără argumente al clasei de bază va fi apelat automat
  - Aceasta poate cauza o eroare dacă clasa de bază nu are definit un constructor fără argumente
- Cum variabilele instanţă moştenite ar trebui iniţializate, iar menirea constructorului clasei de bază este să facă acest lucru, *întotdeauna ar trebui folosit un apel explicit al lui* **super**

Constructori şi subclase

- Cuvântul cheie **this** este folosit pentru a invoca un alt constructor al aceleiaşi clase. Exemplu:

```
public class Circle extends ClosedFigure
{
  private int radius;
  public Circle(int radius)
  {
    super();
    this.radius = radius;
  }
  public Circle()
  {
    this(1); // invoca constructorul cu argumentul 1
  }
…
}
```


Accesul la o metodă redefinită din clasa de bază

- În definiţia unei metode dintr-o clasă derivată, versiunea suprascrisă a unei metode a clasei de bază poate totuşi fi invocată
  - Pur şi simplu prefixaţi numele metodei cu **super** şi un punct

```
public String toString()
{
  return (super.toString() + "$" + wageRate);
}
```

- Cu toate acestea, la folosirea unui obiect al clasei derivate în afara definiţiei clasei, nu există nici o cale de invocare a versiunii unei metode suprascrise din clasa sa de bază

Nu puteţi folosi mai mulţi **super**

- **super** poate fi folosit pentru a invoca o metodă doar din-tr-un părinte (strămoş direct)
  - Repetarea lui **super** nu va invoca o metodă din vreo altă clasă strămoş
- Spre exemplu, dacă clasa **Employee** ar fi fost derivată din clasa **Person**, iar clasa **HourlyEmployee** ar fi fost derivată din clasa **Employee** , nu ar fi fost posibil să invocăm metoda **toString** a clasei **Person** dintr-o metodă a clasei **HourlyEmployee**

```
super.super.toString() // ILEGAL!
```


Constructorul **this**

- În definiţia unui constructor pentru o clasă, **this** poate fi folosit ca nume pentru invocarea unui alt constructor din aceeaşi clasă
  - Se aplică aceleaşi restricţii de folosire ca pentru **super** şi pentru **this**
- Daca este necesar să fie apelat atât **super** cât şi **this**, trebuie efectuat mai întâi apelul care foloseşte **this**, iar apoi constructorul invocat cu **this** trebuie să invoce **super** ca primă acţiune a sa

Constructorul `this`

- Adesea, un constructor fără argumente foloseşte `this` pentru a invoca un constructor cu valori explicite
  - Constructor fără argumente (invocă un constructor cu valori explicite folosind `this` şi argumente implicite):

```
public ClassName(){
  this(argument1, argument2);
}
```

  - Constructor cu valori explicite (primeşte valori implicite):

```
public ClassName(type1 param1, type2 param2){
  . . .
}
```


Constructorul `this`

```
public HourlyEmployee()
{
  this("No name", new Date(), 0, 0);
}
```

- Constructorul de mai sus va determina invocarea constructorului cu următorul antet:

```
public HourlyEmployee(String theName, Date
  theDate, double theWageRate, double
  theHours)
```


Un obiect al unei clase derivate are mai mult de un tip

- Un obiect al unei clase derivate are tipul clasei derivate şi are şi tipul clasei de bază
- Mai general, un obiect al unei *clase derivate* are *tipul fiecăruia dintre clasele din ascendenţa sa*
  - De aceea, un obiect dintr-o clasă derivată poate fi asignat unei variabile de tipul oricărui părinte/strămoş al său

Un obiect al unei clase derivate are mai mult de un tip

- Un obiect al unei clase derivate poate fi folosit ca parametru în locul oricăruia dintre clasele părinte/strămoş ale sale
- De fapt, un obiect dintr-o clasă derivată poate fi folosit în orice loc în care se poate folosi un obiect de tipurile părintelui/strămoşilor săi
- Observaţi, totuşi, că relaţia nu merge şi invers
  - Un tip strămoş/părinte nu poate fi niciodată folosi în locul unuia dintre tipurile derivate din el

Capcană: termenii "subclasă" şi "superclasă"

- Uneori termenii *subclasă* şi *superclasă* sunt inversaţi din greşeală
  - O *superclasă* / clasă de bază este *mai generală* şi mai cuprinzătoare, dar *mai puţin complexă*
  - O *subclasă* / clasă derivată este *mai specializată,* mai puţin cuprinzătoare şi *mai complexă*
    - Pe măsură ce sunt adăugate mai multe variabile şi metode, numărul obiectelor care pot satisface definiţia clasei devine mai restrâns

Folosirea claselor abstracte

- O clasă abstractă contribuie la implementarea subclaselor sale concrete.
- Este folosită pentru a exploata polimorfismul.
  - Pentru funcţionalitatea specificată în clasele părinte se pot da implementări corespunzătoare fiecărei subclase concrete.
- Clasele abstracte trebuie să fie stabile.
  - Orice schimbare într-o clasă abstractă se propagă la subclase şi la clienţii lor.
- O clasă concretă poate extinde doar o singură clasă (abstractă)

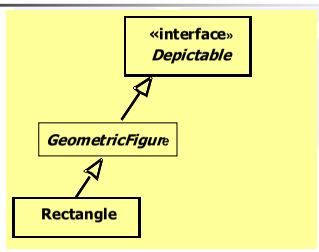
Interfeţe, clase abstracte şi clase concrete

- O ***interfaţă***
  - se foloseşte pentru a specifica funcţionalitatea cerută de un client.
- O ***clasă abstractă***
  - oferă o bază pe care să se construiască clase server concrete.
- O ***clasă concretă***
  - completează implementarea server-ului care a fost specificată de o interfaţă;
  - furnizează obiecte la momentul execuţiei;
  - nu este, în general, potrivită ca bază pentru extindere.

Folosirea interfeţelor

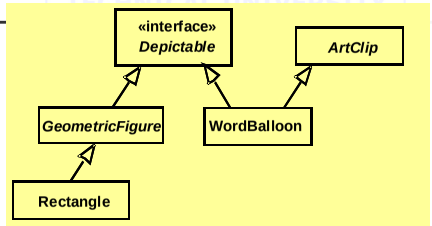
- Interfeţele sunt abstracte prin definiţie.
  - separă implementarea unui obiect de specificarea sa.
  - nu fixează nici un aspect al unei implementări.
- O clasă poate implementa mai mult de o interfaţă.
- Interfeţele permit o folosire mai generalizată a polimorfismului; instanţe din clase relativ neînrudite pot fi tratate ca identice într-un scop anume.
- În programe, folosiţi
  - *interfeţe pentru a partaja comportament comun.*
  - *moştenirea pentru a partaja cod comun.*

Interfeţe, clase abstracte şi clase concrete




```
public boolean isIn (Location point, Depictable figure)
{
        Location l = figure.getLocation();
        Dimension d = figure.getDimension();
        …
}
```


Interfeţe, clase abstracte şi clase concrete




```
public boolean isIn (Location point, Depictable figure)
{
        Location l = figure.getLocation();
        Dimension d = figure.getDimension();
        …
}
```

- Pot fi folosite instanţe ale lui **`WordBalloon`** ca parametri ai lui **`isIn`**.

Exemplu de polimorfism

```
public class Base {
   protected int theInt = 100;

   ...
   public void printTheInt() {
      System.out.println( theInt );
   }
```

```
public class Doubler extends Base {
   ...
   public void printTheInt() {
      System.out.println( theInt*2 );
   }
```

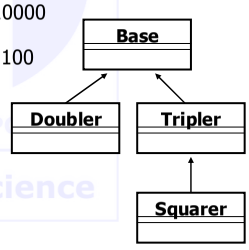
```
public class Tripler extends Base {
   ...
   public void printTheInt() {
      System.out.println( theInt*3 );
   }
```

```
public class Squarer extends Tripler {
   ...
   public void printTheInt() {
            System.out.println( theInt*theInt );
   }
}
```


Exemplu de polimorfism

```
Base theBase;
theBase = new Base();
theBase.printTheInt();
theBase = new Doubler();
theBase.printTheInt();
theBase = new Tripler();
theBase.printTheInt();
theBase = new Squarer();
theBase.printTheInt();
theBase = new Base();
theBase.printTheInt();
...
```

→ 100
→ 200
→ 300
→ 10000
→ 100



Polimorfismul subtipurilor

Pe măsură ce apare legarea dinamică comportamentul (adică metodele) urmează obiectele

Polimorfism

- O variabilă polimorfică poate părea a-şi schimba tipul prin legare dinamică.
- Compilatorul înţelege întotdeauna tipul unei variabile potrivit declaraţiei.
- Compilatorul permite o anume flexibilitate prin modul de conformare la tip.
- La execuţie, comportamentul unui apel de metodă depinde de tipul de *obiect*, nu de variabilă.
- Exemplu:

```
Base theBase;
theBase = new Doubler();
theBase = new Squarer();
theBase.printTheInt();
```


De ce este util polimorfismul?

- Polimorfismul permite unei superclase sa reţină ceea ce este comun, lăsând specificitatea sa fie tratata de subclase.

Sa presupunem ca AView include o metoda **area**, ca mai sus.

```
public class AView {
  ...
  public double calcArea() {
        return 0.0;
  }
}
```

Atunci ARectangle trebuie scris ca ...

```
public class ARectangle extends AView {
  ...
  public double calcArea() {
        return getWidth() * getHeight();
  }
}
```

iar AOval trebuie scris ca ...

```
public class AOval extends AView {
  ...
  public double calcArea() {
        return getWidth()/2. * getHeight()/2. * Math.PI;
  }
}
```

Consideraţi acum

```
public double coverageCost( AView v, double costPerSqUnit) {
        return v.area() * costPerSqUnit;
}
```


Cum se decide care este metoda de executat

1. Dacă există o metodă concreta în clasa curenta, se executa aceea.
2. În caz contrar, se verifică în superclasa directă daca există acolo o metodă; dacă da, se execută.
3. Se repetă pasul 2, verificând în sus pe ierarhie până când se găseşte o metodă concretă şi se execută.
4. Dacă nu s-a găsit nici o metodă, atunci este eroare
   - În Java şi C++ programul nu se compilează

Legarea dinamică

- Apare atunci când decizia privind metoda de executat nu se poate lua decât la execuţia programului
  - Este nevoie de ea atunci când:
    - Variabila este declarată ca având tipul superclasei şi
    - Există mai mult de o metoda polimorfică care se poate executa între tipul variabilei şi subclasele sale

Moştenirea din clasa `Object`

- *Fiecare clasă din Java extinde o alta clasă*.
- Dacă nu specificaţi explicit clasa pe care clasa Dvs. o extinde, atunci se va extinde automat clasa numită ***`Object`***.
- Toate clasele în Java sunt într-o ierarhie în care clasa numita *Object* este rădăcina ierarhiei.
- Unele clase extind *Object* direct, în timp ce altele sunt subclase ale lui *Object* mai jos în ierarhie.
- Clasa *Object* este definită în **`java.lang`**

Metode din clasa `Object`

- *Object* defineşte versiuni implicite ale următoarelor metode:
  - **`toString()`** – returnează un **`String`** (reprezentare "citibilă" a obiectului)
  - **`equals(Object obj)`** – trebuie să fie suprascrisă pentru egalitatea de conţinut
  - **`hashCode()`** – returnează valoarea codului de dispersie pentru obiect; valorile sunt diferite pentru obiecte diferite
  - **`getClass()`** – returnează un obiect de tipul Class; există un obiect de tipul Class pentru fiecare clasă dintr-o aplicaţie
  - **`notify(), notifyAll(), wait(), wait(long timeout), wait(long timeout, int nanos)`** – folosite la multithreading
  - **`clone()`** – creează şi întoarce o copie a acestui obiect. Semnificaţie lui "copie" poate depinde de clasa obiectului.
  - **`finalize()`** – destinat a efectua acţiuni de "curăţare" înainte ca obiectul să fie irevocabil abandonat.

Egalitatea

- Există două feluri diferite de egalitate:
  - *Egalitatea de identitate* care înseamnă că două expresii au aceeaşi identitate. (Adică reprezintă acelaşi obiect.)
  - *Egalitatea de conţinut* care înseamnă că două expresii reprezintă obiecte cu aceeaşi valoare/conţinut.

Simbolul == testează egalitatea de identitate atunci când este aplicat datelor referinţă.

- Exemplu:

```
AOval ov1, ov2;
ov1 = new AOval(0, 0, 100, 100);
ov2 = new AOval(0, 0, 100, 100);
if (ov1 == ov2){ System.out.println("identity equality");}
 else {System.out.println("content equality");}
```

- Metoda *equals* din **Object** poate fi folosită pentru a verifica egalitatea de conţinut.

Clasa Class

- Clasa **Class** este definită astfel:

```
public final class Class extends Object
    implements Serializable, …
```

- Instanţele clasei **Class** reprezintă clase şi interfeţe dintr-o aplicaţie Java în curs de execuţie.
- Un obiect de tipul **Class** conţine informaţii despre clasa a cărei instanţe este obiectul care apelează
- Nu are constructor propriu
- Obiectele **Class** sunt construite la execuţie de către JVM
- Există două moduri pentru a construi obiecte de acest tip:
  - **getClass()** din clasa **Object**
  - **forName()** din clasa **Class** (metodă statică)

Clasa Class

- Metode:
  - **public String getName()**
    - returnează un **String** care reprezintă numele entităţii reprezentate de obiectul **Class this**
    - entitatea poate fi: clasă, interfaţă, tablou, tip primitiv, void
  - **public static Class forName(String className) throws ClassNotFoundException**
    - returnează un obiect de tipul **Class** care conţine informaţii despre clasa obiectului
  - **public Class[] getClasses()**
    - returnează un tablou de obiecte de tip **Class**;
    - toate clasele şi interfeţele, membri publici ai clasei reprezentate de acest obiect **Class**

Clasa Class

- Metode (continuare)
  - **Field[] getFields**
    - returnează un tablou care conţine obiecte **Field** care reflectă toate câmpurile accesibile public ale clasei sau interfeţei reprezentate de acest obiect **Class**
  - **Method[]getMethods()**
    - returnează un tablou care conţine obiecte **Method** care reflectă toate metodele publice *membre* ale clasei sau interfeţei reprezentate de acest obiect **Class**, inclusiv cele declarate de clasă sau interfaţă şi cele moştenite din superclase şi superinterfeţe.
  - **Constructor[]getConstructors()**
    - returnează un tablou care conţine obiecte **Constructor** care reflectă toţi constructorii publici ai clasei sau interfeţei reprezentate de acest obiect **Class**.

Operatorul instanceof

- Operatorul **instanceof** verifică dacă un obiect este de tipul dat ca al doilea argument al său

```
Obiect instanceof NumeClasa
```

  - Va returna **true** dacă **Obiect** este de tipul **NumeClasa**; altfel va returna **false**
  - Observaţi că aceasta înseamnă că va returna **true** dacă **Obiect** are tipul *oricărei clase care este descendentă* a lui **NumeClasa**

Metoda getClass()

- Fiecare obiect moşteneşte aceeaşi metodă **getClass()** din clasa **Object**
  - Această metodă este marcată **final**, deci nu poate fi suprascrisă
- O invocare a lui **getClass()** pe un obiect returnează o reprezentare *numai* pentru clasa care a fost folosită cu operatorul **new** pentru a crea obiectul
  - Rezultatele a oricare două asemenea invocări pot fi comparate cu **==** sau **!=** pentru a determina dacă ele reprezintă sau nu aceeaşi clasă

```
(obiect1.getClass() == obiect2.getClass())
```

`instanceof` şi `getClass()`

- Atât operatorul **`instanceof`** cât şi metoda **`getClass()`** se pot folosi pentru a verifica clasa unui obiect
- Totuşi, metoda **`getClass()`** este mai exactă
  - Operatorul **`instanceof`** doar testează clasa unui obiect
  - Metoda **`getClass()`** folosită într-un test cu **`==`** or **`!=`** testeza dacă doua obiecte *au fost create* din aceeaşi clasă

Exemple

- Afişarea numelui unei clase folosind un obiect de tip **`Class`**

```
void printClassName(Object obj) {
   System.out.println(obj + " is of class " +
   obj.getClass().getName());
}
```

- Alte exemple

```
Circle c = new Circle(5);
printClassName(c);
Class c1 = c.getClass();
System.out.println(c1.getName()); // tipareste "Circle"
Triangle t = new Triangle(7);
printClassName(t);
try {
   Class c2 = Class.forName("Triangle");
   System.out.println(c2.getName()); // tipareste "Triangle"
}
catch (ClassNotFoundException e) {
    System.err.println("No class for \"Triangle\"" +
  e.getMessage());
}
```


Rezumat

- Moştenire
  - clasă de bază, superclasă
  - clasă derivată, subclasă
  - ierarhii
  - constructorii **`super`** şi **`this`**
  - accesul la variabilele instanţă
  - suprascrierea vs supraîncărcarea unei metode
- Interfeţe, clase abstracte, clase concrete
- Polimorfism
- Clasa **`Object`**
  - toate moştenesc din clasa Object
  - metode de suprascris
- Clasa **`Class`**
  - metode utile
- Operatorul **`instanceof`** vs metoda **`getClass()`**