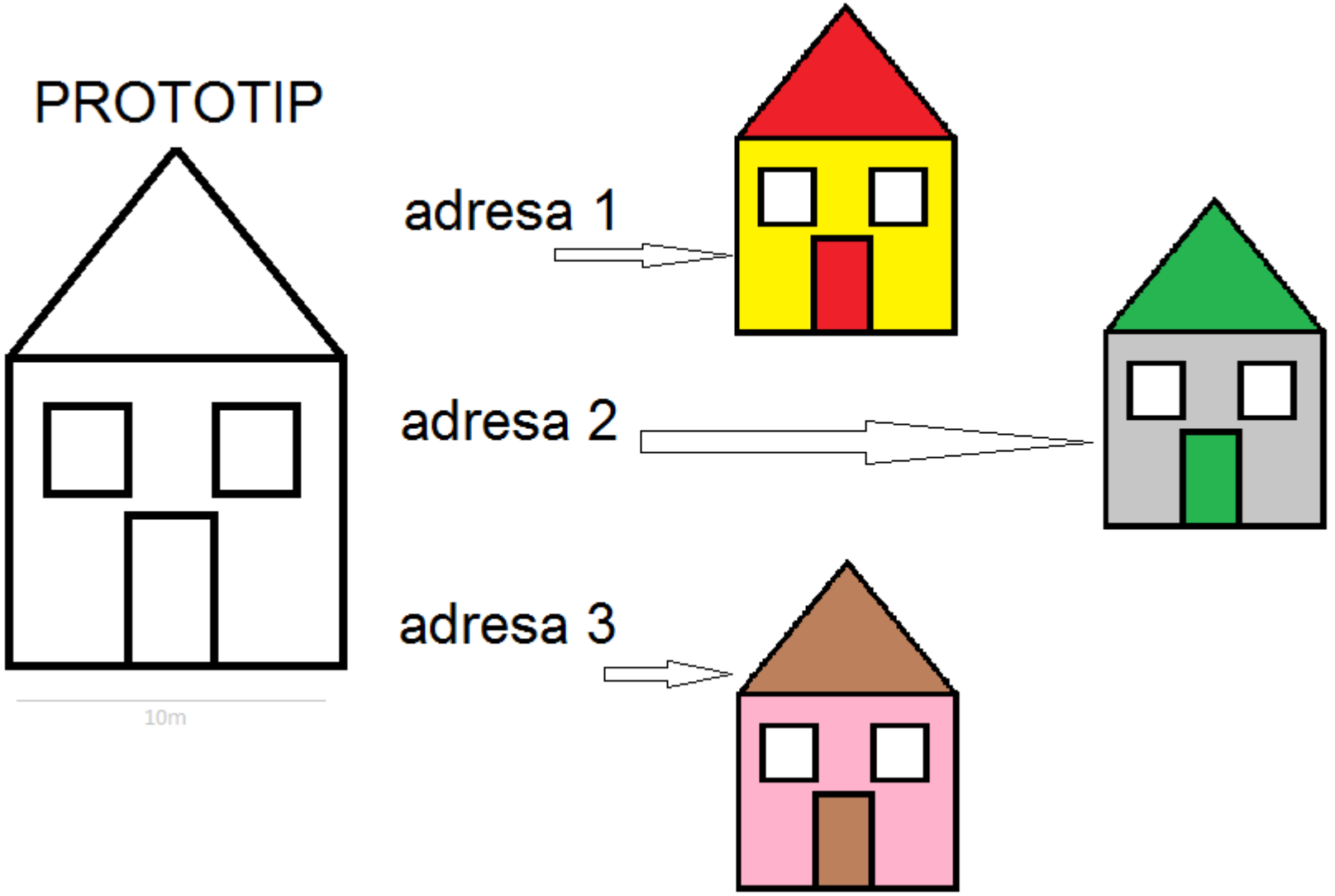
Programare avansată
Obiecte și clase

Concepte POO

- **Obiect** = Entitate software descrisă de o *stare* și de un *comportament*.
- **Clasă** = Prototip ce descrie obiecte:

  obiectele sunt instanțe ale claselor.
- **Referință** = Entitate ce oferă informații necesare localizării în mod unic a unui obiect
- **Program** = Mulțime dinamică de obiecte ce interacționează
- **Interfață** = Contract la care aderă o anumită clasă
- **Pachet** = Spațiu de nume necesar organizării

Clasă – Referință - Obiect



Crearea obiectelor

Declarare, Instanțiere, Inițializare

NumeClasa numeRef = new NumeClasa([argumente]);

```
Rectangle r1;
r1 = new Rectangle();

Rectangle r2 = new Rectangle(0, 0, 100, 200);

Rectangle patrat = new Rectangle(
    new Point(0,0), new Dimension(100, 100));
```

NullPointerException

```
Rectangle patrat;
```

(declarație echivalentă cu: `Rectangle patrat = null;`*)*

```
patrat.x = 10;
```

```
Rectangle[] patrate = new Rectangle[10];
```

```
patrate[0].x = 10;
```

Utilizarea obiectelor

- **referintaObiect.variabila**

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
System.out.println(patrat.width);
patrat.x = 10;
patrat.y = 20;
patrat.origin = new Point(10, 20);
```

- **referintaObiect.metoda([parametri])**

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
patrat.setLocation(10, 20);
patrat.setSize(200, 300);
```

Distrugerea obiectelor

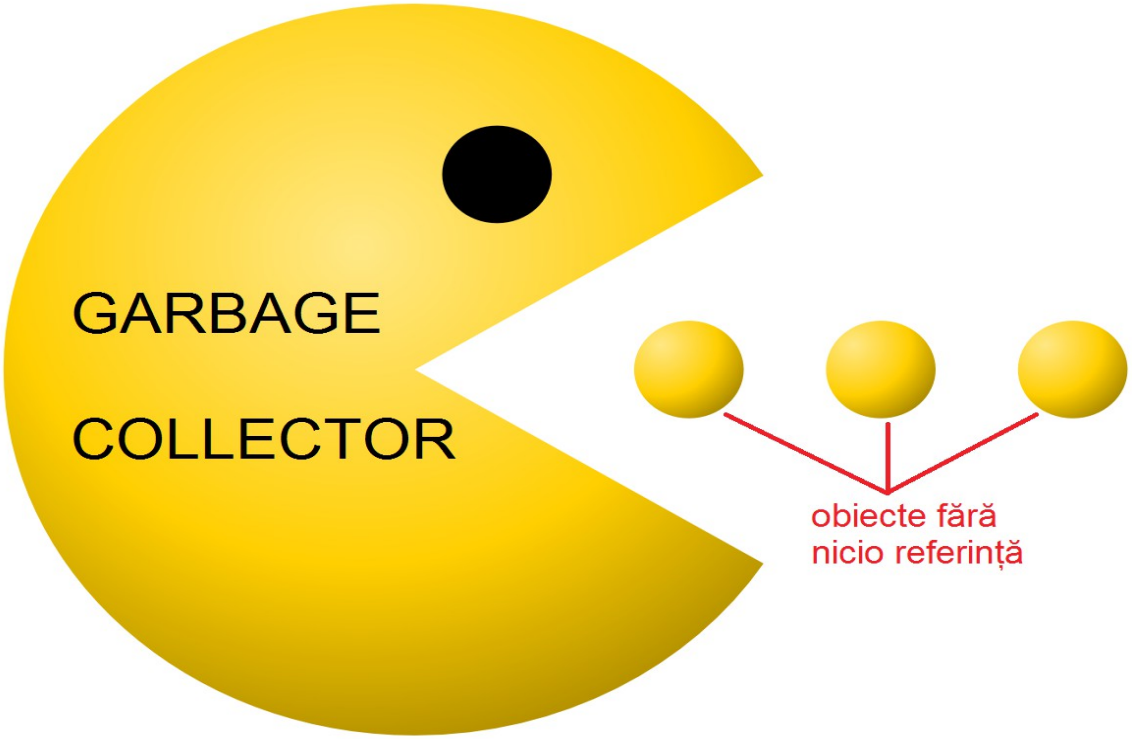
Obiectele care nu mai sunt referite vor fi distruse automat. Referințele sunt distruse:

- *natural*
- *explicit*, prin atribuirea valorii `null`.

```
class Test {
  String a;
  void init() {
    a = new String("aa");
    String b = new String("bb");
  }
  void stop() { a = null; }
}
```

Garbage Collector

Componenta JVM responsabilă cu eliberarea memoriei



`System.gc()`: ”Sugerează” JVM să elibereze memoria

Metoda **`finalize`**: apelată înainte de eliminarea unui obiect din memorie.

finalize ≠ destructor !

java *-verbose:gc*

Gestiunea memoriei

- **Heap, Stack**
- *java.lang.OutOfMemoryError*
  - *-Xms1024m*
  - *-Xmx2G*
- *java.lang.StackOverflowError*
  - *-Xss512k*
- *java.lang.Runtime*

```
Runtime runtime = Runtime.getRuntime();
long memory = runtime.totalMemory() - runtime.freeMemory();
```

Declararea unei clase

```
[public][abstract][final]class NumeClasa
    [extends NumeSuperclasa]
    [implements Interfata1 [, .. ]] {

    Corpul clasei
        Variabile
        Constructori
        Metode
        Clase imbricate (interne)
}
```

Exemplu de clasă

```
public class Persoana {

    private int id;
    protected String nume;

    public Persoana(String nume) {
        this.nume = nume;
    }

    public String getNume() {
        return nume;
    }

    void setNume(String nume) {
        this.nume = nume;
    }

}
```

Modificatori de acces

- **`private`**

  membru accesibil doar clasei
- **`protected`**

  membru accesibil doar clasei și subclaselor
- **`public`**

  membru accesibil tuturor
- *implicit*

  membru accesibil doar la nivel de pachet

Moștenirea

Moștenire simplă

O clasă are un singur părinte … cu excepția?

```
public class Student extends Persoana {
  // Persoana este superclasa clasei Student
  // Student este o subclasa a clasei Persoana
}
```

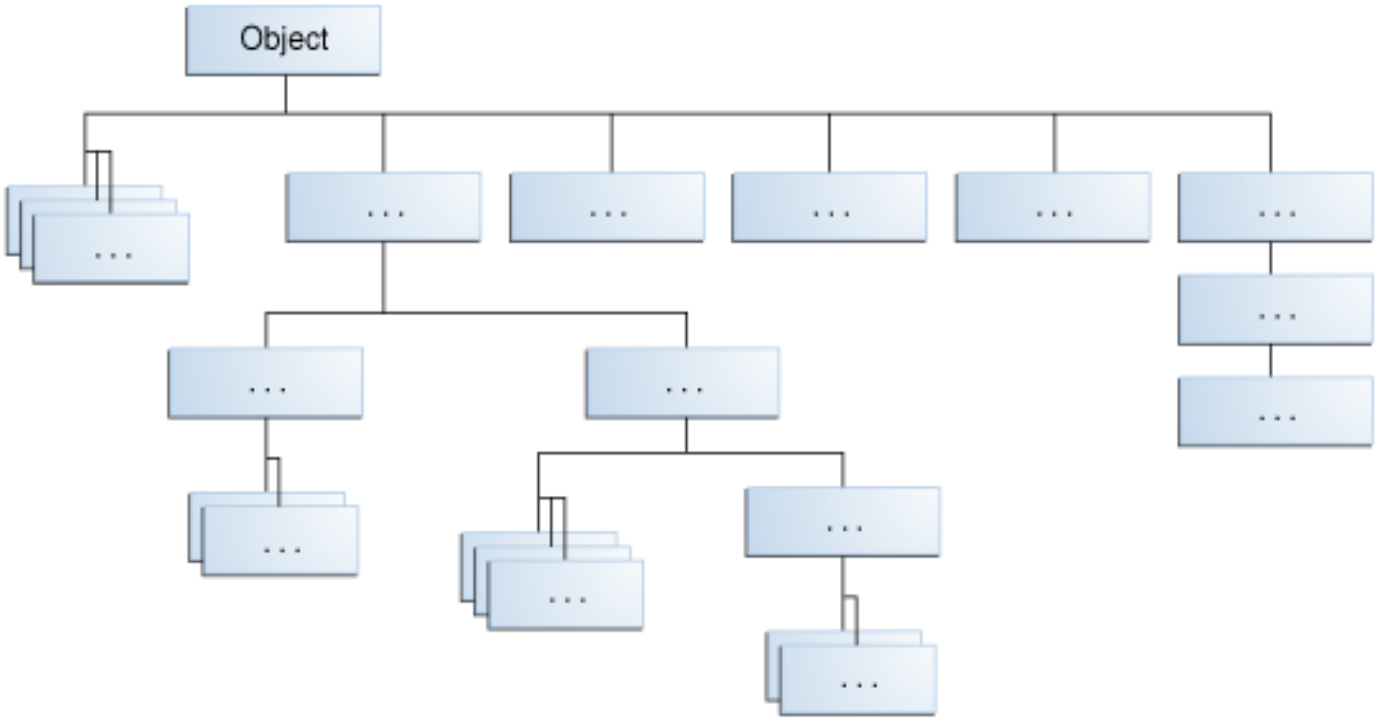
Moștenirea multiplă nu este posibilă!

```
public class Student extends Persoana, Robot {
  // Eroare
}
```

Clasa *Object*

Object este superclasa tuturor claselor.

```
class A {}
class A extends Object {}
```



Metodele clasei *Object*

Toate obiectele, inclusiv tablourile, moștenesc metodele acestei clase:

- **`toString`** : returnează reprezentarea ca șir de caractere a unui obiect
- **`equals`** : testează egalitatea conținutului a două obiecte
- **`hashCode`** : returnează valoarea *hash* corespunzătoare unui obiect
- **`getClass`** : returnează clasa din care a fost instanțiat obiectul
- **`clone`** : creează o copie a obiectului (implicit: *shallow copy*)
- **`finalize`** : apelată de GC înainte de distrugerea obiectului
- `...`

Exemplu de supradefinire a metodelor clasei Object

```
public class Complex {
  private double a, b;
  public Complex aduna(Complex comp) {
    return new Complex(a + comp.a, b + comp.b);
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null) return false;
    if (!(obj instanceof Complex)) return false;
    Complex comp = (Complex) obj;
    return ( comp.a==a && comp.b==b);
  }
  @Override
  public String toString() {
    String semn = (b > 0 ? "+" : "-");
    return a + semn + b + "i";
  }
}
...
Complex c1 = new Complex(1,2); Complex c2 = new Complex(2,3);
System.out.println(c1.aduna(c2));    // 3.0 + 5.0i
System.out.println(c1.equals(c2));   // false
```

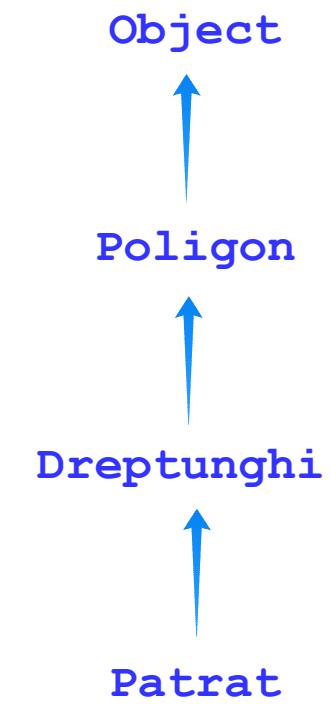
Tip referință – instanță clasă

Un obiect poate fi referit de o variabilă având tipul potrivit.

```
Patrat ref1     = new Patrat();
Dreptunghi ref2 = new Patrat();
Poligon ref3    = new Patrat();
Object ref4     = new Patrat();

Patrat badRef = new Dreptunghi();

Poligon metoda1( ) {
  if (…)
    return new Patrat();          // Corect
  else
    return new Dreptunghi();      // Corect
}
Dreptunghi metoda2( ) {
  if (…)
    return new Poligon();         // Eroare
  else
    return new Patrat(;           // Corect
}
```



Constructorii unei clase

```
public class NumeClasa {
  [modificatori] NumeClasa([argumente]) {
    // Constructor
  }
}
```

```
class A {
    protected int x;
    public A(int x) { this.x = x; }
    public A() { this(0);}
}

class B extends A{
    public B(int x) { super(x);}
}

class C {
    //Constructor generat implicit
}
```

Invocarea constructorilor

```
class A {
    public A() {
        System.out.println("A");
    }
}

class B extends A {
    public B() {
        System.out.println("B");
    }
}

class C extends B {
    public C() {
        System.out.println("C");
    }
}
```

```
C c = new C();
```

Metodele unei clase

```
public class NumeClasa {
   [modificatori] TipReturnat metoda([argumente]) {
     // Corpul metodei
   }
}
```

```
class A {
    public void hello() {
        System.out.println("Hello");
    }
    public void hello(String str) {
        System.out.println("Hello " + str);
    }
}
class B extends A {
    public void hello() {
        super.hello();
        System.out.println("Salut");
    }
    public void hello(String str) {
        System.out.println("Salut " + str);
    }
}
```

Supraîncărcare (Overloading)

Supradefinire (Overriding)

Trimiterea parametrilor

Argumentele sunt trimise doar prin valoare (pass-by-value)!

```
void metoda(StringBuilder s1, StringBuilder s2, int numar)
{

  // StringBuilder este tip referinta
  // int este tip primitiv
  s1.append("bc");
  s2 = new StringBuilder("yz");
  numar = 123;
}
...
StringBuilder s1 = new StringBuilder("a");
StringBuilder s2 = new StringBuilder("x");
int n = 0;
metoda(s1, s2, n);
System.out.println(s1 + ", " + s2 + ", " + n);
```

Metode cu număr variabil de argumente

[modificatori] TipReturnat metoda(TipArgumente ... args)

```
void metoda(Object ... args) {
  for(int i=0; i<args.length; i++)
  System.out.println(args[i]);
}
...
metoda("Hello");
metoda("Hello", "Java", 1.5);
```

```
System.out.printf("%s %d %n", "Total:", 1000);
```

Modificatorul **`final`**

- Variabile finale – nu mai pot fi schimbate

```
final int MAX = 100; MAX = 200;
```

- Metode finale – nu mai pot fi supradefinite

```
class Student {
  ...
  final float calcMedie(float note[])
  {
    ...
  }
}
```

```
class StudentInformatica
    extends Student {
  float calcMedie(float note[]) {
    return 10.00;
  }
}// Eroare la compilare!
```

- Clase finale – nu mai pot fi extinse

```
final class A{}, class B extends A {}
```

Modificatorul `static`

- Variabile statice (de clasă) – valori memorate la nivelul clasei și nu la nivelul fiecărei instanțe.

  Exemplu: declararea eficientă a constantelor

  ```
  static final double PI = 3.14;
  ```

- Metode statice (de clasă) – metode aplicabile la nivel de clasă și nu de instanță (pot opera doar pe variabile statice)

  Exemplu: metode "globale"

  ```
  double x = Math.sqrt(2);
  ```

Utilizarea membrilor statici

```
public class Exemplu {
  int x ;          // Variabila de instanta
  static long n;  // Variabila de clasa
  public void metodaDeInstanta() {
    n ++; // Corect
    x --; // Corect
  }
  public static void metodaStatica() {
    n ++; // Corect
    x --; // Eroare la compilare !
  }
}
Exemplu.metodaStatica();       // Corect
Exemplu obj = new Exemplu();
obj.metodaStatica();           // Corect

Exemplu.metodaDeInstanta();    // Eroare
Exemplu obj = new Exemplu();
obj.metodaDeInstanta();        // Corect
```

Blocuri statice de inițializare

"Constructori" la nivelul clasei și nu al obiectelor

```
static {
  // Bloc static de initializare;
  ...
}
```

```
public class Test {
  // Declaratii de variabile statice
  static int x = 0, y, z;

  // Bloc static de initializare
  static {
    System.out.println("Initializare clasa...");
    int t=1;
    y = 2;
    z = x + y + t;
  }
  public Test() { ... }
}
```

Clase imbricate

Clase declarate în interiorul altor clase

```
public class ClasaDeAcoperire {

  private class ClasaImbricata1 {
    // Clasa membru
    // Acces la membrii clasei de acoperire
  }
  void metoda() {
    class ClasaImbricata2 {
      // Clasa locala metodei
      // Acces la membrii clasei de acoperire si
      // la variabilele finale ale metodei
    }
  }
}
```

Identificare claselor imbricate

```
ClasaDeAcoperire.class
ClasaDeAcoperire$ClasaImbricata1.class
ClasaDeAcoperire$ClasaImbricata2.class
```

Clase și metode abstracte

```
[public] abstract class ClasaAbstracta {
  // Declaratii de metode abstracte
  abstract void metodaAbstracta();
  // Declaratii uzuale
  void metodaNormala() { … }
}
```

- Definesc un *șablon* pe care pot fi construite clase *concrete*, obținute prin extinderea clasei abstracte
- Pot fi parțial implementate sau complet abstracte
- Nu pot fi instanțiate

Exemple:

java.awt.Component: **Button, List, ...**

java.lang.Number: **Integer, Double, ...**

Conversii automate între tipuri primitive și referință

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

```
Integer refi = new Integer(1);
int i = refi.intValue();

Boolean refb = new Boolean(true);
boolean b = refb.booleanValue();
```

```
Integer refi = 1; //(auto)boxing
int i = refi;     //(auto)unboxing

Boolean refb = true;
boolean b = refb;
```

Tipuri de date enumerare

```
public enum CuloareSemafor {
    ROSU, GALBEN, VERDE;
}
```

```
class EnumTest {
    CuloareSemafor culoare;
    public EnumTest(CuloareSemafor culoare) {
        this.culoare = culoare;
    }
    public boolean traversam() {
        switch (culoare) {
            case VERDE: return true;
            default   : return false;
        }
    }
}
    new EnumTest(CuloareSemafor.GALBEN).traversam();
```

Enumerările sunt transformate de compilator în clase; acesta mai adaugă și alte metode: `CuloareSemafor.values()`

Crearea unor tipuri speciale de clase

Creational Design Patterns

- Singleton
- Object Factory
- Object Pool
- Prototype
- Builder
- *...*