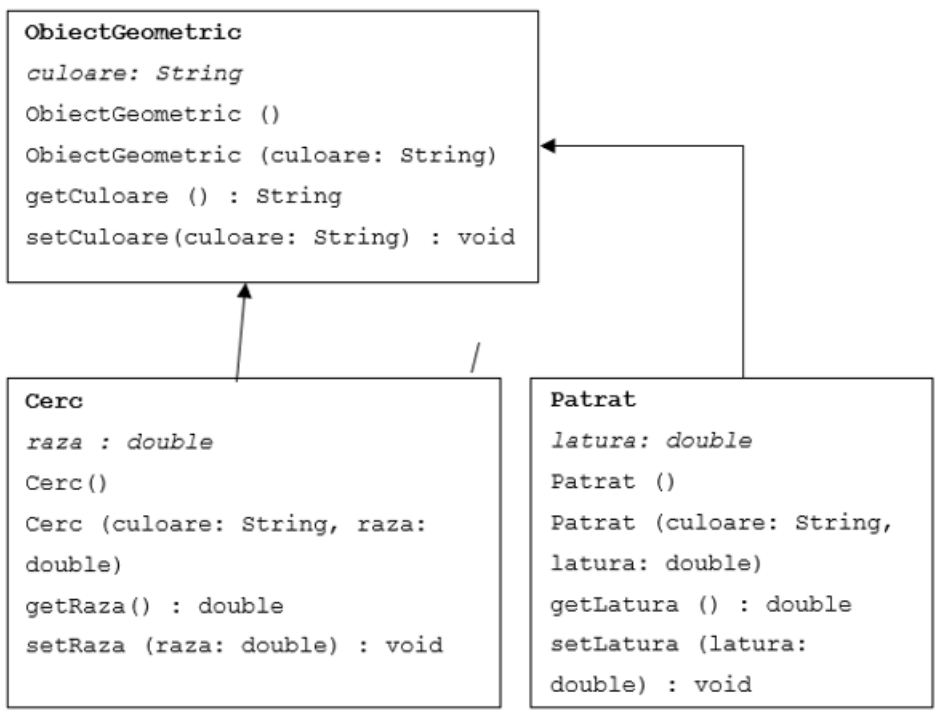
Programarea orientată pe obiecte

SERIA CB

MOȘTENIRE ȘI POLIMORFISM

Programarea orientată pe obiecte permite derivarea de noi clase din cele existente. Acest procedeu se numește **moștenire** și presupune definirea unei clase generale și extinderea ei într-o clasă specializată.

Clasele specializate **moștenesc proprietățile și metodele** din clasele generale.



`ObiectGeometric` este **clasa de bază** (numită și **superclasă** sau **clasă părinte**), iar `Cerc` și `Patrat` sunt **clasele derivate** (numite și **clase copil** sau **subclase**).

O clasă derivată **moștenește** variabilele membru și metodele accesibile din clasa de bază și **poate adăuga noi atribute și metode.**

Clasa `Cerc` moștenește atributul `culoare` din `ObiectGeometric`, dar și metodele ei, adăugând variabila `raza` de tip `double`, precum și metode care o manipulează: `getRaza()` și `setRaza(raza: double)`. Similar se întâmplă cu clasa `Patrat`.

- Cuvântul cheie `extends` precizează clasa din care se extinde.
- Subclasa nu este un subset al clasei de bază. De cele mai multe ori, *clasa derivată conține mai multe variabile membru și metode decât clasa părinte.*
- Variabilele membru de tipul `private` din

```java
public class ObiectGeometric {
    private String culoare;

    public ObiectGeometric(){}

    public ObiectGeometric(String culoare) {
        this.culoare = culoare;
    }
    String getCuloare() {
        return culoare;
    }
    void setCuloare(String culoare) {
        this.culoare = culoare;
    }
    public String toString() {
        return "Culoarea este: " + culoare;
    }
}

public class Cerc extends ObiectGeometric
{
    private double raza;
    public Cerc() {}
    public Cerc(String culoare, double raza) {
        this.setCuloare(culoare);
        this.raza = raza;
    }
    double getRaza() {
        return raza;
    }
    void setRaza(double raza) {
        this.raza = raza;
    }
}

public class Patrat extends ObiectGeometric
{
    private double latura;
    public Patrat() {}

    public Patrat(String culoare, double latura) {
        this.setCuloare(culoare);
        this.latura = latura;
    }
    double getLatura() {
        return latura;
    }
    void setLatura(double latura) {
        this.latura = latura;
    }
}
```

Instrucțiunea `super()` invocă constructorul fără parametri al clasei de bază, iar `super(parameters)`, pe cel cu parametri:

```java
public Cerc(String culoare, double raza) {
    super(culoare);
    this.raza = raza;
}
```

clasa de bază nu sunt accesibile direct din clasa derivată. Ele pot fi accesate sau modificate prin intermediul metodelor **get** și **set** (dacă acestea există).

- Între superclasă și subclasă trebuie să existe o relație de tipul ***is-a***.
- În Java, o clasă copil **poate extinde o singură clasă de bază/părinte**. Acest lucru este cunoscut sub denumirea de *single inheritance*.

Constructorul cu parametri al clasei `Cerc`:

`public Cerc (String culoare, double raza)`

apelează metoda `setCuloare(culoare)` din clasa de bază.

Modificarea valorii culorii se face prin intermediul metodei `setCuloare` deoarece nu se poate accesa câmpul `culoare` din clasa de bază, acesta fiind privat!

În acest exemplu, `super(culoare)` apelează constructorul clasei `ObiectGeometric`.
Apelul lui `super` trebuie mereu să fie prima instrucțiune a metodei!

Crearea unui obiect al unei clase implică apelul constructorilor tuturor superclaselor de-a lungul lanțului de moștenire.

Când se creează un obiect al unei subclase, constructorul subclasei invocă constructorul superclasei.

Dacă această superclasă este derivată dintr-o altă clasă, atunci se apelează și constructorul ei, și așa mai departe. Acest mecanism poartă numele de ***constructor chaining*.**

```
public class Student extends Tanar{
    public Student() {
        System.out.println("Acesta este un student ");
    }
    public static void main(String[] args) {
        new Student();
    }
}
class Tanar extends Persoana{
    public Tanar() {
        System.out.println("Acesta este un tanar ");
        System.out.println("Lui ii place muzica ");
    }
}
class Persoana{
    public Persoana() {
        System.out.println("Acesta este o persoana ");
    }
}
```

Metoda `main` din clasa `Student` va afișa:

```
Aceasta este o persoana
Acesta este un tanar
Lui ii place muzica
Acesta este un student
```

Este de preferat să se scrie constructori fără parametri pentru fiecare clasă, în așa fel încât să se evite erorile de compilare la extinderea claselor.

Cuvântul cheie `super` poate însoți și apelul unei metode ale clasei de bază. De exemplu, `super.getCuloare()`, apelat într-o metodă a clasei `Cerc` sau `Patrat`, returnează culoarea obiectului.

```
public void afisareCerc() {
    System.out.println ("Culoarea: " + super.getCuloare());
}
```

În cazul de mai sus, nu este nevoie de prezența cuvântului `super`, întrucât `getCuloare()` este o metodă moștenită din clasa părinte, `ObiectGeometric`.

Modificatorul de acces `protected` este utilizat în clasa de bază pentru a însoți variabilele membru și metodele ce vor putea fi accesate și din clasele derivate.

Exemplu suprascrierea metodelor:

SUPRASCRIEREA METODELOR

Subclasele preiau metodele din clasa de bază. Uneori este nevoie să se modifice implementarea metodelor definite în clasa de bază. Acest lucru se numește suprascrierea metodelor **(overriding).**

Metoda `toString()` din clasa `ObiectGeometric` furnizează o reprezentare sub formă de `String` pentru obiectele de tipul `ObiectGeometric`. Această metodă, disponibilă în clasa `Object` (părinte al clasei `ObiectGeometric`), poate fi suprascrisă pentru a returna o reprezentare `String` a obiectelor de tipul `Cerc`. (A SE VEDEA PRIMUL EXEMPLU DE PE PAGINA URMĂTOARE)

O metodă statică poate fi moștenită, dar nu poate fi suprascrisă!

Dacă o metodă definită în clasa de bază este redefinită în clasa copil, atunci definiția ei din clasa de bază este ascunsă.

O metodă statică ascunsă poate fi accesată prin apelul `NumeClasa.numeMetoda()`

O metodă non-statică poate fi suprascrisă doar dacă este accesibilă.

De exemplu, o metodă privată din clasa de bază nu este accesibilă în clasa derivată.

```java
public class Cerc extends ObiectGeometric {
    public String toString() {
        return super.toString() + " raza: " + raza;
    }
}
```

SUPRAÎNCĂRCAREA METODELOR

Supraîncărcarea metodelor (overloading) înseamnă definirea unor noi metode cu același nume, dar cu semnătură diferită (fie prin numărul parametrilor, fie prin tipul lor dacă numărul parametrilor este același).

Suprascrierea se referă la redefinirea unei metode moștenite dintr-o clasă de bază. **Redefinirea în clasa derivată** se face folosind aceeași semnătură a funcției și același tip returnat.

```java
class B {
    public void f(double x) {
        System.out.println(x);
    }
}
class A extends B {
    // exemplu de suprascriere
    public void f(double x ) {
        System.out.println(x+1);
    }
}

class B {
    public void f(double x) {
        System.out.println(x);
    }
}
class A extends B {
    // exemplu de supraîncărcare
    public void f(int x ) {
        System.out.println(x+1);
    }
}
```

Pentru apelul:

```java
A a = new A();
a.f(5);
a.f(5.0);
```

`a.f(5)` se va folosi de metoda `f(int x)`, iar `a.f(5.0)` se va folosi de metoda `f(double x);`

O metodă redefinită în subclasă și căreia i se schimbă modificatorul de acces nu influențează definiția din clasa de bază

Orice clasă Java este descendentă din clasa `java.lang.Object`, chiar dacă acest lucru nu este precizat explicit. Metoda `toString()` din clasa `Object` returnează o reprezentare sub formă de `String` a obiectului, sub forma – nume obiect (respectiv clasa din care face parte) + simbolul @ + adresa sa de memorie în hexazecimal.

De exemplu, următoarele linii de cod:

```java
ObiectGeometric obj = new ObiectGeometric();
System.out.println(obj.toString());
```

echivalent cu

```java
System.out.println(obj);
```

va afișa ceva similar cu: `ObiectGeometric@1234f`

O subclasă poate suprascrie o metodă care este `protected` în clasa de bază și poate schimba vizibilitatea ei în `public`.

În subclasă nu se poate micșora accesibilitatea unei metode. De exemplu, dacă ea este definită publică în clasa de bază, atunci trebuie să fie publică și în clasa derivată.

POLIMORFISMUL

Polimorfismul se referă la faptul că un obiect din clasa derivată poate fi folosit ca parametru al unei metode ce primește la definire un obiect al clasei de bază (a se vedea următorul exemplu).

```
public static void main (String[] args) {
    Cerc c = new Cerc("rosu", 5.0);
    Patrat p = new Patrat("verde", 10.0);
    afisareProprietati(c);
    afisareProprietati(p);
}

public void afisareProprietati(ObiectGeometric o) {
    System.out.println(o.getCuloare());
}
```

Fie secvența de mai jos:

```
Object o = new ObiectGeometric();
System.out.println(o.toString());
```

`Object` este ***tipul declarat,*** iar `ObiectGeometric` este ***tipul actual*** pentru `o`. `ObiectGeometric` este derivat din `Object` și are suprascrisă implementarea metodei `toString()`. În acest caz, metoda `toString()` care se va apela va fi cea a clasei `ObiectGeometric`, adică a primei clase care are metoda suprascrisă, pe lanțul de moștenire. Dacă `ObiectGeometric` nu ar fi avut metoda `toString()` suprascrisă, atunci s-ar fi apelat metoda `toString()` din clasa `Object`.

Tipul declarat al referinței decide ce metode se aleg la compilare. Tot atunci, compilatorul găsește o metodă potrivită, în funcție de numărul de parametri, tipul și ordinea lor. O metodă poate fi reimplementată în mai multe clase derivate. JVM alege implementarea metodei la runtime, decisă de ***tipul actual al obiectului.***

CASTING ȘI INSTANCEOF

```
Object o = new ObiectGeometric();
```

Exemplul anterior ilustrează **casting implicit** deoarece o instanță a lui `ObiectGeometric` este automat și o instanță a lui `Object`.
Asignarea `ObiectGeometric x = o;` conduce la eroare de compilare, deoarece `ObiectGeometric` este o instanță a lui `Object`, dar invers nu este valabil. În acest caz trebuie să se folosească casting explicit.

```
ObiectGeometric x = (ObiectGeometric)o;
```

Se poate converti o instanță a subclasei la una a clasei de bază – **upcasting**. Acest lucru se face **automat**, deoarece o instanță a clasei copil este și o instanță a clasei părinte. Când se convertește o instanță a superclasei la subclasă (**downcasting**), se precizează numele subclasei între paranteze prin casting explicit (a se vedea următorul exemplu).

```java
Object obj = new Cerc();
if (obj instanceof Cerc) {
    System.out.println("Aria: " + ((Cerc)obj).getAria());
}
```

Exemplu de upcasting:

```java
class Masina {
    public void merge() {}
        static void functioneaza(Masina m) {
            m.merge();
    }
}
public class Dacia extends Masina {
    public static void main(String[] args) {
        Dacia d = new Dacia();
        Masina.functioneaza(d); // Upcasting automat
    }
}
```

Deși obiectul `d` este o instanță a clasei `Dacia`, acesta este pasat ca parametru în locul unui obiect de tipul `Masina`, care este o superclasă a clasei `Dacia`. Upcasting-ul se face la trimiterea parametrului.

Downcasting-ul este operația inversă upcasting-ului. El reprezintă o conversie explicită de tip în care se merge în jos pe ierarhia claselor (se convertește o clasă de bază într-una derivată). Acest cast trebuie făcut **explicit** de către programator:

```java
class Angajat {
    public void lucreaza() {
        System.out.println("Angajatul lucreaza");
    }
}

class Contabil extends Angajat {
    public void gestioneaza() {
        System.out.println("Contabilul gestioneaza");
    }

public void lucreaza() {
    System.out.println("Contabilul lucreaza");
    }
}

class Director extends Angajat {
    public void conduce() {
        System.out.println("Directorul conduce");
    }
}

class Test {
    public static void main(String[] args) {
        Angajat a [] = new Angajat[2];
        a[0] = new Contabil();
        a[1] = new Director();
        for (int i = 0; i < a.length; i++) {
            a[i].lucreaza(); // 1
            if (a[i] instanceof Contabil)
                ((Contabil)a[i]).gestioneaza(); // 2
            if (a[i] instanceof Director)
                ((Director)a[i]).conduce(); // 3
        }
    }
}
```

Clasa `Angajat` are metoda `lucreaza()`, ce afișează mesajul *"Angajatul lucreaza"*.

În clasa derivată `Contabil`, această metodă este suprascrisă și se va afișa mesajul *"Contabilul lucreaza"*. Se adaugă în plus aici metoda `gestioneaza()`, ce afișează mesajul *"Contabilul gestioneaza"*.

În clasa `Director`, nu se suprascrie metoda `lucreaza()`, însă se adaugă metoda `conduce()`, ce afișează mesajul *"Directorul conduce"*.

În metoda `main`, se declară un tablou de obiecte de tipul `Angajat`, cu 2 elemente. Pentru primul obiect, tipul declarat este `Angajat`, iar cel actual este `Contabil`. Pentru al doilea obiect, tipul declarat este `Angajat`, iar cel actual, `Director`. Metoda `lucreaza()` este comună ambelor obiecte. Pentru a apela metoda `gestioneaza()`, se face o conversie explicită de la tipul de bază `Angajat` la tipul `Contabil`, iar pentru a apela metoda `conduce()`, se face conversia de la tipul `Angajat` la tipul `Director`. Aceasta este operația de **downcasting.**

APLICAȚII:

Mijloc de transport

Implementați clasa `MijlocTransport`, ce va conține:

- proprietățile `private String culoare` și `private boolean functional;`
- un constructor fără parametri ce setează `culoare = "alb", functional = false;`
- un constructor cu parametri ce setează câmpurile `culoare` și `functional;`
- getter-e și setter-e pentru proprietăți
- metodele `int incasare()` și `int profit()` ce întorc valoarea `0`
- metoda `String toString()` ce va întoarce o reprezentare de tip `String` a obiectului

Autobuz și Microbuz

- Implementați clasele `Autobuz` și `Microbuz` care moștenesc clasa `MijlocTransport`.
- Fiecare clasă va avea în plus câmpurile `private numarPasageri`, `pretBilet` și `intretinrePerBilet` de tipul `int`.
- Suprascrieți metodele `incasare()` și `profit()` care calculează încasările și profitul din vânzarea biletelor astfel: profitul reprezintă 25% din (totalul încasărilor minus cheltuielile de întreținere). Încasările se calculează ca `numarPasageri * pretBilet`. Cheltuielile de întreținere se calculează ca `numarPasageri * intretinerePerBilet`.
- Suprascrieți metoda `String toString()` astfel încât să afișeze:

`Autobuz (culoare, functional) cu numarPasageri si pretBilet are profit: profit` – pentru un obiect din clasa `Autobuz`

`Microbuz (culoare, functional) cu numarPasageri și pretBilet are profit: profit` – pentru un obiect din clasa `Microbuz`

- Implementați următorii constructori utilizând constructorii clasei de bază (se apelează `super()`):

`public Autobuz(String culoare, boolean functional, int numarPasageri, int pretBilet)`

`public Microbuz(String culoare, boolean functional, int numarPasageri, int pretBilet)`

Autogara

- Implementați clasa `Autogara` ce conține ca variabilă membru privată un tablou unidimensional de obiecte de tipul `MijlocTransport;`
- Implementați un constructor cu parametri care inițializează tabloul cu dimensiunea dată ca parametru;
- Implementați metoda `void adaugaMijlocTransport(MijlocTransport m)` care adaugă un mijloc de transport în tabloul unidimensional;
- Implementați metoda `int profitTotal()` care calculează profitul total al mijloacelor de transport din tablou;
- Implementați metoda `String toString()` care afișează informațiile despre fiecare obiect din tablou pe câte o linie pe ecran, prin apeluri ale funcțiilor `toString()` specifice fiecărui obiect, iar la final profitul total;
- În cadrul funcției `main` creați un obiect de tipul `Autogara` și adăugați un număr de 2 obiecte de tip `Autobuz` și un număr de 2 obiecte de tip `Microbuz`. Afișați detaliile autogării.

```
Autobuz a1 = new Autobuz( culoare: "alb", functional: true, numarPasageri: 20, pretBilet: 3);
Autobuz a2 = new Autobuz( culoare: "verde", functional: true, numarPasageri: 30, pretBilet: 4);
Microbuz m1 = new Microbuz( culoare: "albastru", functional: true, numarPasageri: 15, pretBilet: 5);
Microbuz m2 = new Microbuz( culoare: "gri", functional: true, numarPasageri: 10, pretBilet: 6);
```

Referințe:

1. Oana Balan, Mihai Dascalu. **Programarea orientata pe obiecte in Java**. Editura Politehnica Press, 2020
2. Bert Bates, Kathy Sierra. **Head First Java: Your Brain on Java - A Learner's Guide 1st Edition**. O'Reilly Media. 2003
3. Herbert Schildt. **Java: A Beginner's Guide**. McGraw Hill. 2018
4. Barry A. Burd. **Java For Dummies**. For Dummies. 2015
5. Joshua Bloch. **Effective Java**. ISBN-13978-0134685991. 2017
6. Harry (Author), Chris James (Editor). **Thinking in Java: Advanced Features (Core Series) Updated To Java** 8
7. Learn Java online. Disponibil la adresa: https://www.learnjavaonline.org/
8. Java Tutorial. Disponibil la adresa: https://www.w3schools.com/java/
9. The Java Tutorials. Disponibil la adresa: https://docs.oracle.com/javase/tutorial/