

## **Tema: Object-oriented programming**

## Lista e figurave

<a href="#">Fig. 1</a> .....	7
<a href="#">Fig. 2</a> .....	7
<a href="#">Fig. 3</a> .....	8
<a href="#">Fig. 4</a> .....	9
<a href="#">Fig. 5</a> .....	9
<a href="#">Fig. 6</a> .....	10
<a href="#">Fig. 7</a> .....	10
<a href="#">Fig. 8</a> .....	11
<a href="#">Fig. 9</a> .....	11
<a href="#">Fig. 10</a> .....	12
<a href="#">Fig. 11</a> .....	12
<a href="#">Fig. 12</a> .....	13
<a href="#">Fig. 13</a> .....	14
<a href="#">Fig. 14</a> .....	14
<a href="#">Fig. 15</a> .....	15
<a href="#">Fig. 16</a> .....	15

Sot do të flasim për një koncept tepër të rëndësishëm. Si e kemi përmendur dhe në fillim Java është një gjuhë object-oriented (e orientuar në objekte).

**Object-oriented** do të thotë organizimi i programit si një kombinim tipesh të ndryshme objektesh të cilat kanë të dhëna dhe sjellje. Është një metodologji e cila ndihmon në thjeshtësimin e zhvillimit dhe mirëmbajtjes së një programi. Sipas OOP një program Java krijohet duke përdorur **Objekte** dhe **Klasa**.

Object-oriented programming (OOP) është një metodologji e cila ndihmon në thjeshtësimin e

*Konceptet bazë të OOP janë:*

- Objekti (Object)
- Klasa (Class)
- Trashëgimia (Inheritance)
- Polimorfizmi (Polymorphism)
- Abstraksioni (Abstraction)
- Enkapsulimi (Encapsulation)

## 1. Objekti (Object)

Një objekt Java përfaqëson një entitet fizik ose logjik të botës reale si përshembull: mollë, iPhone, një llogari bankare etj.

Ai kanë tre karakteristika:

- gjendja (state)
- sjellja (behaviour)
- identiteti (identity)



Po marr një shembull me objektin Njeri për ta bërë më të qartë. Njeriu ka gjendje (state) si emri, ngjyra e flokut, ngjyra e syze etj. dhe sjellje (behavior) si fle, ushqehet, ecën etj. Përsa i përket identitetit; është një ID unike, e cila përdoret prej **JVM** për të identifikuar veçantinë e çdo objekti. Kjo ID nuk është e dukshme për përdoruesin e jashtëm.

Një objekt është një instancë e një klase. Por, çfarë është Klasa?

## 2. Klasa (Class)

Klasa është një model(template) nga e cila krijohen objektet. Pra, një klasë përfaqëson një grup objektesh me cilësi të përbashkëta. Ajo është një entitet logjik. Një **Class** në Java përmban:

- Fusha (Fields/Variables)
- Metoda (Methods)
- Konstruktor (Constructor)
- Blloqe (Blocks)
- Klasa të ndërfutura dhe ndërfaqe (Nested classes and interface)

Sintaksa për deklarimin e një klase në Java është si më poshtë:

```
class <emri_i_klasës> {  
    fushë;  
    metodë;  
}
```

**Fusha** (Field) është një variabël brenda një klase.

**Variablat** janë të tre tipeve të ndryshme:

- **Variabla lokale** (local variables) janë variabla të deklaruara brenda një metode. Ato duhen inicializuar para se të përdoren pasi nuk kanë një vlerë default, për më tepër kompiluesi nuk ju lejon të lexoni një vlerë të painicializuar.
- **Variabla instance** (instance variables) janë variabla të cilat nuk janë lokale. Këto lloj variablash përdoren në shumë objekte. Variablat e instancës quhen dhe fusha (fields).
- **Variabla statike/klase** (static/class variables) janë variabla të cilat definojnë duke përdorur fjalën kyçe (keyword) 'static'. Këto variabla si ato të instancës deklarohen brenda klasës, por jashtë metodave dhe blloqeve të kodit.

**Variablat e instancës dhe ato të klasës nuk kanë nevojë për inicializim, pasi marrin një vlerë default sapo deklarohen.**

**Metoda** (Method) është një funksion, i cili përdoret për të ekspozuar sjelljen (behaviour) e një objekti. Avantazhet e metodave janë:

- Ripërdorimi i kodit (code reusability)

- Optimizimi i kodit (code optimization)
- Kursimi i kohës në rast se kemi detyra të gjata
- Mundësojnë polimorfizmin

**Konstruktori** është një lloj metode speciale, i cili përdoret për të inicializuar objektet. Kur duam të krijojmë një objekt nga një klasë e caktuar, atëherë thërrasim konstruktorin. Gjithashtu, me anë të tij mund të vendosim vlerat fillestare për atributet e objektit.

**Mbani mend!**

**Konstruktori duhet të ketë të njëjtin emër si i klasës dhe nuk kthen vlerë (no return type).**

Nëse në një klasë nuk krijojmë vet një konstruktor, ajo ka një konstruktor bosh si default, por meqënëse është bosh, nuk vendos dot vlera për atributet e objektit. Gjithashtu, diçka tjetër që duhet të keni parasysh: **nëse krijoni vet një konstruktor me parametra dhe doni të përdorni konstruktorin bosh, atëherë duhet të krijoni dhe konstruktorin bosh pasi nuk ekziston më by default.**

**Bloqe** (Blocks) janë një set kodi i futur brenda kllapave gjarpërushe { } në çdo klasë, metodë ose konstruktor. Ata fillojnë me një kllapë gjarpërushe hapëse ( { ) dhe përfundojnë me një kllapë gjarpërushe mbyllëse ( } ). Midis kllapave, mund të shkruajmë kod i cili mund të jetë një pohim i vetëm ose më tepër.

**Klasë e ndërfutur** (Nested class) është një klasë brenda një klase tjetër. Klasat e ndërfutura ndahen në dy kategori: statike dhe jo-statike. Klasat e ndërfutura jo-statike ndryshe quhen inner classes. Ato statike ndryshe quhen klasa të ndërfutura statike (static nested classes).

Sintaksa për deklarimin e Nested Classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

## Përse të përdorim Nested Classes?

- Nëse një klasë është e dobishme vetëm për një klasë tjetër është *më e logjikshme* të vendoset brenda saj.
- *Rrit enkapsulimin* (encapsulation). Duke fshehur klasën B brenda klasës A, atëherë pjesëtarët e A-së mund të deklarohen 'private' dhe B mund t'i aksesojë. Gjithashtu dhe vet klasa B është e fshehur.
- Kodi bëhet *më i thjeshtë për t'u lexuar dhe mirëmbajtur*.

**Ndërfaqja** (Interface) është një "klasë plotësisht abstrakte" që përdoret për të grupuar metoda boshe që çdo klasë që implementon ndërfaqen duhet t'i definojë. Një ndërfaqe mund gjithashtu të përfshijë një listë *variablash konstante* dhe *metodash default*. Ajo specifikon se çfarë një klasë duhet të bëjë dhe jo si; pra funksionon si një model i klasës. E shpjegojmë gjithashtu me një shembull. *Kemi ndërfaqen Llogari (për llogari bankare). Kemi një klasë LLogariKursimesh dhe një klasë LLogariRrjedhëse. Këto dy klasa dhe pse janë të ndryshme kanë disa funksionalitete të njëjta.* Në këtë mënyrë përdorimi i **Ndërfaqes** paracakton disa sjellje specifike standarde për të gjitha tipet e llogarive.

**Sintaksa e krijimit të ndërfaqes është si më poshtë:**

```
public interface A {  
  
    public static final int MAX_VALUE = 15; // variabël statike  
  
    public int getMaximum(); // metodë abstrakte  
  
}
```

## Përse përdoren ndërfaqet?

- Për të arritur abstraksionin
- Duke qenë se Java nuk lejon trashëgiminë e shumëfishtë të klasave, atëherë mund të përdoren ndërfaqet për të arritur trashëgiminë e shumëfishtë
- Për të definuar sjellje specifike

## 3. Trashëgimia (Inheritance)

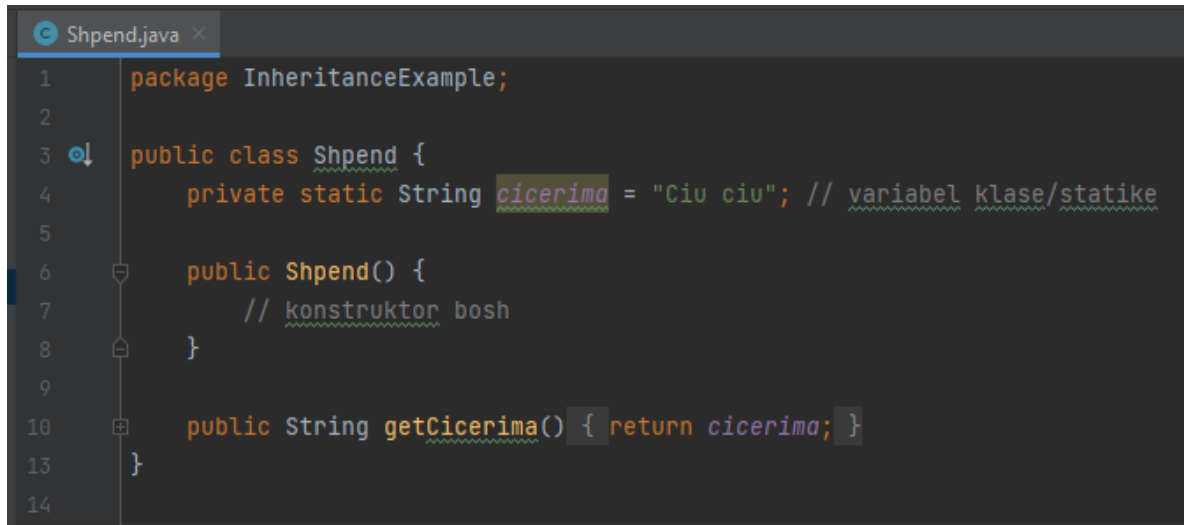
Java lejon që një klasë të trashëgojë sjelljen e një klase tjetër dhe gjithashtu të mund ta ndryshojë atë sjellje nëse duhet. Pra, **trashëgimia** na lejon të ndërtojmë klasa duke u bazuar në klasa të tjera. Kjo na mundëson që të shmangim duplikimin dhe përsëritjen e kodit.

Një klasë mund të trashëgojë një klasë tjetër duke përdorur fjalën kyçe '**extends**'.

```
class Harabel extends Shpend {...}
```

Klasa e cila trashëgon vetitë e klasës tjetër (në këtë rast klasa *Harabel*) quhet ndryshe '**child class/derived class/subclass**', kurse klasa e cila trashëgohet (klasa *Shpend*) quhet '**parent class/base class/superclass**'.

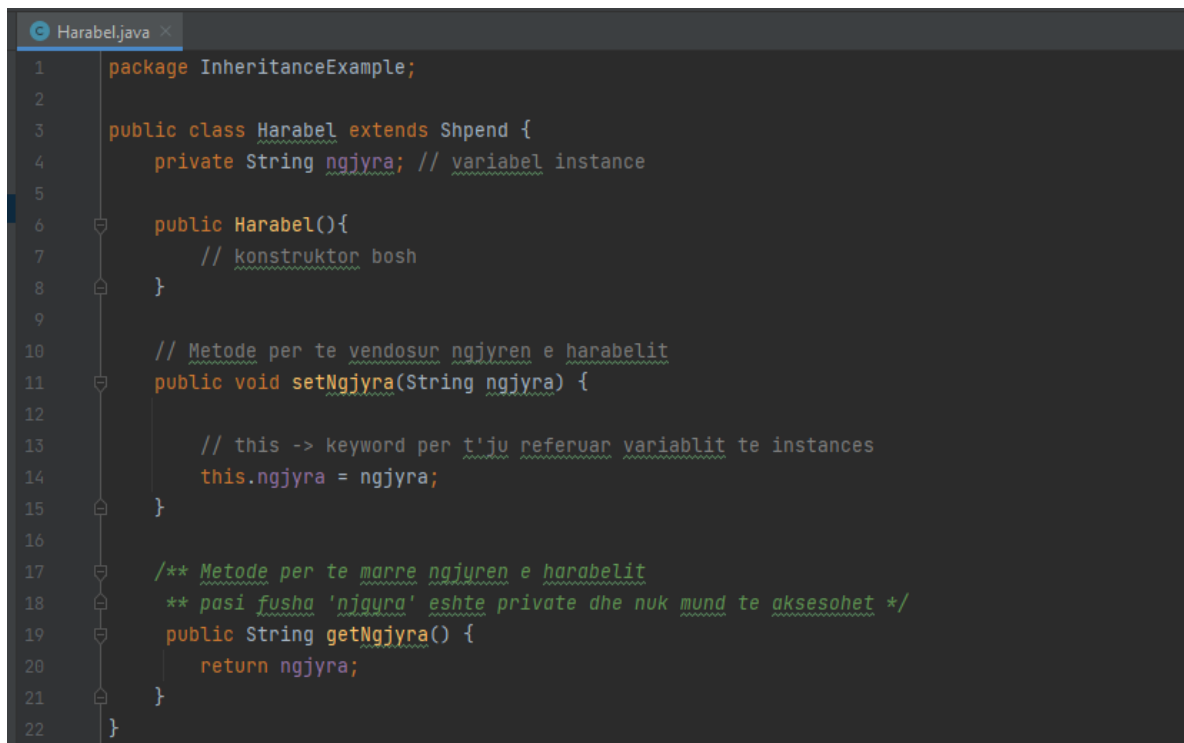
### Klasa Shpend (Shpend.java)

A screenshot of a code editor showing the Shpend.java file. The code is in Java and defines a class Shpend within the InheritanceExample package. It includes a static String variable cicerima, a constructor, and a getter method. The code is as follows:

```
1 package InheritanceExample;
2
3 public class Shpend {
4     private static String cicerima = "Ciu ciu"; // variabel klase/statike
5
6     public Shpend() {
7         // konstruktor bosh
8     }
9
10    public String getCicerima() { return cicerima; }
13 }
14
```

Fig. 1

### Klasa Harabel (Harabel.java)

A screenshot of a code editor showing the Harabel.java file. The code is in Java and defines a class Harabel that extends the Shpend class. It includes a private String variable ngjyra, a constructor, a setter method, and a getter method. The code is as follows:

```
1 package InheritanceExample;
2
3 public class Harabel extends Shpend {
4     private String ngjyra; // variabel instance
5
6     public Harabel(){
7         // konstruktor bosh
8     }
9
10    // Metode per te vendosur ngjyren e harabelit
11    public void setNgjyra(String ngjyra) {
12
13        // this -> keyword per t'ju referuar variablit te instances
14        this.ngjyra = ngjyra;
15    }
16
17    /** Metode per te marre ngjyren e harabelit
18     ** pasi fusha 'ngjyra' eshte private dhe nuk mund te aksesohet */
19    public String getNgjyra() {
20        return ngjyra;
21    }
22 }
```

Fig. 2

## Klasa TestClass (TestClass.java)

```
1 package InheritanceExample;
2
3 public class TestClass {
4     public static void main(String[] args) {
5         Harabel h = new Harabel(); // Krijimi i objektit Harabel
6         h.setNgjyra("kafe"); // Vendosja e ngjyres se Harabelit
7         System.out.println("Zhurma: " + h.getCicerima());
8         System.out.println("Ngjyra: " + h.getNgjyra());
9     }
10 }
```

Fig. 3

Në **klasën Shpend** kemi **variablin statik 'cicerima'**, i cili përcakton cicërimën e shpendëve (supozojmë se të gjithë bëjnë "Ciu ciu").

Në **klasën Harabel** me anë të keyword **'extends'** trashëgojmë **klasën Shpend**. Klasa Harabel ka **një variabël instance 'ngjyra'**, e cila përcakton ngjyrën e Harabelit. Metoda **setNgjyra()** përdoret për të vendosur ngjyrën e Harabelit, kurse metoda **getNgjyra()** përdoret për të marr ngjyrën e Harabelit pasi variabli i instancës është deklaruar si **'private'** (do e shpjegojmë në vazhdim).

Në rastin e metodës **setNgjyra()** përdoret keyword **'this'** për të referuar variablin **'ngjyra'** pasi parametri i vendosur në metodë ka të njëjtin emër si variabla e instancës.

Në mënyrë që të ekzekutojmë kodin duhet të deklarojmë metodën **main()**.

**Përse?** Në çdo program Java, metoda **main()** është pika fillestare nga ku **kompiluesi**(compiler) fillon ekzekutimin e programit. Pra, kompiluesi duhet të thërrasë metodën **main()** për të ekzekutuar programin.

Kjo metodë është deklaruar në **klasën 'TestClass'**. Në rreshtin numër 3 është krijuar objekti Harabel. Në këtë moment, Harabeli nuk ka ngjyrë akoma, por cicërima e tij është "Ciu ciu", pasi është trashëguar nga **klasa prind Shpend**. Në rreshtin 4 duke thirrur metodën **setNgjyra("kafe")** i vendosim Harabelit ngjyrën kafe. Dy rreshtat e fundit bëjnë të mundur **printimin e cicërimës dhe ngjyrës në Console**, si mund ta shohim dhe më poshtë.



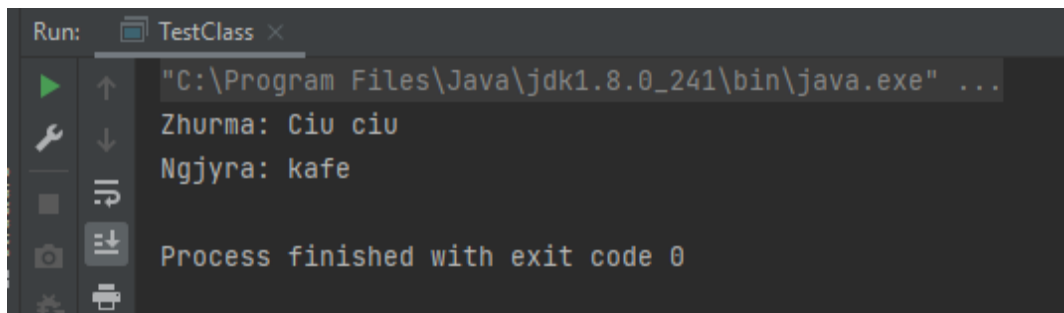


Fig. 4

## 4. Polimorfizmi (Polymorphism)

**Polimorfizmi** është i lidhur ngushtë me trashëgiminë, pra ndodh kur kemi trashëgimi. Ai vjen nga dy fjalë greke: *poly* = shumë dhe *morphs* = forma. Pra, *Polymorphism* = shumë forma

I referohet vetisë së një objekti që të shndërrohet në shumë forma të ndryshme. Më saktë një objekt mund të aksesohet duke përdorur një *reference me të njëjtin* tip si objekti, *një referencë e cila është një superclass* e objektit apo *një referencë që përcakton një ndërfaqe*(interface) që objekti implementon (direkt ose nga superclass).

Le të marrim një shembull për ta kuptuar më qartë. Në shembullin e mëparshëm kishim **klasën Shpend**. Në këtë shembull e ndryshojmë klasën Shpend si më poshtë.

```

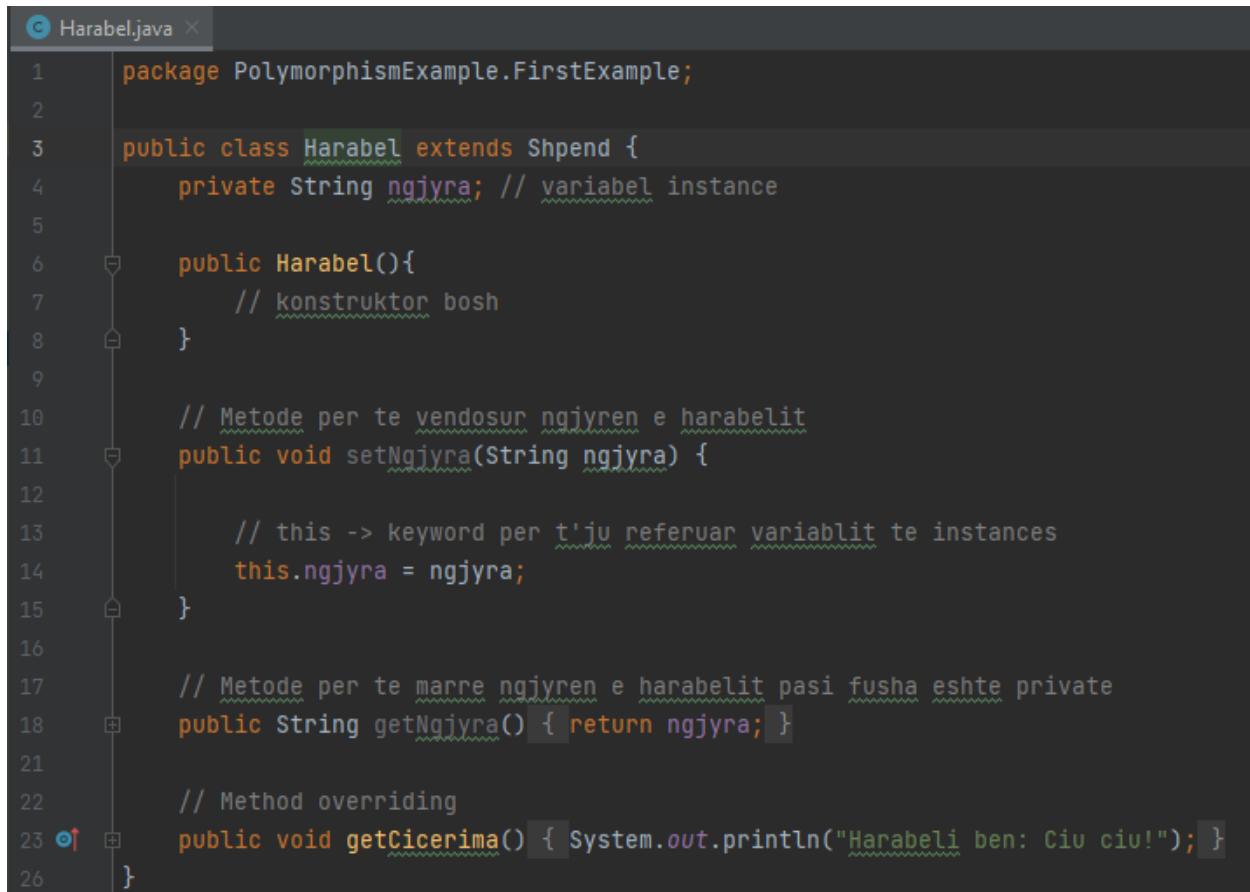
Shpend.java x
1 package PolymorphismExample.FirstExample;
2
3 public class Shpend {
4
5     public Shpend() {
6         // konstruktor bosh
7     }
8
9     public void getCicerima() { System.out.println("Shpendet cicerijne"); }
12 }

```

Fig. 5

Më pas shtojmë metodën **getCicerima()** në klasën Harabel. Krijojmë dhe një klasë tjetër të quajtur **Pellumb**, e cila implementon përsëri metodën **getCicerima()**. Ndryshojmë **klasën 'TestClass'** si më poshtë dhe bëjmë run programin.

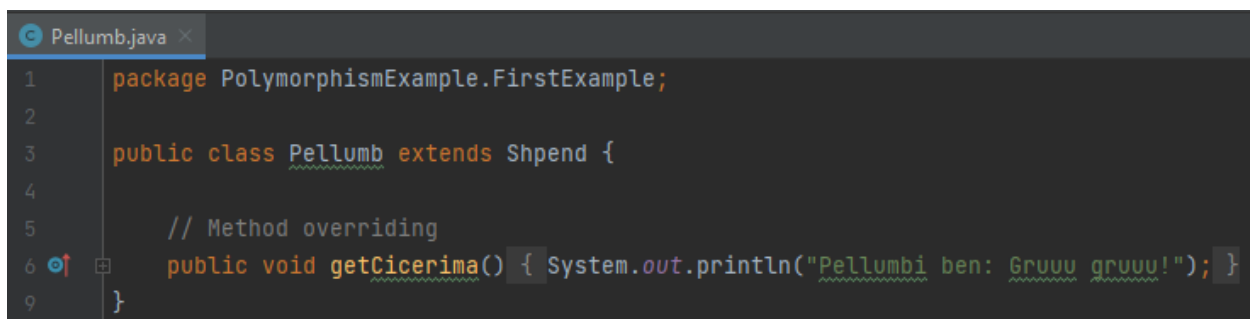
### Klasa Harabel (Harabel.java)

A screenshot of a code editor showing the implementation of the Harabel class. The file is named 'Harabel.java'. The code is in Java and defines a class 'Harabel' that extends 'Shpend'. It includes a private 'ngjyra' attribute, a constructor, a 'setNgjyra' method, a 'getNgjyra' method, and an overridden 'getCicerima' method.

```
1 package PolymorphismExample.FirstExample;
2
3 public class Harabel extends Shpend {
4     private String ngjyra; // variabel instance
5
6     public Harabel(){
7         // konstruktor bosh
8     }
9
10    // Metode per te vendosur ngjyren e harabelit
11    public void setNgjyra(String ngjyra) {
12
13        // this -> keyword per t'ju referuar variablit te instances
14        this.ngjyra = ngjyra;
15    }
16
17    // Metode per te marre ngjyren e harabelit pasi fusha eshte private
18    public String getNgjyra() { return ngjyra; }
19
20
21
22    // Method overriding
23    public void getCicerima() { System.out.println("Harabeli ben: Ciu ciu!"); }
24
25 }
```

Fig. 6

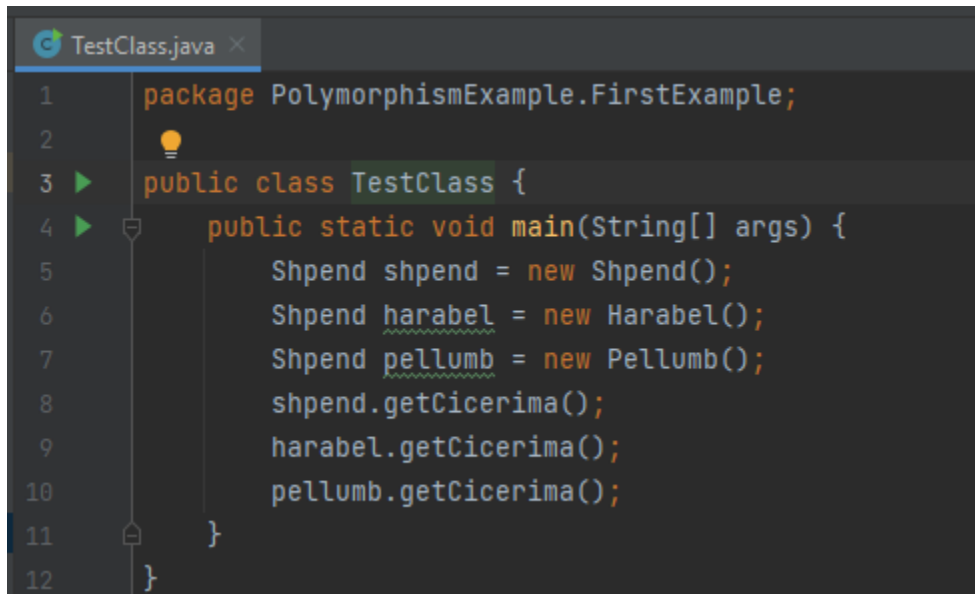
### Klasa Pellumb (Pellumb.java)

A screenshot of a code editor showing the implementation of the Pellumb class. The file is named 'Pellumb.java'. The code is in Java and defines a class 'Pellumb' that extends 'Shpend'. It includes a 'getCicerima' method that overrides the one in the parent class.

```
1 package PolymorphismExample.FirstExample;
2
3 public class Pellumb extends Shpend {
4
5     // Method overriding
6     public void getCicerima() { System.out.println("Pellumbi ben: Gruuu gruuv!"); }
7
8 }
9
```

Fig. 7

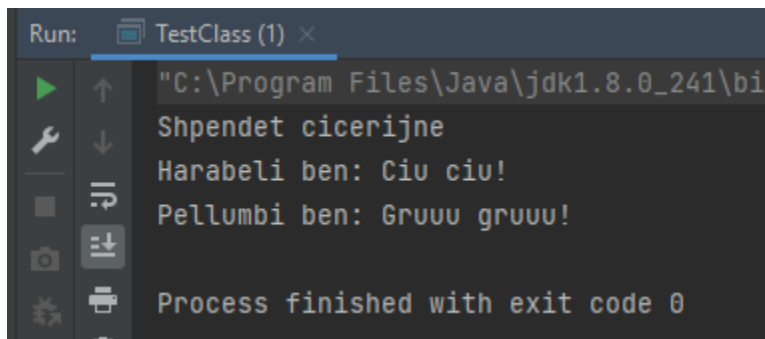
### Klasa TestClass (TestClass.java)



```
1 package PolymorphismExample.FirstExample;
2
3 public class TestClass {
4     public static void main(String[] args) {
5         Shpend shpend = new Shpend();
6         Shpend harabel = new Harabel();
7         Shpend pellumb = new Pellumb();
8         shpend.getCicerima();
9         harabel.getCicerima();
10        pellumb.getCicerima();
11    }
12 }
```

Fig. 8

Pasi i bëjmë run marrim rezultatin e mëposhtëm.



```
Run: TestClass (1) x
"C:\Program Files\Java\jdk1.8.0_241\bin
Shpendet cicerijne
Harabeli ben: Ciu ciu!
Pellumbi ben: Gruuu gruuv!
Process finished with exit code 0
```

Fig. 9

Pra, si mund të kuptohet nga output i përfituar, **polimorfizmi** na lejon që një detyrë të caktuar ta kryejmë në disa mënyra të ndryshme.

Polimorfizmi në Java ka dy lloje:

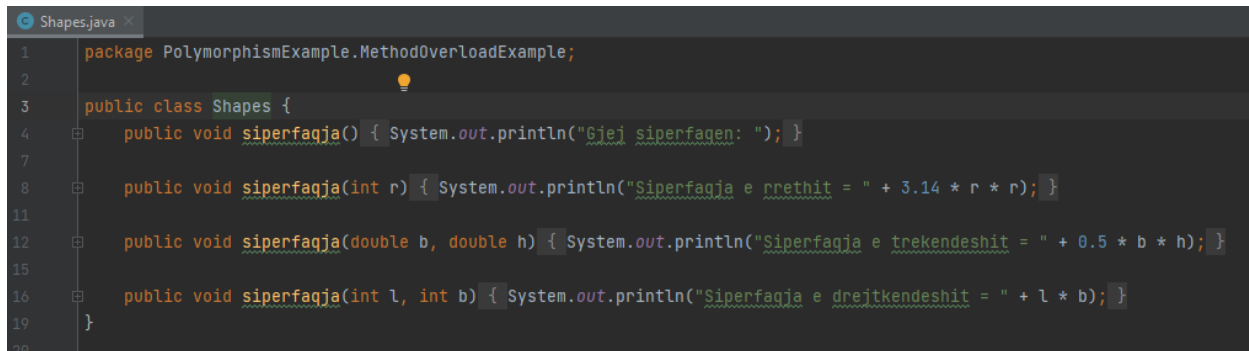
- compile-time polymorphism
- run-time polymorphism

## 4.1 Compile-time polymorphism

Në këtë proces, thirrja e metodës kryhet gjatë kohës së kompilimit (compile-time). Kjo në Java arrihet përmes **Method Overloading**. *Method Overload* ndodh kur një klasë ka disa metoda me të njëjtin emër, por numri, tipi dhe renditja e parametrave dhe return type e metodave janë të ndryshme. Për të shpjeguar më qartë Method Overloading shohim shembullin e mëposhtëm:

Kemi **klasën Shapes** ku metoda **siperfaqja()** është implementuar katër herë.

### Klasa Shapes (Shape.java)



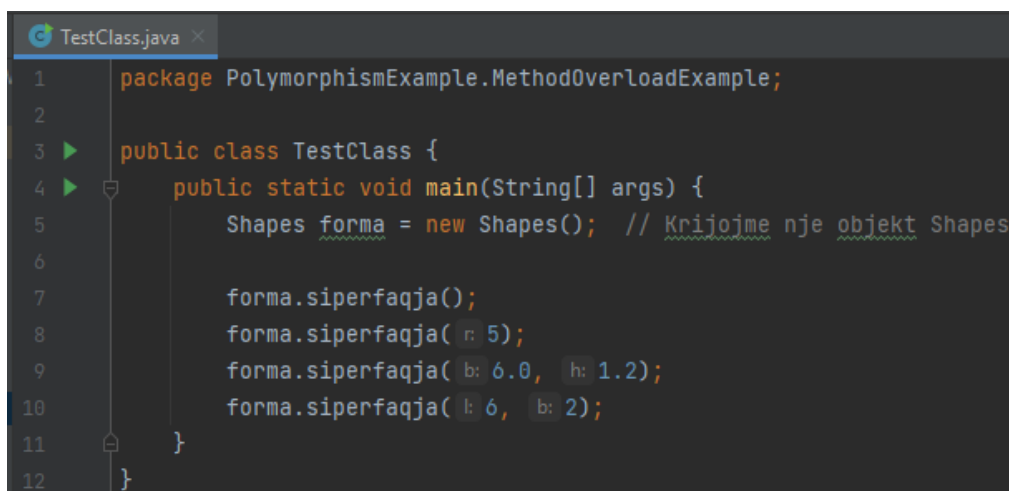
```
1 package PolymorphismExample.MethodOverloadExample;
2
3 public class Shapes {
4     public void siperfaqja() { System.out.println("Gjej siperfaqen: "); }
5
6     public void siperfaqja(int r) { System.out.println("Siperfaqja e rrethit = " + 3.14 * r * r); }
7
8     public void siperfaqja(double b, double h) { System.out.println("Siperfaqja e trekendeshit = " + 0.5 * b * h); }
9
10    public void siperfaqja(int l, int b) { System.out.println("Siperfaqja e drejtkendeshit = " + l * b); }
11
12 }
```

Fig. 10

Nëse metodat nuk do të kishin parametra do të shfaqej një mesazh i tillë: *'Ambiguous method call: 'Shapes.siperfaqja()' and 'Shapes.siperfaqja()' match'*. Kjo ndodh pasi metodat ndodhen në të njëjtën klasë dhe Java nuk mund të dallojë se cilën metodë të thërrasë.

Në **klasën TestClass** krijojmë një objekt **Shapes** dhe më pas thërrasim metodën **siperfaqja()** me parametra të ndryshëm.

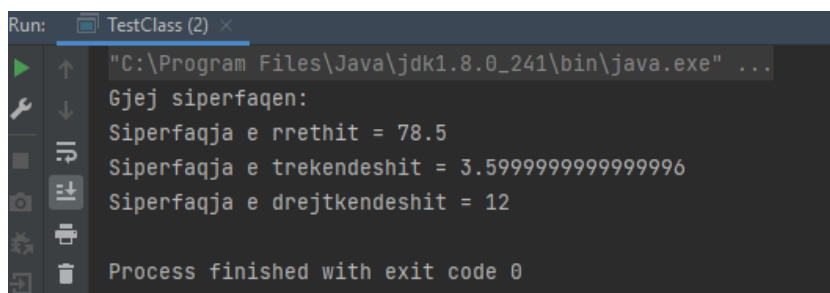
### Klasa TestClass (TestClass.java)



```
1 package PolymorphismExample.MethodOverloadExample;
2
3 public class TestClass {
4     public static void main(String[] args) {
5         Shapes forma = new Shapes(); // Krijojme nje objekt Shapes
6
7         forma.siperfaqja();
8         forma.siperfaqja(5);
9         forma.siperfaqja(6.0, 1.2);
10        forma.siperfaqja(6, 2);
11    }
12 }
```

Fig. 11

Duke parë outputin e shfaqur në Console dallojmë se cila metodë është ekzekutuar për çdo rresht.



```
Run: TestClass (2) x
"C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
Gjej siperfaqen:
Siperfaqja e rrethit = 78.5
Siperfaqja e trekendeshit = 3.5999999999999996
Siperfaqja e drejtkendeshit = 12
Process finished with exit code 0
```

Fig. 12

## 4.2 Run-time polymorphism

Në këtë proces, thirrja për një metodë të bërë Override kryhet në mënyrë dinamike në kohën e ekzekutimit (run-time) dhe jo në kohën e kompilimit. **Polimorfizmi në runtime** arrihet përmes **Method Overriding**. *Method Overriding* ndodh kur një **child class / subclass** ka një metodë me të njëjtin emër, parametra dhe return type si **klasa prind/superclass**, atëherë ai funksion i bën override funksionit në superclass. Në terma më të thjeshtë, nëse **nënklasa** jep përkufizimin e saj për një metodë të pranishme në **superclass**, atëherë ai funksion në superclass thuhet se është overridden.

*Shembulli i mëparshëm me klasat Shpend, Harabel dhe Pellumb e tregojnë më së miri si ndodh **Method Overriding** me anë të metodës **getCicerima()**.*

## 5. Abstraksioni (Abstraction)

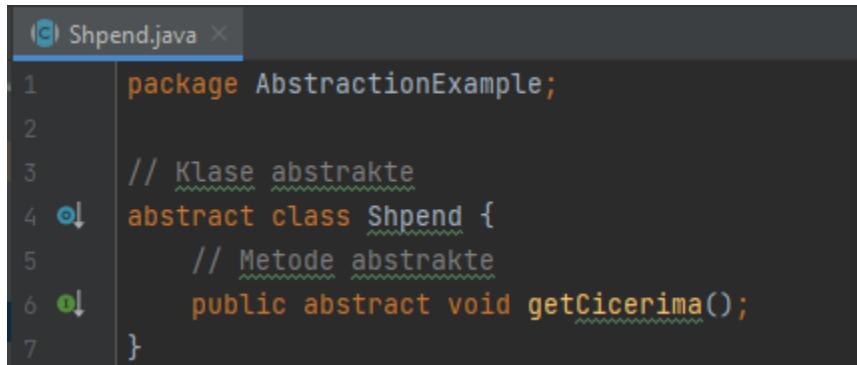
Abstraksioni është një proces i cili shfaq vetëm informacionin e nevojshëm dhe fsheh informacionin e panevojshëm. Mund të themi se qëllimi kryesor i abstraksionit është fshehja e të dhënave nga përdoruesi. Për shembull, kur shohim një makinë në rrugë, ne shohim vetëm se si ajo duket nga jashtë, nuk shohim komponentët e saj individual të brendshëm.

Në Java kemi klasa dhe metoda abstrakte. Një **klasë abstrakte** (abstract class) është një lloj klase e cila deklaron një ose më shumë metoda abstrakte. Një **metodë abstrakte** (abstract method) është një metodë e cila ka definimin e metodës, por jo implementimin. **Metodat abstrakte** përdoren kur dy ose më shumë nënklasa kryejnë të njëjtën detyrë në mënyra të ndryshme dhe përmes zbatimeve të ndryshme. Një klasë abstrakte mund t'i ketë të dyja metodat, pra, *metoda abstrakte dhe metoda të rregullta*.

**KUJDES! Klasa abstrakte duke qenë e pa-plotë (metodat abstrakte të paimplementuara), nuk mund të inicializohet, pra nuk mund të krijojmë një objekt të saj.**

Shohim shembullin me klasën Shpend dhe Harabel. Në këtë shembull *klasa Shpend* është deklaruar si **klasë abstrakte** dhe përmban **metodën abstrakte** *getCicerima()*. Siç vihet re, metoda *getCicerima()* nuk është implementuar.

### Klasa Shpend (Shpend.java)

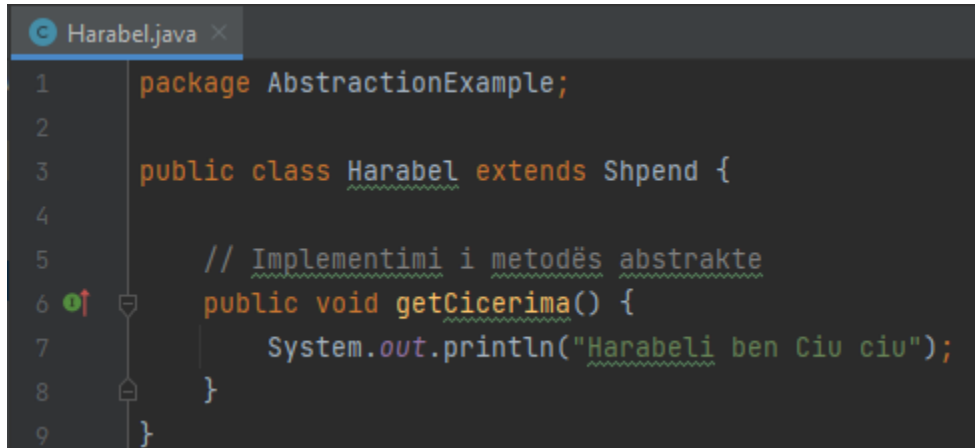
A screenshot of a code editor showing the Shpend.java file. The code defines an abstract class Shpend in the package AbstractionExample. It includes a comment indicating it's an abstract class and a comment for the abstract method getCicerima().

```
1 package AbstractionExample;
2
3 // Klase abstrakte
4 abstract class Shpend {
5     // Metode abstrakte
6     public abstract void getCicerima();
7 }
```

Fig. 13

Në klasën Harabel shohim implementimin e kësaj metode, ku printohet shkrimi "Harabeli ben Ciu ciu".

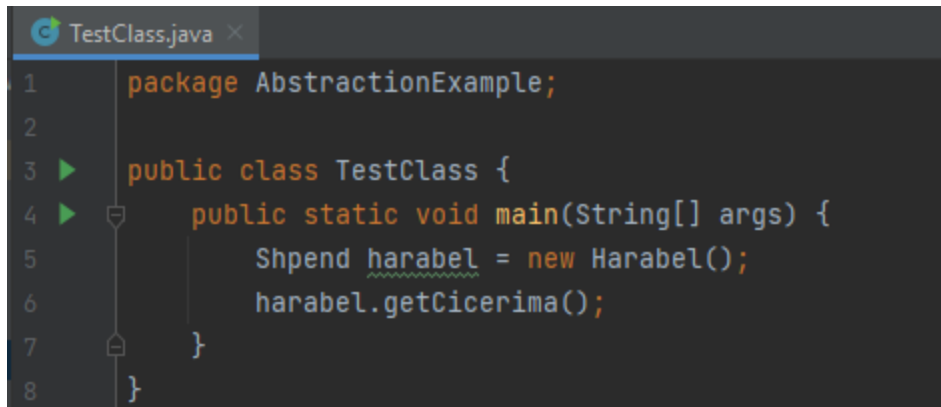
### Klasa Harabel (Harabel.java)

A screenshot of a code editor showing the Harabel.java file. The code defines a class Harabel that extends Shpend. It includes a comment indicating it's the implementation of the abstract method. The method getCicerima() prints the string "Harabeli ben Ciu ciu".

```
1 package AbstractionExample;
2
3 public class Harabel extends Shpend {
4
5     // Implementimi i metodës abstrakte
6     public void getCicerima() {
7         System.out.println("Harabeli ben Ciu ciu");
8     }
9 }
```

Fig. 14

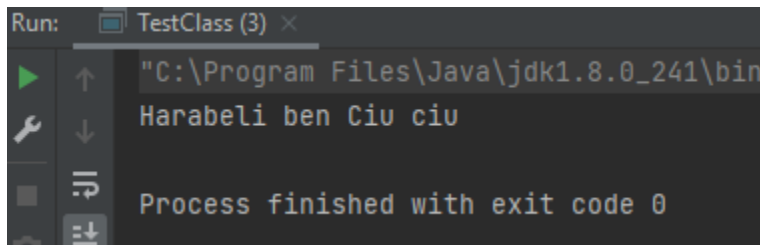
### Klasa TestClass (TestClass.java)



```
1 package AbstractionExample;
2
3 public class TestClass {
4     public static void main(String[] args) {
5         Shpend harabel = new Harabel();
6         harabel.getCicerima();
7     }
8 }
```

Fig. 15

Në **klasën TestClass** krijojmë objektin `harabel` dhe më pas thërrasim metodën e implementuar. Në console shfaqet outputi.



```
Run: TestClass (3) x
"C:\Program Files\Java\jdk1.8.0_241\bin\
Harabeli ben Ciu ciu
Process finished with exit code 0
```

Fig. 16

## 6. Enkapsulimi (Encapsulation)

**Enkapsulimi** përcaktohet si mbështjellja ose bashkimi i të dhënave dhe metodave, që operojnë në këto të dhëna, në një njësi të vetme. Koncepti themelor i enkapsulimit është *fshehja e implementimit të brendshëm të një objekti nga jashtë*. Kjo njihet edhe si fshehje e të dhënave. Në përgjithësi, enkapsulimi kufizon klasat e jashtme për të hyrë dhe modifikuar fushat dhe metodat e një klase. Aksesimi në këto variabla kryhet vetëm përmes metodave të klasës përkatëse.

### Si t'i enkapsulojmë të dhënat?

Nëse përdorim **access modifier 'private'** në një variabël/metodë të një klase atëherë ajo mund të aksesohet vetëm brenda të njëjtës klase. Për ta bërë një variabël private të aksesueshme jashtë klasesë së saj, përdorim *getters* dhe *setters* public (public). Në shembullin e shpendëve në **klasën Harabel** (Fig. 2) shohim metodat `setNgjyra()`, `getNgjyra()`. Variabli i instancës 'ngjyra' është deklaruar privat dhe për ta aksesuar përdoren dy metodat e parapërmendura.

## **Përfundim**

Në këtë shkrim shpjegohet çfarë është OOP dhe konceptet e saj kryesore.

Objekti përfaqëson një entitet fizik ose logjik të botës reale dhe është një instancë e klasës.

Klasa shërben si një model për krijimin e objektit.

Trashëgimia na lejon të ndërtojmë klasa duke u bazuar në klasa të tjera.

Polimorfizmi i referohet vetisë së një objekti që të shndërrohet në shumë forma të ndryshme.

Abstraksioni është një proces i cili shfaq vetëm informacionin e nevojshëm dhe fsheh informacionin e panevojshëm.

Enkapsulimi përcaktohet si mbështjellja ose bashkimi i të dhënave dhe metodave, që operojnë në këto të dhëna, në një njësi të vetme.