

Odissee
DE CO-HOGESCHOOL

Encapsulation en Overerving




Maarten Troost – Jens Baetens



Encapsulation

Wat is er de voorbije weken reeds gezegd over Data Encapsulation?

- ▣ Wat is het?
- ▣ Hoe kunnen we het gebruiken? Welk programmeerconcept wordt hiervoor gebruikt?
- ▣ Waarom gebruiken we het?



Data encapsulation

From Wikipedia, the free encyclopedia

Data encapsulation, also known as data hiding, is the mechanism whereby the implementation details of a class are kept hidden from the user. The user can only perform a restricted set of operations on the hidden members of the class by executing special functions commonly called *methods* to prevent attributes of objects from being easily viewed and accessed. **Data encapsulation** may refer to:

- The wrapping of private data in classes in object-oriented programming languages: see [Encapsulation \(object-oriented programming\)](#), [information hiding](#), [separation of concerns](#)
- The wrapping of network data by a lower layer in the [OSI model](#) into a single unit where a higher layer can extract the relevant data: see [Encapsulation \(networking\)](#)



Wat is het?

- ▣ Fundamenteel concept van Object Oriented Programming
- ▣ Bundelen van data en de methoden die werken op die data
 - Verbergen van een interne state of waarden in een klasse
 - Voorkom niet geautorizeerde toegang tot de waarden
 - Publieke methoden kunnen voorzien worden om gecontroleerd toegang te bieden



Waarom is het goed om het te gebruiken?

- ▣ Functionaliteit en data bevindt zich op 1 plaats/file ipv in meerdere
- ▣ Data in het object kan niet onverwachts aangepast worden door code in een ongerelateerde plaats in de applicatie
- ▣ Indien je gebruik wilt maken van een publieke methode, dan moet je enkel de definitie weten van de methode en niet hoe het object intern werkt
 - ▬ Interne code kan gewijzigd worden zonder externe verschillen te veroorzaken.
- ▣ Een object weet hoe zijn data moet behandeld worden
 - ▬ Slecht idee om die functionaliteit te omzeilen en het rechtstreeks te doen

Hoe kunnen we het gebruiken

- ▣ Groepeer data die logisch samenhoort in een klasse met de functies die inwerken op deze data
 - ▬ Bijvoorbeeld een student heeft een studentnummer, naam, voornaam, adres, ...
- ▣ Verberg interne state en waarden door gebruik te maken van het private keyword
- ▣ Voeg getters en setters toe voor de zaken die aangepast kunnen worden
- ▣ Gebruik package visibility en plaats samenwerkende classes in 1 package

Encapsulation Oefening 1

- Wat is de fout in deze code?
 - Bewijs je antwoord door een unit-test te schrijven die dit aantoont.
- Fix de bug in de code

```
public class Rectangle {  
  
    public double length;  
    public double width;  
    private double area;  
  
    public Rectangle(final double length, final double width){  
        this.width = width;  
        this.length = length;  
        area = length * width;  
    }  
  
    public double getArea(){  
        return area;  
    }  
}
```

Wat is de fout?

```
class RectangleTest {  
  
    @Test  
    void getArea_WithWidthChanged_AreaAlsoChanged() {  
        // Arrange  
        Rectangle rectangle = new Rectangle( length: 10, width: 5);  
        rectangle.width = 10;  
  
        // Act  
        double area = rectangle.getArea();  
  
        // Assert  
        assertEquals(area, actual: 100);  
    }  
}
```

Wat is de oplossing?

```
public class RectangleSolution {  
    private double length;  
    private double width;  
    private double area;  
  
    public RectangleSolution(final double length, final double width){  
        this.width = width;  
        this.length = length;  
        calculateArea();  
    }  
  
    private void calculateArea(){  
        area = length * width;  
    }  
  
    public void setLength(double length){  
        this.length = length;  
        calculateArea();  
    }  
}
```

```
    public void setWidth(double width){  
        this.width = width;  
        calculateArea();  
    }  
  
    public double getLength(){  
        return length;  
    }  
  
    public double getWidth(){  
        return width;  
    }  
  
    public double getArea(){  
        return area;  
    }  
}
```



Overervig

Wat denk je dat overerving inhoudt?

- ▣ Wat betekent overerving?
- ▣ Waarvoor wordt het gebruikt?
- ▣ Hoe kunnen we overerving in een applicatie toepassen?

Wat betekent overerving?

▣ Wikipedia:

In [object-oriented programming](#), **inheritance** is the mechanism of [basing an object or class upon another object](#) ([prototype-based inheritance](#)) or class ([class-based inheritance](#)), retaining similar [implementation](#). Also defined as deriving new classes ([sub classes](#)) from existing ones such as [super class or base class](#) and then forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance, a "child object", acquires all the properties and behaviors of the "parent object", with the exception of: [constructors](#), destructor, [overloaded operators](#) and [friend functions](#) of the base class. Inheritance allows programmers to create classes that are built upon existing classes,^[1] to specify [a new implementation while maintaining the same behaviors](#) ([realizing an interface](#)), [to reuse code and to independently extend original software via public classes and interfaces](#). The relationships of objects or classes through inheritance give rise to a [directed acyclic graph](#).

Inheritance was invented in 1969 for [Simula](#)^[2] and is now used throughout many object-oriented programming languages such as [Java](#), [C++](#), [PHP](#) and [Python](#).

Wat betekent inheritance of overerving?

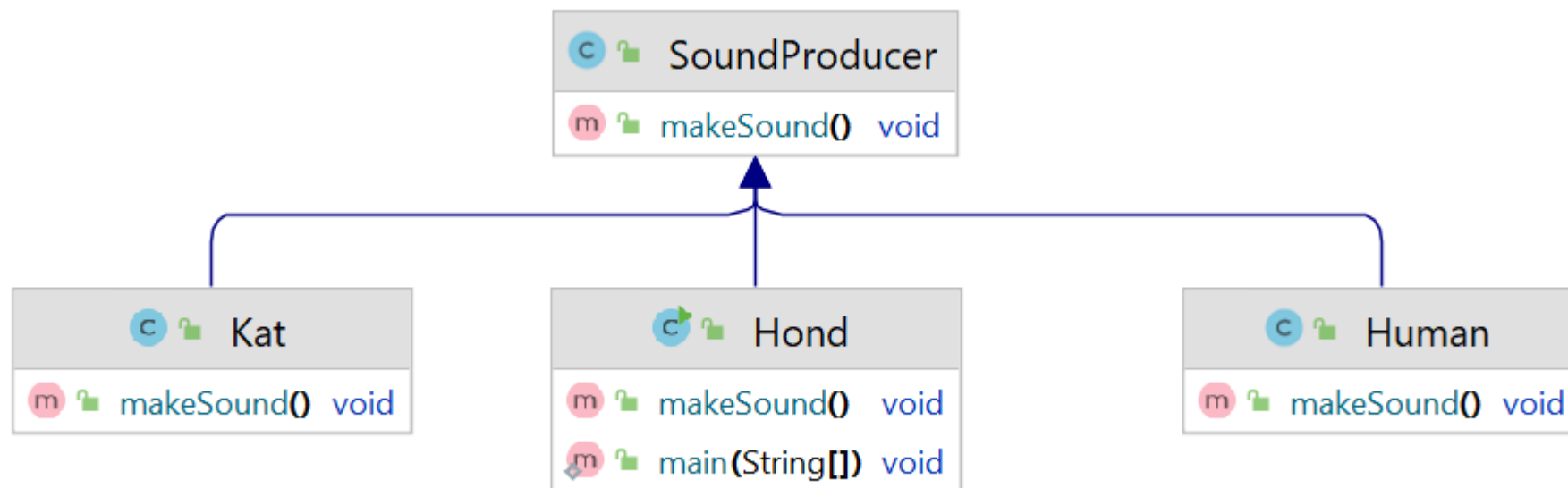
- ▣ Een klasse die de functionaliteit van een andere klasse uitbreidt
 - ▬ Door alle attributen en methoden over te erven
 - ▬ Extra functionaliteit toe te voegen
 - ▬ Bestaande methods te overschrijven/implementeren
- ▣ Single inheritance : Een klasse kan maar van max 1 klasse overerven
 - ▬ C# en Java
- ▣ Multiple inheritance: Een klasse kan van meerdere klassen overerven
 - ▬ Python en C++



Redenen voor overerving

- ▣ Hergebruik van code
- ▣ Open-Closed principe van de SOLID – principles
 - Open for extension, closed for modifications
- ▣ Abstractere code is gemakkelijker voor mensen
- ▣ Flexibelere applicatie door gemakkelijker wisselen van objecten om werking van de code aan te passen

Hoe overerving toe passen in Java?



Hoe overerving toepassen in Java?

```
public static void main(String[] args) {  
    SoundProducer[] noisyNeighbours = new SoundProducer[]{  
        new Human(),  
        new Kat(),  
        new Hond(),  
        new SoundProducer()  
    };  
    for(SoundProducer soundProducer : noisyNeighbours){  
        soundProducer.makeSound();  
    }  
}
```

Een subclasse is ook van het type van zijn parent

```
public class SoundProducer {  
  
    public void makeSound(){  
        System.out.println("Some generic sound");  
    }  
}
```

Parent class

Extends keyword = overerving
max 1 class in java

```
public class Human extends SoundProducer {  
  
    @Override  
    public void makeSound(){  
        System.out.println("Ik ben een mens");  
    }  
}
```

Child class

@Override -> methode aangepast in child

De functies van de Parent-klasse gebruiken

■ Via het keyword super

▬ Constructor

- super()
- super({parameter list})

▬ Methods

- Super.makeSound()

Wat is de output hiervan?

```
public class Hond extends SoundProducer {  
  
    @Override  
    public void makeSound(){  
        super.makeSound();  
        System.out.println("Woof woof");  
    }  
  
    public static void main(String[] args) {  
        Hond hond = new Hond();  
        hond.makeSound();  
    }  
}
```

De functies van de Parent-klasse gebruiken

▣ Via het keyword super

▬ Constructor

- `super()`
- `super({parameter list})`

▬ Methods

- `Super.makeSound()`

Wat is de output hiervan?

```
public class Hond extends SoundProducer {  
  
    @Override  
    public void makeSound(){  
        super.makeSound();  
        System.out.println("Woof woof");  
    }  
  
    public static void main(String[] args) {  
        Hond hond = new Hond();  
        hond.makeSound();  
    }  
}
```

```
Some generic sound  
Woof woof
```

```
Process finished with exit code 0
```

SOLID – principes van software architectuur

- ▣ S – Single Responsibility Principle
- ▣ O – Open-closed principle
- ▣ L – Liskov Substitution principle
 - A derived class should be substitutable for their base class
 - Resulteert in meer hergebruik van code en een lossere koppeling
- ▣ I – Interface Segregation Principle
- ▣ D – Dependency inversion principle

Welke overerving is toegelaten?

- ▣ Een overerving moet altijd een is-een relatie hebben:
Een tafel is een meubel -> `class Tafel extends Meubel`
Een tafel is GEEN stoel -> ~~`class Tafel extends Stoel`~~

//A list is an ordered collection of elements allowing access to any element

```
class List {}
```

//A Queue is an ordered collection of elements allowing to read only the first element

```
class Queue extends List { }
```

```
class GoodQueue { //composition over inheritance  
    private List elements;  
}
```

Welke overerving is toegelaten?

```
class Cat {  
    private String name;  
    private String chipID;  
    private Race primaryRace;  
    private int height,width, length;  
    private Color primaryColor, secondaryColor, eyeColor;  
    public void move(int distance) {}  
    public void landOnFeet() {}  
    public HairBall HarkHaaaaark() throws UnsupportedOperationException {}  
}  
class Dog extends Cat {  
    public boolean fetch() {}  
    @Override public HairBall HarkHaaaaark() throws UnsupportedOperationException {  
        throw new UnsupportedOperationException();  
    }  
}
```

SOLID – principes van software architectuur

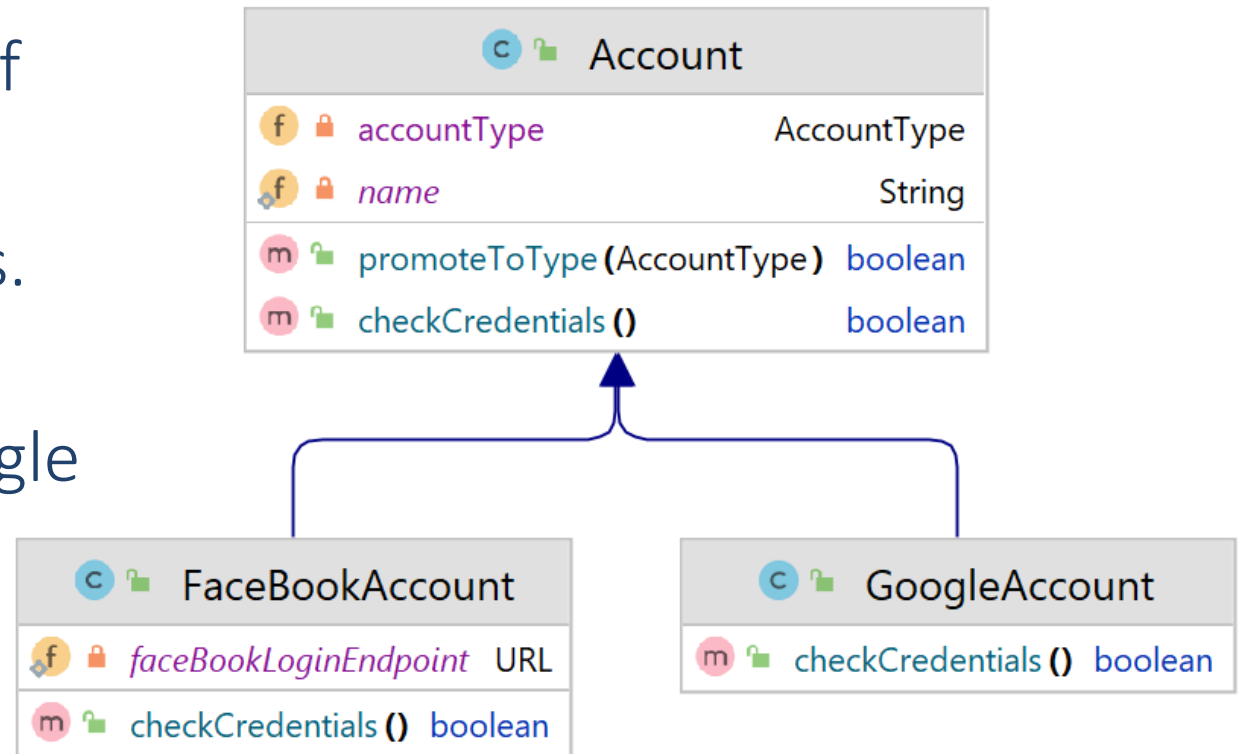
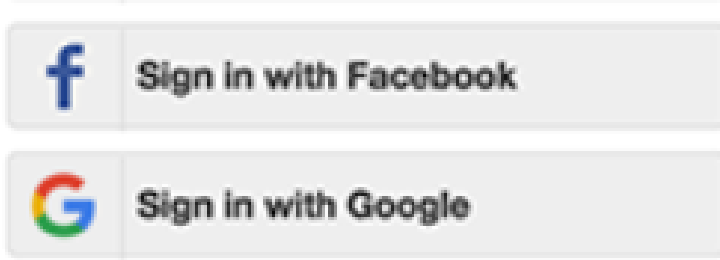
- ▣ S – Single Responsibility Principle
- ▣ O – Open-closed principle
- ▣ L – Liskov Substitution principle
- ▣ I – Interface Segregation Principle
- ▣ D – Dependency inversion principle
 - ▣ A higher level module should not depend on a lower level module but on abstractions of it

Abstracte code >> specifieke code

Een applicatie biedt de mogelijkheid om in te loggen met een facebook of google account.

Account is de superclass/parentclass.

In de view bestaat een UI voor facebook en een aparte UI voor google logins.



Abstracte code >> specifieke code

```
{  
    //DO AVOID using subclasses  
    FaceBookAccount userAccount=new FaceBookAccount(txt_userName.getText());  
    if(userAccount.checkCredentials()==true) {  
        LoginController.setStatusMessage("FaceBook login successful");  
    }  
}  
{  
    //DO use high level abstractions/parent classes  
    Account userAccount=new Account(txt_userName.getText());  
    if(userAccount.checkCredentials()==true) {  
        LoginController.setStatusMessage(loginMethod.getName()+" login successful");  
    }  
}
```

Overerving en final

- ▣ Een Class kan ook als **final** gemarkeerd worden. Het betekent dan dat er van deze klasse niet meer kan overgeërfd worden.
- ▣ Vb. `public final class Employee { ... }`
`public class ExperiencedEmployee extends Employee { ... }`
`"Cannot inherit from final Employee"`
- ▣ `java.lang.String` is een klasse die in de JDK als final werd gedeclareerd.

Overerving en final

- Een method kan ook als **final** gemarkeerd worden. Het betekent dan dat er van geen **@Override** kan gebeuren op deze method.

```
public class Employee {  
    private int age;  
    public final int getAge() {  
        return age;  
    }  
}
```

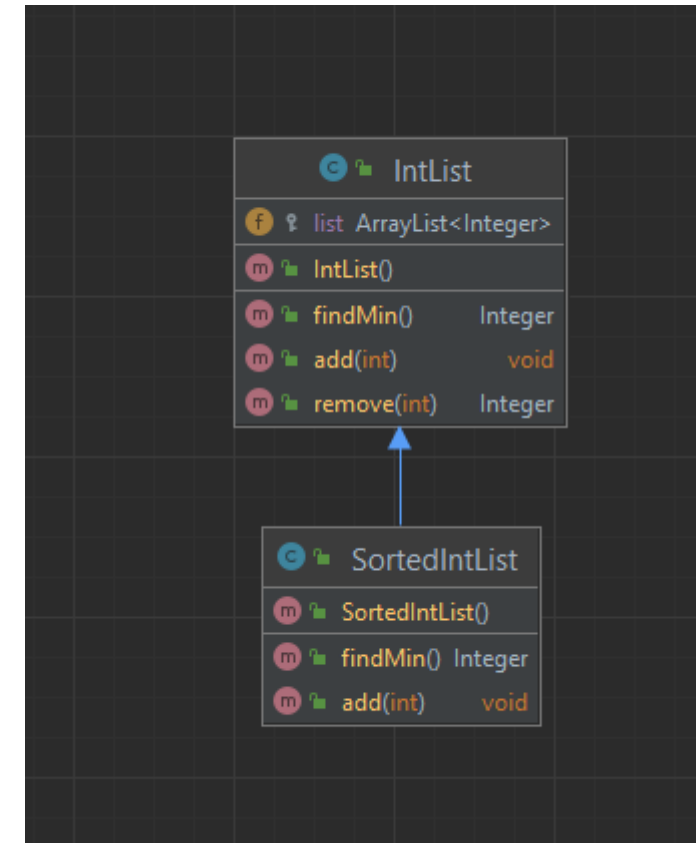
```
class TimeDefyingEmployee {  
    @Override public final int getAge() {  
        return -1;  
    }  
}
```



java: method does not override or
implement a method from a supertype

Inheritance oefening

- Maak het klasse diagram hiernaast na
 - ▬ Gebruik een ArrayList van Integers
 - ▬ IntList houdt de getallen bij in volgorde van toevoegen
 - ▬ SortedIntList houdt de getallen gesorteerd bij
- add() -> voeg element toe aan de lijst
- remove(index) -> verwijder element op die plaats
- findMin() -> zoek het minimum
- Hergebruik zoveel mogelijk code
- Test beide klassen zo grondig mogelijk





Abstract classes and methods

Abstract keyword

- ▣ Keyword op een methode of klasse welke zelf niet “gebruikt” kan worden
 - Kan zowel aan klassen als aan methoden toegekend worden
- ▣ Abstracte klasse
 - Er kunnen geen objecten aangemaakt worden van deze klasse
 - Deze klasse moet van overgeerfd worden om te kunnen gebruiken
- ▣ Abstracte methode
 - Een methode zonder implementatie dus zonder code block
 - Kan enkel gebruikt worden in een abstracte klasse
 - Geeft aan dat een subklasse deze functie **moet** implementeren

Abstract – Wanneer gebruiken?

- ▣ De parentclass kan niet compleet zijn
 - Als deze te abstract is
 - vb een animal is te abstract; elk dier is altijd een object van een subclass van animal
 - omdat gedrag ontbreekt
- ▣ De parentclass bevat reeds een deel van implementatie en alle subclasses moeten hetzelfde idee volgen



Abstract - voorbeeld

- ▣ Maak van de klasse SoundProducer een abstract klasse
- ▣ De functie makeSound moet ook een abstracte methode zijn

Abstract - oplossing

```
public abstract class SoundProducer {  
  
    protected String name;  
  
    public SoundProducer(String name){  
        this.name = name;  
    }  
  
    public abstract void makeSound();  
}
```

```
public class Dog extends SoundProducer {  
  
    public Dog(final String name) { super(name); }  
  
    @Override  
    public void makeSound() {  
        System.out.println(name + ":\tWoof");  
    }  
}  
  
public class Main {  
  
    public static void main(String[] args) {  
        SoundProducer dog = new Dog(name: "Charlie");  
        dog.makeSound();  
    }  
}
```