



Java traineeship

Working with inheritance



Leerdoelen

- Het begrijpen en het implementeren van overerving (inheritance)
- Het ontwikkelen van code waarin polymorfisme in wordt toegepast
- Het verschil weten tussen een referentie en een object
- Bepalen of er gebruik moet worden gemaakt van casting
- `Super` en `this` operatoren kunnen gebruiken
- Gebruik maken van `abstract` klassen en `interfaces`
- Schrijf een eenvoudige Lambda-uitdrukking die een Lambda-predikaatuitdrukking verbruikt

Inheritance

Een van de kernprincipes van objectgeoriënteerd programmeren

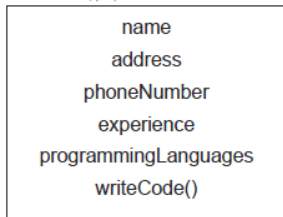
- **overerving** - stelt ons in staat om bestaande code te hergebruiken of een bestaand type uit te breiden.

Simpel gezegd, in Java kan een klasse een andere klasse en meerdere interfaces erven, terwijl een interface andere interfaces kan erven.

Waarom hebben we het nodig?



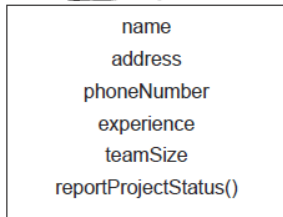
Programmer



```
class Programmer {  
    String name;  
    String address;  
    String phoneNumber;  
    float experience;  
    String[] programmingLanguages;  
    void writeCode() {}  
}
```



Manager



```
class Manager {  
    String name;  
    String address;  
    String phoneNumber;  
    float experience;  
    int teamSize;  
    void reportProjectStatus() {}  
}
```

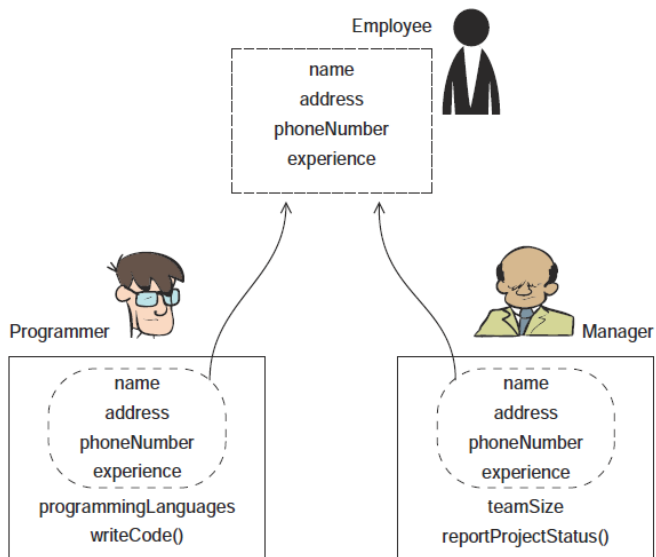
Hoewel de programmeur en de manager twee andere werknemers zijn, hebben ze een aantal elementen met elkaar gemeen:

- Naam
- Adres
- Telefoonnummer
- Ervaring
- ...

Sommige elementen zijn juist weer specifiek voor deze type werknemer zoals:

- Programmeertalen
- Schrijf code
- Team grootte
- Project status

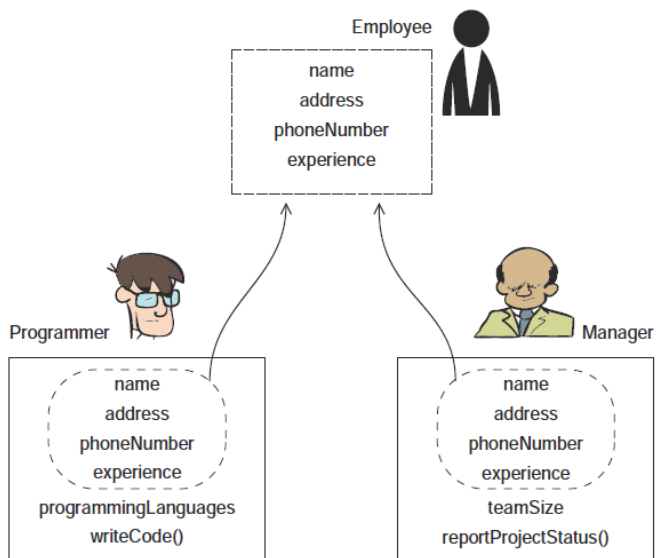
Waarom hebben we het nodig?



We kunnen daarom een generiek object maken die deze algemene informatie verzameld. Dit kan het object 'werknemer' zijn. Vervolgens kan een 'programmeur' of een 'manager' elementen uit dit object erven.

Hoe zou de code hier van uit zien denk je?
(reminder: Werknemer, Programmeur en Manager zijn allemaal klassen)

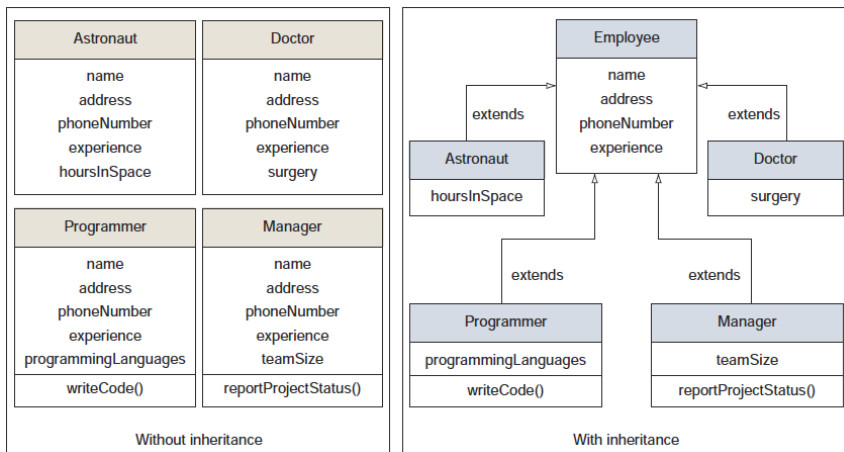
Waarom hebben we het nodig?



```
class Werknemer {  
    String naam;  
    String adres;  
    String telefoonnummer;  
    float ervaring;  
}  
  
class Programmeur extends Werknemer {  
    String[] programmeerTalen;  
    void programmeer() {  
        // code  
    }  
}  
  
class Manager extends Werknemer {  
    int teamGrootte;  
    void projectStatus() {  
        // code  
    }  
}
```

Voordelen van overerving: Kleinere afgeleide klassedefinities

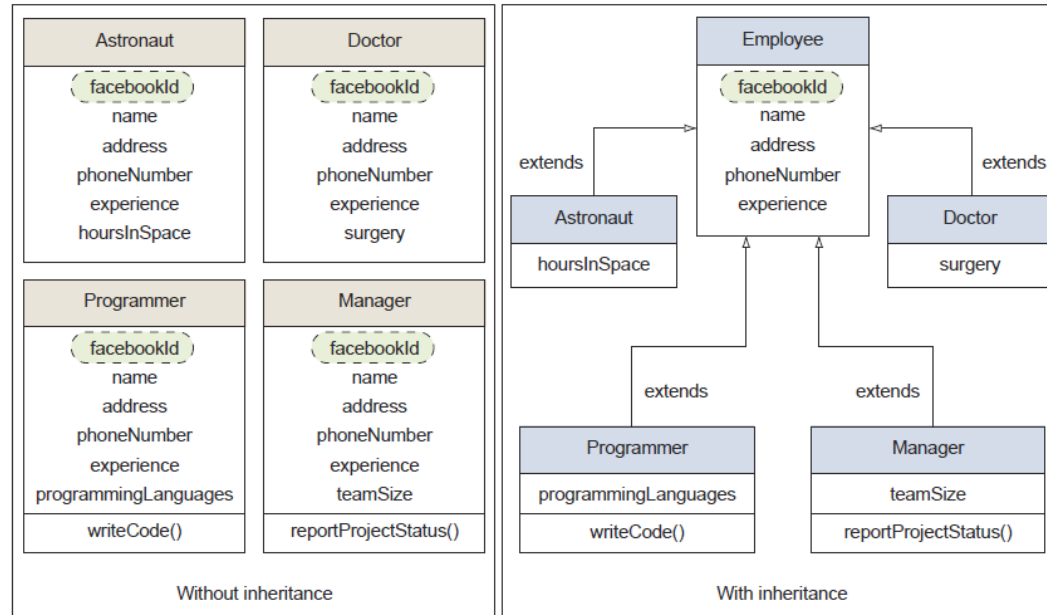
In de vorige slides ging het over een Programmeur en een Manager. Hierbij zagen we dat we algemene informatie zoals de naam kunnen splitsen in een generiek object. Dit is vooral handig als je veel verschillende objecten hebt die zo een naam hebben:



Voordelen van overerving:

Makkelijk aanpasbaarheid van de gedeelde eigenschappen

Als we bijvoorbeeld meer informatie of andere informatie willen opslaan over onze werknemers hoeven we maar één object aan te passen in plaats van alle.



Voorbeeld: De product klasse

```
class Product {  
    private static final int BTW_PERCENTAGE = 21;  
    int prijsEx;  
  
    public int getPrijsInc() {  
        return (int) (prijsEx * (1.0 + BTW_PERCENTAGE /  
100.0));  
    }  
  
    public String toString() {  
        return "Product met prijs: " + (getPrijsInc() /  
100.0);  
    }  
}
```

Voorbeeld: De product klasse (2)

Deze klasse laat zich prima instantiëren tot een object:

```
Product zakHooi = new Product();  
zakHooi.prijsEx = 3000;  
System.out.println(zakHooi);  
// Product met prijs: 36.3
```

Legokasteel

We kunnen nu een stuk speelgoed definiëren:

```
class Legokasteel extends Product {  
  
}
```

Het keyword **extends** zegt hier dat Product een **superklasse** van Legokasteel is.

Vervolgend kunnen we een nieuwe instantie aanmaken:

```
Legokasteel legokasteel = new Legokasteel();  
legokasteel.prijsEx = 10000;  
System.out.println(legokasteel);  
// Product met prijs: 121.0
```

Er staat hier nog steeds product

Overriding van methoden

De toString van LegoKasteel wordt nu nog geërfd van de Product klasse.

Als we onze eigen toString() willen definiëren van vervangen we daarmee de toString-methode van de superklasse.

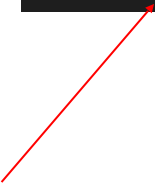
Dit concept van vervangen van al bestaande methodes in de superklassen noemen we **overriding**.

Legokasteel met overridden toString()

```
class Legokasteel extends Product {  
    public String toString() {  
        return "Legokasteel met prijs: " + (getPrijsInc() / 100.0);  
    }  
}
```

Als we vervolgens dezelfde code als net aanroepen:

```
Legokasteel legokasteel = new Legokasteel();  
legokasteel.prijsEx = 10000;  
System.out.println(legokasteel);  
// Legokasteel met prijs: 121.0
```



Overriding

Wat zien we gebeuren?

- De Product klasse bevat de variabele PrijsEx en de getPrijsInc-methode
- LegoKasteel heeft een nieuwe definitie van de toString() methode
 - De subclass (LegoKasteel) heeft de toString-methode van de superclass **overriden**

Overloading vs Overriding

Het is ook prima mogelijk om een methode van de superklasse te overladen, je maakt dan een extra definitie met een andere signatuur aan. Bijvoorbeeld:

```
class LegoKasteel extends Product {  
    public String toString() {  
        return "Legokasteel met prijs: " + (getPrijsInc() / 100.0);  
    }  
    public String toString(int aantal) {  
        return aantal + " legokastelen met een totaalprijs van: " + (getPrijsInc(aantal) / 100.0);  
    }  
    public int getPrijsInc(int aantal) {  
        return aantal * getPrijsInc();  
    }  
}  
  
LegoKasteel legoKasteel = new LegoKasteel();  
legoKasteel.prijsEx = 10000;  
System.out.println(legoKasteel.toString(2)); // 2 legokastelen met een totaalprijs van: 242.0
```

Overriding en overloading

Tot zover nog wel logisch toch?

- We kunnen extra methoden toevoegen aan de subklasse
- We kunnen bestaande methoden uitbreiden: **overloading**
- We kunnen bestaande methoden vervangen: **overriding**

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++){
            System.out.println("woof ");
        }
    }
}
```

Same Method Name,
Different Parameter

Overriding en overloading

Opdracht:

Creëer een programma met een **hoofd**klasse 'vogel' met een aantal **kind**klassen die de eigenschappen van de vogel erven. Maak daarbij gebruik van overriding en overloading.

Abstracte klassen

Abstracte klasse hebben de volgende eigenschappen:

- Ze zijn niet instantieerbaar
- Bevatten zowel methoden **met** implementatie
- Als methoden **zonder** een implementatie: *abstracte methoden*

@override

Als je bewust een methode wilt overriden mag je ook de decorator @override boven de definitie van de methode plaatsen.

De compiler waarschuwt je dan als jouw methode niet een andere methode override.

Override voorbeeld

```
class Spielgoed {  
    public String toString() {  
        return "Spielgoed!";  
    }  
}  
  
class Bordspel extends Spielgoed {  
    public String toString() {  
        return "Bordspel";  
    }  
}
```

Een tikfout is snel gemaakt! De **toString**-methode van de Bordspel klasse is nu net anders dan die van de Spielgoed klasse.

Override voorbeeld (2)

```
class Spielgoed {  
    public String toString() {  
        return "Spielgoed!";  
    }  
}  
  
class Bordspel extends Spielgoed {  
    @Override  
    public String toString() {  
        return "Bordspel";  
    }  
}
```

```
error: method does not override or implement a method from a supertype  
    @Override  
    ^  
1 error
```

Constructoren met overerving

Als je een constructor definieert, **garandeert** java dat deze wordt aangeroepen. Normaal gesproken gebeurde dit direct bij het aanmaken van een nieuw object.

Maar hoe zit dat met overerving?

De constructor van een subklasse moet altijd de constructor van de superklasse aanroepen. Indien je dit niet zelf doet, zal Java dit zelf doen!

Constructoren met overerving

Aanschouw de volgende klassen:

```
class Persoon {  
    String naam;  
    public String toString() {  
        return naam;  
    }  
}  
class Student extends Persoon {  
    int studentnummer;  
}  
class Trainee extends Student {  
    int oefentoetsCijfer;  
}
```

Let op: iedere Trainee heeft dus ook een studentnummer en een naam.

Constructoren met overerving

Nu gaan we langzaam aan alle klassen een constructor toevoegen:

```
class Persoon {  
    String naam;  
    public String toString() {  
        return naam;  
    }  
    Persoon(String naam) {  
        System.out.println("Persoon constructor");  
        this.naam = naam;  
    }  
}
```

Compileert deze code?

Constructoren met overerving

Nu gaan we langzaam aan alle klassen een constructor toevoegen:

```
class Persoon {  
    String naam;  
    public String toString() {  
        return naam;  
    }  
    Persoon(String naam) {  
        System.out.println("Persoon constructor");  
        this.naam = naam;  
    }  
}
```

```
error: constructor Persoon in class Persoon cannot be applied to gi[.]  
class Student extends Persoon {  
^  
    required: String  
    found: no arguments  
    reason: actual and formal arguments lists differ in length  
1 error
```

Waarom compileert de code niet?

De Student-klasse heeft geen constructor, dus voegt Java impliciet een “lege” constructor toe:

```
class Student extends Persoon {  
    int studentnummer;  
  
    Student() {  
        super();  
    }  
}
```

Waarbij `super()` een aanroep naar de constructor van de super klasse is.
Omdat `Persoon` nu geen constructor zonder argumenten meer heeft, werkt dit niet.

Waarom compileert de code niet?

Een mogelijke oplossing lijkt het zelf schrijven van een Student-constructor:

```
class Student extends Persoon {  
    int studentnummer;  
  
    Student(int studentnummer) {  
        this.studentnummer = studentnummer;  
    }  
}
```

```
error: constructor Persoon in class Persoon cannot be applied to gi[...]  
class Student(int studentnummer) {  
    ^  
    required: String  
    found: no arguments  
    reason: actual and formal arguments lists differ in length  
1 error
```

Waarom compileert de code niet?

Maar ook dit compileert niet, omdat Java de code van de vorige slide **impliciet** verbouwt naar het volgende:

```
class Student extends Persoon {  
    int studentnummer;  
    Student(int studentnummer) {  
        super();  
        this.studentnummer = studentnummer;  
    }  
}
```

Wat nog steeds niet werkt omdat er nog steeds geen lege Persoon constructor is.

Wat observeren we?

- Java voegt een aanroep naar **super()** toe in je constructoren!
- Tenzij je zelf aanroep naar een constructor van de super-klasse toevoegt!

De oplossing!

```
class Student extends Persoon {  
    int studentnummer;  
  
    Student(String naam, int studentnummer) {  
        super(naam);  
        this.studentnummer = studentnummer;  
    }  
}
```

Zelf proberen: vogel.java

Opdracht:

We vullen onze code van de Vogel.java en de andere rassen aan, door constructoren toe te voegen.

Daag jezelf uit door bijvoorbeeld in Parkiet.java een constructor toe te voegen die twee parameters nodig heeft, terwijl je maar één parameter nodig hebt voor de Vogel.java klasse (zie vorige slides)

Visibility modifiers

Zoals eerder besproken zijn er 4 **visibility modifiers**:

1. Public
2. Private
3. Protected
4. Package-private

Modifier	Class	Package	Subclass	World
public	X	X	X	X
protected	X	X	X	-
geen modifier	X	X	-	-
private	X	-	-	-

Private variabelen

Private variabelen kunnen dus niet door een subklasse benaderd worden:

```
class Persoon {  
    private String naam;  
}  
  
class Student extends Persoon {  
    int studentnummer;  
    public String toString() {  
        return naam;  
    }  
}
```

Error: naam has private access in Persoon

return naam;

^

1 error

Protected

Als je een methode of variabele dus voor de buitenwereld wilt afschermen, maar wel zichtbaar wil maken voor subklassen moet je de variabele dus **protected** maken.

Access modifiers met overriden

Je mag als je een methode override enkel de zichtbaarheid **vergroten**

Een **public** methode in de superklasse mag je dus niet overriden met een **private** methode in de subklasse

Public methode private overriden

```
class Persoon {  
    protected String naam;  
    public String getNaam() {  
        return naam;  
    }  
}  
  
class Student extends Persoon {  
    int studentnummer;  
    private String getNaam() {  
        return naam + " " + studentnummer;  
    }  
}
```

error: getNaam() in student cannot override getNaam() in Persoon
private String getNaam()
 ^
attempting to assign weaker access privileges; was public
1 error

Overriding en hiding

Bij methoden zagen we dat een nieuwe definitie een bestaande override, onafhankelijk van het type van de referentie.

```
class Product {  
    static int prijs = 300;  
    static String getNaam() {  
        return "Product";  
    }  
}  
  
class Armband extends Product {  
    static int prijs = 500;  
    static String getNaam() {  
        return "Armband";  
    }  
}
```

Static dynamic binding

```
Armband armbandje = new Armband();
System.out.println(armbandje.prijs);           // 500
System.out.println(Armband.prijs);             // 500
System.out.println(((Product) armbandje).prijs); // 300
System.out.println(Product.prijs);             // 300

Armband armbandje = new Armband();
System.out.println(armbandje.getNaam());        // Armband
System.out.println(Armband.getNaam());         // Armband
System.out.println(((Product) armbandje).getNaam()); // Product
System.out.println(Product.getNaam());         // Product
```

	Non-static	Static
Variabele	Type van de referentie	Type van de referentie
Methode	Type van het object	Type van de referentie

Overloading

We hebben al gezien dat Java automatisch primitieve datatypen upcast als je een methode hebt welke een double als argument nodig heeft en je een integer meegeeft als argument.

De vraag is nu: Wat doet Java bij objecten?

```
class Fruit {}  
class Appel extends Fruit {}
```

Overloading

```
public class College {  
    public static void main(String[] args) {  
        Appel a = new Appel();  
        Fruit f = a;  
        Object o = f;  
        test(a);  
        test(f);  
        test(o);  
    }  
    static void test(Object o) {  
        System.out.println("Object!");  
    }  
    static void test(Fruit f) {  
        System.out.println("Fruit!");  
    }  
    static void test(Appel a) {  
        System.out.println("Appel!");  
    }  
}
```

Wat zal de code hiernaast printen?

De code print:

Appel!

Fruit!

Object!

In dit geval zal Java de methode aanroepen welke het beste past bij de referentie, niet het daadwerkelijke type van het geïntanceerde object.

Super

De **super**-methode kunnen we dus gebruiken om in een constructor de constructor van de superklasse aan te roepen.

Daarnaast kunnen we **super** ook gebruiken om bij verborgen instantiatievariabelen te komen en overriden methoden.

Super (I)

Even ophalen, wat was de uitvoer van de onderstaande code?

```
class A {  
    String naam() { return "A"; }  
}  
  
class B extends A {  
    String naam() { return "B"; }  
}  
  
B b = new B();  
A a = b;  
System.out.println(a.naam());  
System.out.println(b.naam());
```

Super (I)

En nu?

```
class A {  
    String naam() { return "A"; }  
}  
  
class B extends A {  
    String naam() { return super.naam() + "B"; }  
}  
  
B b = new B();  
A a = b;  
System.out.println(a.naam());  
System.out.println(b.naam());
```

Verborgen variabelen benaderen met super

```
class A {  
    int a = 3;  
}  
  
class B extends A {  
    int a = 4;  
    int som() { return super.a + a; }  
}  
  
System.out.println(b.som());  
// Output: 7
```

Interfaces

Voor modulaire software is het vaak fijn bepaalde garanties over een object te hebben. Bijvoorbeeld: “Object A heeft een click-methode die aangeroepen kan worden zodra iemand klikt op [..].”

Een garantie dat een gegeven object of klasse bepaalde methoden en constanten bevat, kan in veel programmeertalen worden gegeven met behulp van een **interface**.

Een interface is een soort van **contract** waar een klasse zich aan moet houden en bevat een lijst van methoden en constanten die een klasse **moet** implementeren.

Interfaces

Alle methodes van een interface zijn altijd (impliciet) **public**. Hiernaast zijn variabelen in een interface altijd (impliciet) **public**, **static** en **final** (Hoe noemen we zulke variabelen ook al weer?)

De code produceert een foutmelding wanneer je bijvoorbeeld een private variabele in een interface zet.

In de onderstaande code zie je hoe Java onder water de code aanpast:

```
interface Hardloper {  
    int snelheid();  
    double afstand = 70;  
}
```

```
interface Hardloper {  
    public abstract int snelheid();  
    public static final double afstand = 70;  
}
```

Interfaces

Een ander voorbeeld is de onderstaande HondInterface

```
interface HondInterface {  
    public Hond blaf(String geluid);  
}
```

Een klasse kan vervolgens zeggen zich aan deze interface (afpraak) te houden middels het **implements** keyword:

```
class Hond implements HondInterface  
{  
    public Hond blaf(String geluid)  
    {  
        System.out.println(geluid);  
        return this;  
    }  
}
```

Standaard modifiers van een interface

De onderstaande code geeft een compileerfout: waarom?

```
interface HondInterface {  
    Hond blaf(String geluid);  
}  
  
class Hond implements HondInterface  
{  
    Hond blaf(String geluid) {  
        System.out.println(geluid);  
        return this;  
    }  
}
```

Omdat de implementatie van **blaf** niet public is, maar wel (impliciet) in de interface

Het nalaten een methode te implementeren

Als je niet alle methoden van een interface implementeert geeft de compiler een foutmelding:

```
error: Hond is not abstracted and does not override abstracted method  
blaf(String) in HondInterface  
class Hond implements HondInterface {  
  ^  
1 error
```

Constanten

Verder kan een interface ook constanten bevatten:

```
interface KostGeldInterface {
    int BTW_PERCENTAGE = 21;
    int getPrijsInclusiefBtw();
}

class LegoKasteel implements KostGeldInterface {
    int prijsExclusiefBtw = 10000;
    public int getPrijsInclusiefBtw() {
        return (int) prijsExclusiefBtw * (1.0 + BTW_PERCENTAGE / 100.0);
    }
}
```

Abstracte klassen

Tot nu toe hebben we twee principes behandeld:

- Overerving: een klasse erft alle methoden en variabelen van een andere klasse
- Interfaces: Een contract dat definieert welke methode een klasse moet bevatten

Een interface is **niet** direct instantieerbaar en kan ook **geen** implementaties bevatten van methoden.

Daarom is er een tussenweg: Een **abstracte klasse**

Abstracte klassen

Abstracte klassen hebben de volgende eigenschappen:

- Ze zijn niet instantieerbaar (~~Appel a = new Appel();~~)
- Bevatten zowel methoden **met** een implementatie,
- Als methoden **zonder** een implementatie: Abstracte methoden

In tegenstelling tot interfaces kan een abstracte klasse wel instantiatievariabelen bevatten.

Definiëren van een abstracte klassen

```
abstract class Vorm {  
    protected int x, y;  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    abstract int getOppervlakte();  
}
```

De abstracte vorm klasse

Deze abstracte klasse heeft dus instantievariabelen, getters en een abstracte methode.

Echter: een abstracte klasse heeft een unieke eigenschap:

```
Vorm vorm = new Vorm()  
System.out.println(vorm);
```

```
error: Vorm is abstract; cannot be instantiated  
    Vorm vorm = new Vorm();
```

Het compileert niet. Abstracte klassen kun je niet instantiëren.

Implementeren van een abstracte klasse

Vervolgens kun je net zoals een reguliere klasse, een abstracte klasse uitbreiden. Als je alle abstracte methoden override, kun je hem dan als normale klasse instantiëren!

```
class Vierkant extends Vorm {  
    int h, w;  
  
    Vierkant(int x, int y, int w, int h) {  
        this.x = x; this.y = y; this.w = w; this.h = h;  
    }  
  
    int getOppervlakte() {  
        return w * h;  
    }  
}
```

Wanneer een kies je wat?

Abstracte klasse:

- Als je code wilt delen tussen verschillende (gerelateerde) klassen
- Als je non-static of non-final variabelen wilt gebruiken
- Als de **sub** en **super**klasse een “is”-relatie hebben.

Interface:

- Als de klasse en de interface niet per se volledig gerelateerd zijn
- Als je garanties wilt hebben over het bestaan van methoden, maar verder niet bent geïnteresseerd in het gedrag van een klasse
- Als je gebruik moet maken van multiple inheritance

Wanneer ben je verplicht een klasse abstract te maken?

Als jouw klasse niet alle abstracte methoden van een abstracte superklasse of interface implementeert, **moet** jouw klasse ook *abstract* worden.

Casting

Casting is het proces waarbij een variabele van een oorspronkelijk type **A** gevraagd wordt om zich als een ander type **B** te gedragen.

De cast kan naar zijn eigen klassetype zijn of naar een van zijn subklasse- of superklassetypes of interfaces.

Er gelden echter regels voor compileren en runtime-regels voor casten in Java.

Casting

Het casten van objectreferenties hangt af van de **relatie** van de klassen die bij dezelfde hiërarchie betrokken zijn.

Elke objectreferentie kan worden toegewezen aan een referentievariabele van het type Object, omdat de Object-klasse een **superklasse** is van elke Java-klasse.

Er kunnen zich twee casting scenario's voor doen in Java:

1. Upcasting
2. Downcasting

Casting

downcast:

Wanneer we een verwijzing langs de klassenhiërarchie casten in een richting van de hoofdklasse naar de onderliggende of subklassen casten
(van boven naar beneden)

Upcast:

Wanneer we een verwijzing langs de klassenhiërarchie casten in een richting van de subklassen naar de hoofdklasse

We hoeven in het geval van een upcast geen cast-operator te gebruiken.

Casting

De regels van het die bij het casten horen controleren of een object of variabele dat van een bepaald type naar een ander type gecast wordt wel valide is.

Zo kan je niet zomaar van het ene object het andere object maken als deze geen hiërarchische relatie hebben.

Impliciete casting

Over het algemeen wordt een impliciete cast gedaan wanneer een objectreferentie wordt toegewezen (cast) aan:

- Een referentievariabele waarvan het type hetzelfde is als de klasse waaruit het object is geïntanceerd. Een object als object is een superklasse van elke klasse.
- Een referentievariabele waarvan het type een superklasse is van de klasse waaruit het object is geïntanceerd.
- Een referentievariabele waarvan het type een interface is die wordt geïmplementeerd door de klasse van waaruit het object is geïntanceerd.
- Een referentievariabele waarvan het type een interface is die wordt geïmplementeerd door een superklasse van de klasse waaruit het object is geïntanceerd.

Impliciete casting

```
interface Voertuig {  
}  
class Auto implements Voertuig {  
}  
class Ford extends Auto {  
}  
  
a = new Auto();  
f = new Ford();  
a = f;
```

Polymorfisme

De letterlijke betekenis van polymorfisme is “vele vormen” en het komt voor wanneer we veel klassen hebben die door overerving aan elkaar verwant zijn.

Overerving laat ons attributen en methoden erven van een andere klasse.

Polymorfisme gebruikt die methoden om verschillende taken uit te voeren.

Hierdoor kunnen we één handeling op verschillende manieren uitvoeren.

Polymorfisme

```
class Dier {  
    public void diergeluid() {  
        System.out.println("Het dier maakt een geluid");  
    }  
}  
  
class Varken extends Dier {  
    public void diergeluid() {  
        System.out.println("Het varken zegt: 'wee wee'");  
    }  
}  
  
class Hond extends Dier {  
    public void diergeluid() {  
        System.out.println("De hond zegt: 'waf waf'");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Dier mijnDier = new Dier();  
        Dier MijnVarken = new Varken();  
        Dier mijnHond = new Hond();  
        mijnDier.diergeluid();    // het dier maakt een  
geluid  
        MijnVarken.diergeluid(); // het varken zegt: 'wee  
wee'  
        mijnHond.diergeluid();    // de hond zegt: 'waf waf'  
    }  
}
```

Simpele lambda expressies

Laten we het ten slotte nog kort de lambda expressies behandelen.

Lambda expressies vallen onder het **functioneel programmeren** in Java. Functioneel programmeren maakt het mogelijk om **declaratieve code** te schrijven.

Declaratieve code laat je definiëren *wat* je moet doen, in plaats van je te concentreren op *hoe* je het moet doen.

Simpele lambda expressies

Een lambda-expressie is een kort codeblok dat parameters opneemt en een waarde retourneert.

Lambda-expressies zijn vergelijkbaar met methoden, maar ze hebben geen naam nodig en ze kunnen rechtstreeks in de hoofdtekst van een methode worden geïmplementeerd.

Simpele lambda expressies

De eenvoudigste lambda-expressie bevat een enkele parameter en een expressie:

```
parameter -> expressie
```

Als je meer dan één parameter wilt gebruiken stop je er haakjes omheen:

```
(parameter1, parameter2) -> expressie
```

Simpele lambda expressies

Uitdrukkingen zijn beperkt. Ze moeten **onmiddellijk** een waarde retourneren en mogen geen variabelen, toewijzingen of instructies bevatten zoals *if* of *for*.

Om complexere bewerkingen uit te voeren, kan een codeblok worden gebruikt met accolades. Als de lambda-expressie een waarde moet retourneren, moet het codeblok een return-instructie hebben.

```
(parameter1, parameter2) -> { code blok }
```

Simpele lambda expressies: voorbeeld

Verwarrend? Laten we eens kijken naar een simpel voorbeeld:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> nummers = new ArrayList<Integer>();
        nummers.add(5);
        nummers.add(9);
        nummers.add(8);
        nummers.add(1);
        nummers.forEach( (n) -> { System.out.println(n); } );
    }
}
```

We hebben eigenlijk een kleine functie gebouwd die elk getal uit de ArrayList ophaald.

Hiervoor is geen hele functie nodig, want dat kost meer regels code en maakt dus onze code onoverzichtelijker.

Eindopdracht

Implementeer de methodes die beschreven zijn in de Queue interface.

```
interface Queue {
    // voeg een item toe aan de FIFO queue
    public void add(int value);
    // verwijder een item uit de FIFO queue
    public int remove();
    // geef het eerste item in de FIFO queue terug, maar haal het er niet uit
    public int peek();
    // geef aan of de FIFO queue leeg is
    public boolean isEmpty();
    // geef de lengte van de FIFO queue terug
    public int size();
    // Print de inhoud van de FIFO queue
    public void print();
    // verwijder alle items uit de FIFO queue
    public void clear();
    // verwijder de eerste n items uit de FIFO queue
    public void clear(int n)
    // print de inhoud van de FIFO queue in omgekeerde volgorde
    public void printReverse();
    // plaats een element op een bepaalde positie in de FIFO queue
    public void jumpTheQueue(int n, int value);
    // Zet de FIFO queue om naar een String
    public String toString();
    // Kijk of de FIFO queue gelijk is aan een andere FIFO queue
    public boolean equals(Queue q);
    // Bepaal de index van een bepaalde waarde in de FIFO queue
    public int indexOf(int value);
    // bepaal de laatste index van een bepaalde waarde in de FIFO queue
    public int lastIndexOf(int value);
}
```

Leerdoelen

- ✓ Het begrijpen en het implementeren van overerving (inheritance)
- ✓ Het ontwikkelen van code waarin polymorfisme in wordt toegepast
- ✓ Het verschil weten tussen een referentie en een object
- ✓ Bepalen of er gebruik moet worden gemaakt van casting
- ✓ `super` en `this` operatoren kunnen gebruiken
- ✓ Gebruik maken van `abstract` klassen en `interfaces`
- ✓ Schrijf een eenvoudige Lambda-uitdrukking die een Lambda-predikaatuitdrukking verbruikt

Vragen?

- E-mail mij op joris.vanbreugel@code-cafe.nl!
- Join de Code-Café community op discord!

