## methode

# Klasse en objecten

### Voorbeeld

### Te gebruiken bij:

-

### Eigenschappen:

-

### Info

#### Autoboxing

Net als de CreditCard-class een constructor heeft, heeft een object-variant van een primitieve (zoals de Integer) ook een constructor. Die zorgt er namelijk voor dat je er een waarde mee kunt aanmaken:

Integer x = new Integer(7);

Hoewel deze manier van aanmaken werkt, hoef je deze niet te gebruiken. Sterker nog: deze schrijfwijze wordt zelfs afgeraden. Dat komt omdat deze object typen zó vaak gebruikt worden dat het een standaard onderdeel van Java is. Deze standaard integratie noemen we **autoboxing** en zorgt ervoor dat het converteren van een primitieve naar een object automatisch gebeurt:

Integer x = 7;

In bovenstaand voorbeeld kan de primitieve waarde 7 (int) direct worden toegekend aan de variabele van type Integer, omdat Java de conversie voor ons regelt. Dit werkt ook in omgekeerde volgorde! Dit heet het **auto-unboxing** principe, waardoor het werken met de object types net zo makkelijk is als het werken met primitieven:

Integer x = 7;

Integer y = 13;

// de waardes in Integer objecten x en y worden eerst automatisch primitieven gemaakt, daarna opgeteld en daarna wordt van het resultaat een nieuwe Integer gemaakt:

Integer z = x + y;

// Als we deze speciale syntax niet zouden hebben, zouden we het elke keer zelf moeten doen... zoals hier:

Integer x = new Integer(7);

Integer y = new Integer(13);

// gelukkig hoeven we niet deze bijna-onleesbare-variant te gebruiken!

Integer z = new Integer(x.intValue() + y.intValue());

#### overzicht

// Superklasse (hoofdklasse)

public class Superklasse {

// attribuut1 en attribuut2 behoren tot de Superklasse

private String attribuut1; // behoort tot Superklasse

private int attribuut2; // behoort tot Superklasse

// Constructor van de Superklasse

public Superklasse(String attribuut1, int attribuut2) {

this.attribuut1 = attribuut1;

this.attribuut2 = attribuut2;

}

// Getter en Setter voor attribuut1

public String getAttribuut1() {

return attribuut1;

}

public void setAttribuut1(String attribuut1) {

this.attribuut1 = attribuut1;

}

// Getter en Setter voor attribuut2

public int getAttribuut2() {

return attribuut2;

}

public void setAttribuut2(int attribuut2) {

this.attribuut2 = attribuut2;

}

}

// Subklasse 1 (afgeleide klasse van Superklasse)

public class Subklasse1 extends Superklasse {

// attribuut3 behoort tot Subklasse1

private boolean attribuut3; // behoort tot Subklasse1

// Constructor van Subklasse1

public Subklasse1(String attribuut1, int attribuut2, boolean attribuut3) {

// Roep de constructor van de Superklasse aan

super(attribuut1, attribuut2);

this.attribuut3 = attribuut3;

}

// Getter en Setter voor attribuut3

public boolean isAttribuut3() {

return attribuut3;

}

public void setAttribuut3(boolean attribuut3) {

this.attribuut3 = attribuut3;

}

}

// Subklasse 2 (afgeleide klasse van Superklasse)

public class Subklasse2 extends Superklasse {

// attribuut3 behoort tot Subklasse2

private double attribuut3; // behoort tot Subklasse2

// Constructor van Subklasse2

public Subklasse2(String attribuut1, int attribuut2, double attribuut3) {

// Roep de constructor van de Superklasse aan

super(attribuut1, attribuut2);

this.attribuut3 = attribuut3;

}

// Getter en Setter voor attribuut3

public double getAttribuut3() {

return attribuut3;

}

public void setAttribuut3(double attribuut3) {

this.attribuut3 = attribuut3;

}

}

// Voorbeeldgebruik

public class Main {

public static void main(String[] args) {

// Instantie van Subklasse1

Subklasse1 subklasse1 = new Subklasse1("waarde1", 2, true); // Instantie van Subklasse1 (object van Subklasse1)

// Instantie van Subklasse2

Subklasse2 subklasse2 = new Subkl

#### Stappen

De code begint met het definiëren van de Superklasse. Dit is de hoofdklasse waarvan andere klassen erven.

De Superklasse bevat twee attributen: attribuut1 (van het type String) en attribuut2 (van het type int). Deze attributen zijn privé en kunnen alleen worden benaderd via getters en setters.

De Superklasse heeft een constructor die attribuut1 en attribuut2 als parameters accepteert en de waarden ervan toewijst aan de overeenkomstige attributen.

Daarna volgen de getters en setters voor attribuut1 en attribuut2, waarmee de waarden kunnen worden opgehaald en gewijzigd.

Vervolgens worden de subklassen gedefinieerd: Subklasse1 en Subklasse2. Deze klassen erven van de Superklasse.

Subklasse1 voegt een extra attribuut attribuut3 (van het type boolean) toe.

Subklasse1 heeft een constructor die attribuut1, attribuut2 en attribuut3 als parameters accepteert. Deze constructor roept de constructor van de Superklasse aan met behulp van super(attribuut1, attribuut2) om de waarden van attribuut1 en attribuut2 in te stellen.

Subklasse1 bevat ook getters en setters voor attribuut3.

Op een vergelijkbare manier voegt Subklasse2 een extra attribuut attribuut3 (van het type double) toe en heeft het een constructor en getters/setters voor attribuut3.

In de Main-klasse worden objecten (subklasse1 en subklasse2) van zowel Subklasse1 als Subklasse2 geïnstantieerd.

De waarden van attributen worden opgehaald en gewijzigd met behulp van de getters en setters.

#### Zonder superklasse

public class MyClass {

// Variabelen

private int myVariable;

private String myString;

// Constructor

public MyClass(int variable, String string) {

myVariable = variable;

myString = string;

}

// Methoden

public void myMethod() {

System.out.println("Dit is een voorbeeldmethode.");

}

public static void main(String[] args) {

// Instantie van MyClass maken

MyClass myObject = new MyClass(10, "Hello");

// Toegang tot variabelen en methoden

int variableValue = myObject.myVariable;

String stringValue = myObject.myString;

myObject.myMethod();

}

}

In dit voorbeeld hebben we een klasse genaamd MyClass met enkele privévariabelen, een constructor en een methode. In de main-methode maken we een instantie van MyClass genaamd myObject met behulp van de constructor. Vervolgens hebben we toegang tot de variabelen en de methode van de instantie.

Opmerking: De privévariabelen myVariable en myString kunnen alleen binnen de klasse MyClass worden gebruikt. Als je toegang tot deze variabelen wilt bieden buiten de klasse, kun je ze als public of protected declareren, afhankelijk van de vereisten van je programma.

#### Refactoring

Refactoring in Java verwijst naar het proces van het herstructureren van bestaande code zonder de externe functionaliteit te wijzigen. Het is een techniek die wordt gebruikt om de leesbaarheid, onderhoudbaarheid en efficiëntie van de code te verbeteren. Het doel van refactoring is om de code te vereenvoudigen, duplicatie te verminderen, betere benamingen te geven aan variabelen en methoden, en algemene verbeteringen aan te brengen zonder de werking van de code te veranderen.

Hier zijn enkele veelvoorkomende technieken die worden gebruikt bij het refactoren van Java-code:

##### Extract Method:

Deze techniek wordt gebruikt om een deel van de code uit een bestaande methode te isoleren en in een aparte methode te plaatsen. Dit verbetert de leesbaarheid en modulariteit van de code.

java

// Oorspronkelijke code

public void printCustomerDetails(Customer customer) {

System.out.println("Name: " + customer.getName());

System.out.println("Age: " + customer.getAge());

// ...

}

// Na refactoring

public void printCustomerDetails(Customer customer) {

printName(customer);

printAge(customer);

// ...

}

private void printName(Customer customer) {

System.out.println("Name: " + customer.getName());

}

private void printAge(Customer customer) {

System.out.println("Age: " + customer.getAge());

}

##### Rename Variable:

Dit houdt in dat je variabelen hernoemt om hun betekenis en doel duidelijker weer te geven. Goede namen maken de code begrijpelijker en verminderen de behoefte aan commentaar.

java

// Oorspronkelijke code

int x = 5;

int y = 10;

int z = x + y;

// Na refactoring

int firstNumber = 5;

int secondNumber = 10;

int sum = firstNumber + secondNumber;

##### Eliminate Duplicate Code:

Hierbij zoek je naar duplicatie van code en consolideer je deze in een gemeenschappelijke methode of klasse. Dit verbetert de onderhoudsvriendelijkheid en vermindert de kans op fouten.

java

// Oorspronkelijke code

public void printProductDetails(Product product) {

System.out.println("Name: " + product.getName());

System.out.println("Price: " + product.getPrice());

// ...

}

public void printOrderDetails(Order order) {

System.out.println("Product Name: " + order.getProduct().getName());

System.out.println("Product Price: " + order.getProduct().getPrice());

// ...

}

// Na refactoring

public void printDetails(String name, double price) {

System.out.println("Name: " + name);

System.out.println("Price: " + price);

// ...

}

public void printProductDetails(Product product) {

printDetails(product.getName(), product.getPrice());

}

public void printOrderDetails(Order order) {

printDetails(order.getProduct().getName(), order.getProduct().getPrice());

}

##### Simplify Conditional Expressions:

Hierbij vereenvoudig je complexe voorwaardelijke expressies door logische operatoren en hulpmethoden te gebruiken. Dit verbetert de leesbaarheid en begrijpelijkheid van de code.

java

// Oorspronkelijke code

if ((x > 0 && x < 10) || (y > 0 && y < 10) || (z > 0 && z < 10)) {

// Doe iets

}

// Na refactoring

if (isInRange(x) || isInRange(y) || isInRange(z)) {

// Doe iets

}

private boolean isInRange(int value) {

return value > 0 && value < 10;

}

#### Classe

##### Variabelen en Datatypen:

java

// Voorbeeld van variabelen en datatypen

int age = 25; // Een variabele van het type int om de leeftijd op te slaan

double salary = 2500.50; // Een variabele van het type double om het salaris op te slaan

boolean isEmployed = true; // Een variabele van het type boolean om de werkstatus op te slaan

String name = "John Doe"; // Een variabele van het type String om de naam op te slaan

In dit voorbeeld worden variabelen van verschillende datatypen gebruikt om gegevens zoals leeftijd, salaris, werkstatus en naam op te slaan.

##### Operatoren:

java

// Voorbeeld van operatoren

int a = 10;

int b = 5;

int sum = a + b; // Opteloperator (+)

int difference = a - b; // Aftrekoperator (-)

int product = a \* b; // Vermenigvuldigingsoperator (\*)

int quotient = a / b; // Delingsoperator (/)

int remainder = a % b; // Modulusoperator (%)

In dit voorbeeld worden verschillende operatoren gebruikt, zoals optellen (+), aftrekken (-), vermenigvuldigen (\*), delen (/) en modulus (%), om berekeningen uit te voeren met twee getallen.

##### Beslissingsstructuren:

java

// Voorbeeld van een if-statement

int age = 18;

if (age >= 18) {

System.out.println("Je bent volwassen.");

} else {

System.out.println("Je bent minderjarig.");

}

In dit voorbeeld wordt een if-statement gebruikt om te controleren of de leeftijd groter of gelijk is aan 18. Als dat het geval is, wordt de boodschap "Je bent volwassen." afgedrukt. Anders, als de leeftijd kleiner is dan 18, wordt de boodschap "Je bent minderjarig." afgedrukt.

##### Lussen:

java

// Voorbeeld van een for-loop

for (int i = 1; i <= 5; i++) {

System.out.println("Getal: " + i);

}

In dit voorbeeld wordt een for-loop gebruikt om de getallen van 1 tot 5 af te drukken. De loop begint bij 1, loopt zolang het getal kleiner of gelijk is aan 5, en verhoogt het getal met 1 bij elke iteratie.

##### Arrays:

java

// Voorbeeld van een array

int[] numbers = {1, 2, 3, 4, 5};

for (int i = 0; i < numbers.length; i++) {

System.out.println("Getal: " + numbers[i]);

}

In dit voorbeeld wordt een array met getallen gemaakt en vervolgens worden de getallen afgedrukt met behulp van een for-loop. De lengte van de array wordt verkregen met behulp van de "length" eigenschap.

##### Encapsulatie:

java

// Voorbeeld van encapsulatie met getter en setter methoden

public class Person {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

In dit voorbeeld wordt encapsulatie toegepast door de attributen "name" en "age" privé te maken en publieke getter- en setter-methoden te bieden om toegang te krijgen tot en de waarden van de attributen in te stellen.

##### Overerving:

java

// Voorbeeld van overerving met een subklasse

public class Vehicle {

protected String brand;

protected String color;

public void start() {

System.out.println("Het voertuig start.");

}

}

public class Car extends Vehicle {

private int numberOfDoors;

public void accelerate() {

System.out.println("De auto versnelt.");

}

}

In dit voorbeeld wordt overerving gebruikt om een subklasse "Car" af te leiden van de superklasse "Vehicle". De subklasse erft de attributen en methoden van de superklasse en heeft ook zijn eigen specifieke attributen en methoden.

##### Abstracte classes:

java

// Voorbeeld van een abstracte klasse met een abstracte methode

public abstract class Shape {

protected String color;

public Shape(String color) {

this.color = color;

}

public abstract double calculateArea();

}

public class Circle extends Shape {

private double radius;

public Circle(String color, double radius) {

super(color);

this.radius = radius;

}

public double calculateArea() {

return Math.PI \* radius \* radius;

}

}

In dit voorbeeld wordt een abstracte klasse "Shape" gemaakt met een abstracte methode "calculateArea()". De subklasse "Circle" erft van de abstracte klasse en implementeert de abstracte methode om de oppervlakte van een cirkel te berekenen.

##### Interfaces:

java

// Voorbeeld van een interface

public interface Drawable {

void draw();

}

public class Circle implements Drawable {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

public void draw() {

System.out.println("Een cirkel tekenen met radius " + radius);

}

}

In dit voorbeeld wordt een interface "Drawable" gemaakt met de methode "draw()". De klasse "Circle" implementeert de interface en implementeert de methode om een cirkel te tekenen.

##### Scope:

java

public class ScopeExample {

// Klassevariabele (beschikbaar in de hele klasse)

static int globalVariable = 10;

public static void main(String[] args) {

// Lokale variabele (beperkt tot de main-methode)

int localVariable = 20;

if (localVariable == 20) {

// Blokvariabele (beperkt tot dit if-blok)

int blockVariable = 30;

System.out.println("Blokvariabele: " + blockVariable);

}

System.out.println("Lokale variabele: " + localVariable);

System.out.println("Klassevariabele: " + globalVariable);

}

public static void anotherMethod() {

// Toegang tot de klassevariabele vanuit een andere methode

System.out.println("Klassevariabele vanuit een andere methode: " + globalVariable);

}

}

In dit voorbeeld worden verschillende niveaus van scope gedemonstreerd. De klasse "ScopeExample" heeft een klassevariabele genaamd "globalVariable", die beschikbaar is in de hele klasse. De "main" methode bevat een lokale variabele genaamd "localVariable", die alleen beschikbaar is binnen de scope van de "main" methode.

Binnen de "if" statement wordt een blokvariabele genaamd "blockVariable" gedefinieerd, die alleen toegankelijk is binnen dat specifieke blok.

De waarden van de variabelen worden afgedrukt met behulp van de "System.out.println" methode. Merk op dat de klassevariabele kan worden benaderd vanuit zowel de "main" methode als een andere methode genaamd "anotherMethod".

##### Hiërarchie:

java

// Voorbeeld van een hiërarchie van klassen

public class Animal {

public void eat() {

System.out.println("Het dier eet.");

}

}

public class Dog extends Animal {

public void bark() {

System.out.println("De hond blaft.");

}

}

public class Cat extends Animal {

public void meow() {

System.out.println("De kat miauwt.");

}

}

In dit voorbeeld wordt een hiërarchie van klassen gemaakt, waarbij de klassen "Dog" en "Cat" afgeleid zijn van de klasse "Animal". Elke subklasse heeft zijn eigen specifieke gedrag naast het gedrag dat wordt geërfd van de superklasse.

##### Generics:

java

// Voorbeeld van generieke klassen en methoden

public class Box<T> {

private T content;

public void add(T item) {

this.content = item;

}

public T get() {

return content;

}

}

public class Main {

public static void main(String[] args) {

Box<String> stringBox = new Box<>();

##### annotation

@Override

Het @Override-annotation wordt gebruikt in Java om aan te geven dat een methode een overriden (overschreven) versie is van een methode in de superclass (ouderklasse) of een interface die wordt geïmplementeerd.

Wanneer je de @Override-annotation voor een methode plaatst, controleert de Java-compiler of de methode daadwerkelijk wordt overschreven vanuit een superclass of geïmplementeerd vanuit een interface. Als er geen overeenkomende methode in de superclass of interface is gevonden, geeft de compiler een foutmelding.

Het gebruik van @Override is optioneel, maar het wordt sterk aanbevolen om het te gebruiken wanneer je een methode wilt overschrijven. Het helpt bij het voorkomen van typfouten en zorgt ervoor dat de methode correct wordt geïmplementeerd.

java

public class ParentClass {

public void printMessage() {

System.out.println("Hello, world!");

}

}

public class ChildClass extends ParentClass {

@Override

public void printMessage() {

System.out.println("Hello from the child class!");

}

}

###### @Override:

Geeft aan dat een methode een overriden versie is van een methode in de superclass of geïmplementeerde interface.

Voorbeeld:

java

@Override

public void someMethod() {

// Implementatie van de overriden methode

}

###### @Deprecated:

Geeft aan dat een element (bijv. methode, klasse, veld) verouderd is en niet langer wordt aanbevolen om te gebruiken.

Voorbeeld:

java

@Deprecated

public void oldMethod() {

// Oude implementatie van de methode

}

@SuppressWarnings:

Onderdrukt compilerwaarschuwingen voor specifieke situaties of types van waarschuwingen.

Voorbeeld:

java

@SuppressWarnings("unchecked")

public List<String> getList() {

// Onderdruk waarschuwingen voor ongecontroleerde conversie

return new ArrayList();

}

###### @FunctionalInterface:

Geeft aan dat een interface een functioneel interface is, wat betekent dat deze slechts één abstracte methode heeft en kan worden gebruikt als de basis voor lambda-expressies of method references.

Voorbeeld:

java

@FunctionalInterface

public interface MyFunctionalInterface {

void doSomething();

}

###### @SafeVarargs:

Wordt gebruikt om aan te geven dat een methode of constructor een variabel aantal argumenten (varargs) accepteert van een generiek type, zonder een waarschuwing voor ongecontroleerde variargs-conversie te genereren.

Voorbeeld:

java

@SafeVarargs

public final <T> List<T> createList(T... elements) {

return Arrays.asList(elements);

}