## Interfaces, generices en abstracte klasse.

### Recap

* has-a (composition)
* is-a (inheritance) = tightly coupled, geen acces controlle, lasting testen
* Uses-a (dependence)

#### Compisition over inheritance

#### abstractie

##### absyracte classes

* bestaat en kan alleen bestaan binnen andere classe
* zorgt ervaar dat implementaties verplaats of weggelaten kunnen worden.

// Abstracte class: Speelgoed

class Speelgoed {

constructor(naam, prijs) {

this.naam = naam;

this.prijs = prijs;

}

// Abstracte methode: spelen

spelen() {

// Deze methode moet worden geïmplementeerd in de subklassen

throw new Error("Deze methode moet worden geïmplementeerd.");

}

}

// Subklasse: Auto

class Auto extends Speelgoed {

constructor(naam, prijs, kleur) {

super(naam, prijs);

this.kleur = kleur;

}

spelen() {

console.log(`De ${this.kleur} auto rijdt vrolijk rond.`);

}

}

// Subklasse: Pop

class Pop extends Speelgoed {

constructor(naam, prijs, kapsel) {

super(naam, prijs);

this.kapsel = kapsel;

}

spelen() {

console.log(`De pop met ${this.kapsel} haar wordt mooi gekamd.`);

}

}

// Gebruik van de classes

const rodeAuto = new Auto("Rode auto", 10, "rood");

rodeAuto.spelen(); // Output: De rode auto rijdt vrolijk rond.

const blondePop = new Pop("Blonde pop", 15, "blond");

blondePop.spelen(); // Output: De pop met blond haar wordt mooi gekamd.

In dit voorbeeld hebben we een abstracte class genaamd "Speelgoed" gemaakt die een gemeenschappelijke constructor en een abstracte methode "spelen()" heeft. We hebben ook twee subklassen, "Auto" en "Pop", die de abstracte class erven en hun eigen specifieke attributen en implementaties van de "spelen()" methode hebben.

We maken vervolgens objecten van de subklassen (rodeAuto en blondePop) en roepen de "spelen()" methode aan. Merk op dat we niet rechtstreeks objecten kunnen maken van de abstracte class, omdat deze alleen dient als een blauwdruk voor de subklassen.

Bij het uitvoeren van de code krijgen we de gewenste uitvoer die het specifieke gedrag van elk speelgoed weergeeft.

Ik hoop dat dit voorbeeld je helpt om abstracte classes beter te begrijpen in de context van Jip en Janneke!

#### Interfaces

* Een abstracte class
* Infeiten een contrat tussen een abstracte classe en implements.
* Hebben methodes, geen classs of variablen.
* Moet methode uit de abstracte classe impolementeren.
  + Tenzij:
  + *Class abstract is*
  + *Methode een default interface is.*
* Interface methode zijn altijd public

Stel je voor dat je een winkel hebt die verschillende soorten muziekinstrumenten verkoopt, zoals gitaren, piano's en fluiten. Elk muziekinstrument heeft verschillende kenmerken, zoals het vermogen om geluid te produceren, de mogelijkheid om noten te spelen, enzovoort.

Om dit te organiseren, kun je standaardinterfaces gebruiken. Een interface is als een contract dat bepaalt welke methoden een klasse moet implementeren zonder de details van de implementatie te specificeren. Het is als een lijst met acties die een object moet kunnen uitvoeren.

In ons geval kunnen we een standaardinterface genaamd "Muziekinstrument" maken. Deze interface zal de methoden definiëren die elk muziekinstrument moet implementeren, zoals "geluidProduceren()" en "notenSpelen()".

Elke klasse die een muziekinstrument vertegenwoordigt, kan de "Muziekinstrument" interface implementeren. Bijvoorbeeld, de "Gitaar" klasse kan de methoden "geluidProduceren()" en "notenSpelen()" implementeren op basis van hoe een gitaar geluid produceert en noten speelt. Op dezelfde manier kan de "Piano" klasse deze methoden implementeren volgens de eigenschappen van een piano.

Hier is een voorbeeld van hoe dit er in code uit zou kunnen zien:

// Standaardinterface: Muziekinstrument

interface Muziekinstrument {

void geluidProduceren();

void notenSpelen();

}

// Klasse: Gitaar

class Gitaar implements Muziekinstrument {

public void geluidProduceren() {

System.out.println("De gitaar maakt een mooi geluid.");

}

public void notenSpelen() {

System.out.println("Je speelt melodieën op de gitaar.");

}

}

// Klasse: Piano

class Piano implements Muziekinstrument {

public void geluidProduceren() {

System.out.println("De piano produceert een helder geluid.");

}

public void notenSpelen() {

System.out.println("Je speelt akkoorden en melodieën op de piano.");

}

}

// Gebruik van de classes

Muziekinstrument elektrischeGitaar = new Gitaar();

elektrischeGitaar.geluidProduceren(); // Output: De gitaar maakt een mooi geluid.

elektrischeGitaar.notenSpelen(); // Output: Je speelt melodieën op de gitaar.

Muziekinstrument concertPiano = new Piano();

concertPiano.geluidProduceren(); // Output: De piano produceert een helder geluid.

concertPiano.notenSpelen(); // Output: Je speelt akkoorden en melodieën op de piano.

In Java wordt de standaardinterface gedefinieerd met het interface-sleutelwoord en de methoden die het bevat. De klassen Gitaar en Piano implementeren vervolgens de interface door de interface-methoden te implementeren met het public-sleutelwoord.

Om objecten te maken, gebruiken we de interface als het datatype, omdat we geen objecten rechtstreeks kunnen maken van een interface. We maken een object van de klasse en koppelen het aan de interface. Vervolgens kunnen we de methoden van de interface oproepen op dat object.

Bij het uitvoeren van de code krijgen we dezelfde uitvoer als in het vorige voorbeeld, waarbij elk muziekinstrument zijn specifieke geluidsproductie en notenweergave heeft.In dit voorbeeld hebben we de standaardinterface "Muziekinstrument" gemaakt, die de methoden "geluidProduceren()" en "notenSpelen()" bevat. De "Gitaar" en "Piano" klassen implementeren deze interface door deze methoden te implementeren.

We maken vervolgens objecten van de klassen (elektrischeGitaar en concertPiano) en roepen de methoden van de interface op. Hierbij krijgen we de gewenste uit

##### Default methode

* + Kun in een interface zitten.
  + Worden geïmplementeerd op de interface
  + Kunnenworden overriden (@override) in setter
  + Class bestaat nog niet
  + Afbeelding met tekst, schermopname, software, Besturingssysteem

    Automatisch gegenereerde beschrijving

java

// Standaardinterface: Muziekinstrument

interface Muziekinstrument {

void geluidProduceren();

void notenSpelen();

default void stemmen() {

System.out.println("Het muziekinstrument wordt gestemd.");

}

}

// Klasse: Gitaar

class Gitaar implements Muziekinstrument {

public void geluidProduceren() {

System.out.println("De gitaar maakt een mooi geluid.");

}

public void notenSpelen() {

System.out.println("Je speelt melodieën op de gitaar.");

}

}

// Klasse: Piano

class Piano implements Muziekinstrument {

public void geluidProduceren() {

System.out.println("De piano produceert een helder geluid.");

}

public void notenSpelen() {

System.out.println("Je speelt akkoorden en melodieën op de piano.");

}

}

// Gebruik van de classes

Muziekinstrument elektrischeGitaar = new Gitaar();

elektrischeGitaar.geluidProduceren(); // Output: De gitaar maakt een mooi geluid.

elektrischeGitaar.notenSpelen(); // Output: Je speelt melodieën op de gitaar.

elektrischeGitaar.stemmen(); // Output: Het muziekinstrument wordt gestemd.

Muziekinstrument concertPiano = new Piano();

concertPiano.geluidProduceren(); // Output: De piano produceert een helder geluid.

concertPiano.notenSpelen(); // Output: Je speelt akkoorden en melodieën op de piano.

concertPiano.stemmen(); // Output: Het muziekinstrument wordt gestemd.

In dit bijgewerkte voorbeeld is er een nieuwe standaardmethode (stemmen()) toegevoegd aan de Muziekinstrument interface. Deze methode heeft een standaardimplementatie in de interface zelf, waardoor alle klassen die de interface implementeren de methode automatisch erven.

Zowel de Gitaar als de Piano klassen implementeren nu de stemmen() methode vanwege de standaardimplementatie in de interface. We kunnen deze methode nu rechtstreeks aanroepen op objecten van deze klassen, naast de andere interface-methoden.

Bij het uitvoeren van de code krijgen we de verwachte uitvoer waarbij de nieuwe stemmen() methode wordt opgeroepen voor zowel de gitaar als de piano.

##### Functionele interfaces

* Type of interface
* Precies 1 methode
* Optie 1
* Afbeelding met tekst, elektronica, schermopname, software

  Automatisch gegenereerde beschrijving
* Optie 2 Lambda (implements van een interface)
  + Afbeelding met tekst, schermopname, Lettertype, ontwerp

    Automatisch gegenereerde beschrijving
* Afbeelding met tekst, schermopname, software, Website

  Automatisch gegenereerde beschrijving
* Standard functionele interfaces

// Functionele interface: Muziekinstrument

@FunctionalInterface

interface Muziekinstrument {

void geluidProduceren();

default void stemmen() {

System.out.println("Het muziekinstrument wordt gestemd.");

}

}

// Klasse: Gitaar

class Gitaar implements Muziekinstrument {

public void geluidProduceren() {

System.out.println("De gitaar maakt een mooi geluid.");

}

}

// Gebruik van de functionele interface

Muziekinstrument elektrischeGitaar = () -> {

System.out.println("De gitaar maakt een mooi geluid.");

};

elektrischeGitaar.geluidProduceren(); // Output: De gitaar maakt een mooi geluid.

elektrischeGitaar.stemmen(); // Output: Het muziekinstrument wordt gestemd.

In dit bijgewerkte voorbeeld hebben we de interface Muziekinstrument aangeduid als een functionele interface met de annotatie @FunctionalInterface. Een functionele interface is een interface met slechts één abstracte methode. Hier hebben we de geluidProduceren() methode als de enige abstracte methode in de interface.

In plaats van een klasse te implementeren, maken we nu een object van de functionele interface met behulp van een lambda-uitdrukking. De lambda-uitdrukking definieert de implementatie van de enkele abstracte methode van de interface. In dit geval roepen we de System.out.println() aan om het geluid van de gitaar af te drukken.

We maken een object van het functionele interface-type en roepen de geluidProduceren() en stemmen() methoden op. Merk op dat de standaardmethode stemmen() ook beschikbaar is via het functionele interface-object.

Bij het uitvoeren van de code krijgen we de verwachte uitvoer waarbij het geluid van de gitaar wordt afgedrukt en de stemmen() methode wordt opgeroepen.

Dit is een eenvoudig voorbeeld van het gebruik van functionele interfaces in Java voor het bereiken van gedrag met lambda-uitdrukkingen. Functionele interfaces bieden een handige manier om anonieme functies te maken in Java.

* Afbeelding met tekst, schermopname, Lettertype, nummer

  Automatisch gegenereerde beschrijving
  + Consumer 1 argument void return
  + Supplier no arguments 1 return
  + Predicate 1 argument Boolean return
  + Function 1 argument 1 return

Consumer<T>:

java

Copy code

import java.util.Arrays;

import java.util.List;

public class ConsumerExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Consumer om elk element af te drukken

Consumer<String> printer = (name) -> System.out.println(name);

names.forEach(printer);

}

}

Uitleg:

In dit voorbeeld maken we gebruik van de Consumer<String> interface om elk element van een lijst af te drukken.

De Consumer<String> interface accepteert een argument van het type String en heeft geen retourwaarde.

We definiëren een Consumer genaamd printer met een lambda-expressie (name) -> System.out.println(name), die de gegeven naam afdrukt naar de console.

Vervolgens gebruiken we de forEach-methode op de names-lijst om de printer-consumer toe te passen op elk element van de lijst.

Toepassing:

Dit kan handig zijn wanneer je een bewerking wilt uitvoeren op elk element van een lijst, zoals het afdrukken van de elementen, het opslaan in een database, enz.

Supplier<T>:

java

Copy code

import java.util.Random;

import java.util.function.Supplier;

public class SupplierExample {

public static void main(String[] args) {

// Supplier voor het genereren van een willekeurig getal

Supplier<Integer> randomNumberSupplier = () -> new Random().nextInt();

// Gebruik van de Supplier om een willekeurig getal te verkrijgen

int randomNumber = randomNumberSupplier.get();

System.out.println("Willekeurig getal: " + randomNumber);

}

}

Uitleg:

In dit voorbeeld maken we gebruik van de Supplier<Integer> interface om een willekeurig getal te genereren.

De Supplier<Integer> interface heeft geen argumenten en retourneert een waarde van het type Integer.

We definiëren een Supplier genaamd randomNumberSupplier met een lambda-expressie () -> new Random().nextInt(), die een willekeurig getal genereert met behulp van de nextInt()-methode van de Random-klasse.

Vervolgens roepen we de get()-methode aan op de randomNumberSupplier om het willekeurige getal te verkrijgen en het naar de console af te drukken.

Toepassing:

Dit kan handig zijn wanneer je een waarde nodig hebt, zoals een willekeurig gegenereerd getal, om te gebruiken in je code of om door te geven aan andere delen van het programma.

Predicate<T>:

java

Copy code

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Predicate om te controleren of een getal even is

Predicate<Integer> isEvenPredicate = (num) -> num % 2 == 0;

// Filteren van de even getallen met behulp van het Predicate

List<Integer> evenNumbers = filterNumbers(numbers, isEvenPredicate);

System.out.println("Even getallen: " + evenNumbers);

}

public static List<Integer> filterNumbers(List<Integer> numbers, Predicate<Integer> predicate) {

List<Integer> filteredNumbers = new ArrayList<>();

for (Integer number : numbers) {

if (predicate.test(number)) {

filteredNumbers.add(number);

}

}

return filteredNumbers;

}

}

Uitleg:

In dit voorbeeld maken we gebruik van de Predicate<Integer> interface om even getallen te filteren uit een lijst van getallen.

De Predicate<Integer> interface accepteert een argument van het type Integer en retourneert een boolean-waarde.

We definiëren een Predicate genaamd isEvenPredicate met een lambda-expressie (num) -> num % 2 == 0, die controleert of het gegeven getal even is.

We hebben ook een hulpmethode filterNumbers gemaakt die een lijst van getallen en een Predicate accepteert, en de getallen filtert op basis van het gegeven predicaat.

We roepen de filterNumbers-methode aan met de numbers-lijst en het isEvenPredicate, en slaan de resulterende even getallen op in de evenNumbers-lijst.

Tot slot drukken we de evenNumbers-lijst af naar de console.

Toepassing:

Dit kan handig zijn wanneer je een bepaalde voorwaarde wilt controleren op elementen van een collectie, zoals het filteren van even getallen, het controleren op geldigheid van gegevens, enz.

Function<T, R>:

java

Copy code

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) {

// Function voor het converteren van een getal naar een String

Function<Integer, String> toStringFunction = (num) -> String.valueOf(num);

// Gebruik van de Function om een getal naar een String te converteren

String numberString = toStringFunction.apply(123);

System.out.println("Getal als String: " + numberString);

}

}

Uitleg:

In dit voorbeeld maken we gebruik van de Function<Integer, String> interface om een getal naar een String te converteren.

De Function<Integer, String> interface accepteert een argument van het type Integer en retourneert een waarde van het type String.

We definiëren een Function genaamd toStringFunction met een lambda-expressie (num) -> String.valueOf(num), die de String.valueOf()-methode gebruikt om het gegeven getal te converteren naar een String.

Vervolgens roepen we de apply()-methode aan op de toStringFunction en passen we het getal 123 toe, en slaan we het resultaat op in de numberString.

Tot slot drukken we de numberString af naar de console.

Toepassing:

Dit kan handig zijn wanneer je een bepaalde transformatie wilt toepassen op een waarde, zoals het converteren van een datatype naar een ander datatype, het mappen van gegevens, enz.

* generics(bijvoorbeeld een listarray)
  + type of interface
  + exacte type pas bij instantiesering
  + generieke methode of class

Consumer<T> met Generics:

javaCopy code

import java.util.Arrays;   
import java.util.List;   
import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Consumer om elk element af te drukken Consumer<String> printer = (name) -> System.out.println(name); names.forEach(printer);

List<Integer> numbers = Arrays.asList(1, 2, 3);   
// Consumer om elk element te verdubbelen

Consumer<Integer> doubler = (num) -> System.out.println(num \* 2); numbers.forEach(doubler); } }

In dit voorbeeld hebben we de generieke parameter T van de Consumer interface gebruikt om verschillende typen te ondersteunen. We hebben eerst een lijst van String-elementen en een Consumer<String> gemaakt om de namen af te drukken. Vervolgens hebben we een lijst van Integer-elementen en een Consumer<Integer> gemaakt om elk element te verdubbelen en af te drukken.

Supplier<T> met Generics:

import java.util.function.Supplier;

public class SupplierExample {

public static void main(String[] args) { // Supplier voor het genereren van een willekeurig getal

Supplier<Integer> randomNumberSupplier = () -> (int) (Math.random() \* 100); // Gebruik van de Supplier om een willekeurig getal te verkrijgen

int randomNumber = randomNumberSupplier.get();

System.out.println("Willekeurig getal: " + randomNumber); // Supplier voor het genereren van een willekeurige tekst Supplier<String>

randomTextSupplier = () -> "Random Text"; // Gebruik van de Supplier om willekeurige tekst te verkrijgen

String randomText = randomTextSupplier.get(); System.out.println("Willekeurige tekst: " + randomText); } }

In dit voorbeeld hebben we de generieke parameter T van de Supplier interface gebruikt om verschillende typen te ondersteunen. We hebben eerst een Supplier<Integer> gemaakt om een willekeurig getal tussen 0 en 100 te genereren en af te drukken. Vervolgens hebben we een Supplier<String> gemaakt om een vaste willekeurige tekst te retourneren en af te drukken.

Function<T, R> met Generics:

javaCopy code

import java.util.function.Function;

public class FunctionExample {

public static void main(String[] args) { // Function voor het converteren van een getal naar een String

Function<Integer, String> toStringFunction = (num) -> String.valueOf(num); // Gebruik van de Function om een getal naar een String te converteren

String numberString = toStringFunction.apply(123);

System.out.println("Getal als String: " + numberString); // Function voor het converteren van een String naar een Integer

Function<String, Integer> toIntegerFunction = Integer::valueOf; // Gebruik van de Function om een String naar een Integer te converteren

Integer integerValue = toIntegerFunction.apply("456");

System.out.println("String als Integer: " + integerValue);

}

}

In dit voorbeeld hebben we de generieke parameters T en R van de Function interface gebruikt om verschillende typen te ondersteunen. We hebben eerst een Function<Integer, String> gemaakt om een getal naar een String te converteren en af te drukken. Vervolgens hebben we een Function<String, Integer> gemaakt om een String naar een Integer te converteren en af te drukken.

Het gebruik van generics stelt ons in staat om generieke code te schrijven die kan werken met verschillende typen, waardoor herbruikbaarheid en typeveiligheid worden bevorderd. Generics worden vaak gebruikt in Java-collections, streams en functionele interfaces om flexibele en typeveilige code te schrijven.

##### combinatie

// Abstracte klasse met een generieke methode

abstract class AbstractCalculator<T> {

public abstract T calculate(T num1, T num2);

}

// Concrete klasse die de abstracte klasse uitbreidt voor Integer berekeningen

class IntegerCalculator extends AbstractCalculator<Integer> {

public Integer calculate(Integer num1, Integer num2) {

return num1 + num2;

}

}

// Concrete klasse die de abstracte klasse uitbreidt voor Double berekeningen

class DoubleCalculator extends AbstractCalculator<Double> {

public Double calculate(Double num1, Double num2) {

return num1 \* num2;

}

}

// Functionele interface met een enkele methode

@FunctionalInterface

interface Operation<T> {

T operate(T num1, T num2);

}

public class InterfaceAndGenericsExample {

public static void main(String[] args) {

// Abstracte klasse gebruiken

AbstractCalculator<Integer> integerCalculator = new IntegerCalculator();

Integer result1 = integerCalculator.calculate(5, 3);

System.out.println("Integer result: " + result1);

AbstractCalculator<Double> doubleCalculator = new DoubleCalculator();

Double result2 = doubleCalculator.calculate(2.5, 4.5);

System.out.println("Double result: " + result2);

// Functionele interface gebruiken

Operation<Integer> addOperation = (num1, num2) -> num1 + num2;

Integer result3 = addOperation.operate(10, 20);

System.out.println("Addition result: " + result3);

Operation<Double> multiplyOperation = (num1, num2) -> num1 \* num2;

Double result4 = multiplyOperation.operate(2.5, 4.5);

System.out.println("Multiplication result: " + result4);

}

}

We hebben een abstracte klasse AbstractCalculator<T> gedefinieerd met een generieke methode calculate(T num1, T num2). Deze klasse kan worden uitgebreid door klassen die specifieke berekeningen willen uitvoeren op verschillende typen.

We hebben twee concrete klassen IntegerCalculator en DoubleCalculator gemaakt die de AbstractCalculator<T> klasse uitbreiden voor respectievelijk Integer en Double berekeningen.

In de main-methode maken we een instantie van de IntegerCalculator en gebruiken we de calculate-methode om de som van twee integer-getallen te berekenen en op te slaan in result1.

Daarna maken we een instantie van de DoubleCalculator en gebruiken we de calculate-methode om het product van twee double-getallen te berekenen en op te slaan in result2.

We drukken beide resultaten af naar de console.

Vervolgens hebben we een functionele interface Operation<T> gedefinieerd met een enkele methode operate(T num1, T num2).

We hebben twee instanties van de functionele interface gemaakt, addOperation en multiplyOperation, die respectievelijk de optelling en vermenigvuldiging uitvoeren.

We gebruiken de operate-methode van de functionele interface om de berekeningen uit te voeren en de

https://github.com/matthijsvandermaas/Leleystad-airport