# Generics

### Voorbeeld

### Te gebruiken bij:

- Generics stellen je in staat om algemene code te schrijven die kan werken met verschillende datatypen, zonder de exacte datatypen op voorhand te kennen. Het biedt typeveiligheid tijdens het compileren en maakt herbruikbare code mogelijk.

### Eigenschappen:

-

### Info

Hiërarchie geeft ons de mogelijkheid om algemene stukjes code slechts op één plek te hoeven schrijven, omdat we properties en variabelen kunnen voorzien van generieke, hoger gelegen super-class-types.

deze manier van code schrijven is een **best practise**.

zorgt ervoor dat we maximale flexibiliteit in onze code behouden en niet telkens alles overhoop hoeven te gooien wanneer er bijvoorbeeld een nieuw betaalmiddel bij komt.

Dat doen we door de class te voorzien van een speciale variabele, die tussen <> wordt gedefinieerd:

// we definiëren hier de generic als letter 'P', maar dit mag net als een variabele van alles zijn

class Payment<P> {

int amount;

// we vervangen hier het type object met de variabele van de generic

P paymentMethod;

public void setAmount(int amount) {

this.amount = amount;

}

// ... en hier vervangen we ook het type object met de variabele van de generic

public void setPaymentMethod(P paymentMethod) {

this.paymentMethod = paymentMethod;

}

public void submit() {

// imlpementatie voor uitvoeren van de betaling

}

}

Let op: bovengenoemde syntax is expres heel simpel gehouden.

Doordat we generics hebben gebruikt, hoeven we het exacte type niet meer te definiëren bij het declareren van de class. We hoeven het exacte type nu pas mee te geven wanneer we een instantie maken van de class:

Payment<PaymentMethod> payment = new Payment<>();

bij een class die generics gebruikt altijd gebruik moeten maken van de <>, zowel in het type van de variabele als bij het gebruik van de constructor.

Payment<CreditCard> payment = new Payment<>();

Afbeelding met tekst, schermopname, Lettertype, diagram

Automatisch gegenereerde beschrijving

Payment<CreditCard> payment = new Payment<>();

payment.setPayment(new DebitCard(...));

Een debitcard werkt nu niet meer, omdat DebitCard geen hiërarchische afgeleide is van CreditCard. We hebben nu zonder een nieuwe implementatie van Payment te maken toch een beperking kunnen realiseren. Willen we alleen PayPal toestaan?

Payment<PayPal> payment = new Payment<>();

#### Overzicht

een voorbeeld bekijken van het gebruik van generics in een eenvoudige Pair-klasse die twee waarden van hetzelfde datatype kan bevatten.

java

public class Pair<T> {

private T first;

private T second;

public Pair(T first, T second) {

this.first = first;

this.second = second;

}

public T getFirst() {

return first;

}

public T getSecond() {

return second;

}

public void setFirst(T first) {

this.first = first;

}

public void setSecond(T second) {

this.second = second;

}

}

In dit voorbeeld hebben we een generieke Pair-klasse gedefinieerd. Het generieke type T kan elk willekeurig datatype zijn dat we later zullen specificeren bij het maken van een instantie van Pair.

Hier is hoe je Pair kunt gebruiken met verschillende datatypes:

java

Pair<Integer> numberPair = new Pair<>(10, 20);

System.out.println("First: " + numberPair.getFirst()); // Output: First: 10

System.out.println("Second: " + numberPair.getSecond()); // Output: Second: 20

Pair<String> stringPair = new Pair<>("Hello", "World");

System.out.println("First: " + stringPair.getFirst()); // Output: First: Hello

System.out.println("Second: " + stringPair.getSecond()); // Output: Second: World

In dit geval maken we een instantie van Pair<Integer> en een instantie van Pair<String>. We kunnen de getFirst()- en getSecond()-methoden gebruiken om de waarden op te halen en deze afdrukken.

Het gebruik van generics in de Pair-klasse zorgt ervoor dat we dezelfde klasse kunnen hergebruiken voor verschillende datatypes, terwijl de typeveiligheid behouden blijft. Dit voorkomt bijvoorbeeld dat we per ongeluk een Integer-waarde toewijzen aan de second-variabele van een Pair<String>, omdat het datatype op compileertijd wordt afgedwongen.

je kunt een lijst maken van Pair-objecten door gebruik te maken van generics. Hier is een voorbeeld van hoe je een lijst van Pair<Integer> kunt maken en enkele elementen kunt toevoegen:

java

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

List<Pair<Integer>> pairList = new ArrayList<>();

pairList.add(new Pair<>(1, 2));

pairList.add(new Pair<>(3, 4));

pairList.add(new Pair<>(5, 6));

// Itereren over de lijst en waarden afdrukken

for (Pair<Integer> pair : pairList) {

System.out.println("First: " + pair.getFirst() + ", Second: " + pair.getSecond());

}

}

}

In dit voorbeeld maken we een lijst met het generieke type Pair<Integer>, genaamd pairList. We voegen enkele Pair<Integer>-objecten toe aan de lijst met behulp van de add()-methode.

Vervolgens gebruiken we een for-each loop om over de lijst te itereren en de waarden van elk Pair<Integer>-object af te drukken.

De output van dit voorbeeld zal zijn:

sql

First: 1, Second: 2

First: 3, Second: 4

First: 5, Second: 6

Hiermee kun je een lijst maken van Pair-objecten waarvan het generieke type kan worden gespecificeerd, waardoor je flexibiliteit hebt om verschillende datatypes te gebruiken binnen elk Pair-object in de lijst.

#### Mogelijkheden

##### Generieke klassen met meerdere typeparameters:

java

public class Pair<T, U> {

private T first;

private U second;

public Pair(T first, U second) {

this.first = first;

this.second = second;

}

public T getFirst() {

return first;

}

public U getSecond() {

return second;

}

}

// Gebruik:

Pair<String, Integer> pair = new Pair<>("Hello", 123);

String str = pair.getFirst();

Integer num = pair.getSecond();

In dit voorbeeld hebben we een generieke Pair-klasse met twee typeparameters T en U. Hiermee kun je een Pair maken met twee verschillende datatypen.

##### Generieke methoden:

java

public class Utils {

public static <T> T getFirstElement(List<T> list) {

if (list.isEmpty()) {

return null;

}

return list.get(0);

}

}

// Gebruik:

List<String> stringList = Arrays.asList("A", "B", "C");

String firstElement = Utils.getFirstElement(stringList);

In dit voorbeeld hebben we een generieke methode getFirstElement die het eerste element van een lijst retourneert. Het generieke type T wordt gebruikt om het datatype van de elementen in de lijst aan te geven.

##### Bounded type parameters:

java

Copy code

public class MathUtils {

public static <T extends Number> double average(List<T> numbers) {

double sum = 0.0;

for (T number : numbers) {

sum += number.doubleValue();

}

return sum / numbers.size();

}

}

// Gebruik:

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

double avg = MathUtils.average(integerList);

In dit voorbeeld hebben we een generieke methode average die een lijst van getallen accepteert. De typeparameter T wordt beperkt tot subklassen van Number. Dit betekent dat je alleen numerieke datatypen kunt gebruiken met de average-methode.

Deze voorbeelden illustreren enkele van de mogelijkheden van generics in Java, waaronder generieke klassen met meerdere typeparameters, generieke methoden en het gebruik van begrensde typeparameters. Generics bieden flexibiliteit en veiligheid bij het werken met verschillende datatypen.

User

nu voor beeld 4 en 5

ChatGPT

Natuurlijk! Hier zijn voorbeelden van mogelijkheid 4 en 5 van het gebruik van generics in Java:

##### Wildcard-parameters:

java

public class Printer {

public static void printList(List<?> list) {

for (Object item : list) {

System.out.println(item);

}

}

}

// Gebruik:

List<String> stringList = Arrays.asList("Hello", "World");

List<Integer> integerList = Arrays.asList(1, 2, 3);

Printer.printList(stringList);

Printer.printList(integerList);

In dit voorbeeld hebben we een generieke methode printList die een lijst accepteert van onbekend type (List<?>). We gebruiken het wildcard-teken (?) om aan te geven dat we het exacte datatype niet kennen, maar we kunnen nog steeds items uit de lijst halen en ze afdrukken.

##### Generieke interfaces:

java

public interface Repository<T> {

void save(T item);

T findById(int id);

}

public class UserRepository implements Repository<User> {

// Implementeer de methoden van de interface

// ...

}

// Gebruik:

UserRepository userRepository = new UserRepository();

userRepository.save(new User());

User user = userRepository.findById(1);

In dit voorbeeld hebben we een generieke interface Repository gedefinieerd met een typeparameter T. De interface bevat methoden om een item op te slaan en op te zoeken op basis van een ID. Vervolgens implementeren we de Repository-interface in de UserRepository-klasse en specificeren we het concrete datatype User. Dit stelt ons in staat om de interface-methoden te implementeren en te werken met de User-klasse.

Generics bieden in dit geval de mogelijkheid om generieke interfaces te definiëren en deze vervolgens te implementeren met specifieke datatypen, waardoor codehergebruik en flexibiliteit worden geboden.