HoGent
BEDRIJF
EN
ORGANISATIE

Hoofdstuk 20:
GENERIEKE METHODEN EN GENERIEKE KLASSEN

20.1 INLEIDING

- Generieke methoden en generieke klassen zijn nieuw sedert J2SE 5.0.

- Voorbeeld 1: we wensen arrays van integers en arrays van strings te sorteren. In java 1.4 dienen we overloaded methoden sort te schrijven:

```
sort(String[] lijst)
sort(Integer[] lijst)
```

In J2SE 5.0 schrijven we slechts één **generieke methode**:

```
<E> void sort(E[] lijst)
```

Op het moment dat de methode sort wordt opgeroepen, wordt E vervangen door het type van het argument:

```
Integer[] integerArray = {2, 4, 1};
String[] stringArray = {“test”, “abc”, “xyz”};
sort(integerArray)
sort(StringArray)
```

- Voorbeeld 2: we wensen een stack van Integers en een stack van Strings te schrijven. In java 1.4 dienen we twee klassen te schrijven.

```
public class Stack
{ private Integer[] elementen; …

public class Stack
{ private String[] elementen; …
```

In J2SE 5.0 schrijven we slechts één **generieke klasse**:

```
public class Stack<E>
{ private E[] elementen; …
```

Op het moment van creatie wordt het type bepaald.

```
Stack<Integer> integerStack;
Stack<String> stringStack;
```

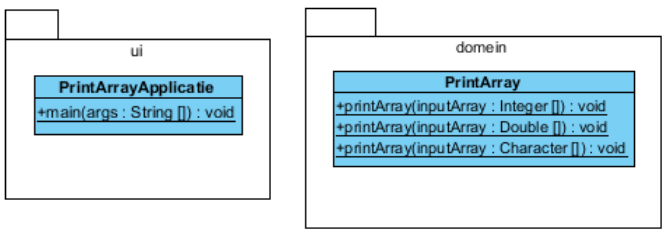
- Voordelen van generieke methoden en generieke klassen:
  - Grote herbruikbaarheid
  - Bij het oproepen van een generieke methode of generieke klasse wordt tijdens compilatie het type gecontroleerd.

20.2 Het nut van een generieke methode

- **Voorbeeld**:

  We wensen alle elementen van een array weer te geven op het scherm. De elementen van de array zijn van type Integer, Double of Character.

  ➔ We zullen drie overloaded methoden schrijven.



OVERLOADED METHODEN

```
public class PrintArray
{
   // geeft een array van Integers weer op het scherm
   public static void printArray( Integer[] inputArray )
   {
       Arrays.stream(inputArray).forEach
           (element -> System.out.printf("%s ", element));
       System.out.println();
   } // einde methode printArray

  // geeft een array van Doubles weer op het scherm
   public static void printArray( Double[] inputArray )
   {
       Arrays.stream(inputArray).forEach
           (element -> System.out.printf("%s ", element));
       System.out.println();
} // einde methode printArray
```

OVERLOADED METHODEN

```
   // geeft een array van Characters weer op het scherm
   public static void printArray( Character[] inputArray )
   {
       Arrays.stream(inputArray).forEach
           (element -> System.out.printf("%s ", element));
      System.out.println();
   } // einde methode printArray

} // einde klasse PrintArray
```

- Drie identieke methoden, enkel het type van de elementen van de array is verschillend.

HoGent

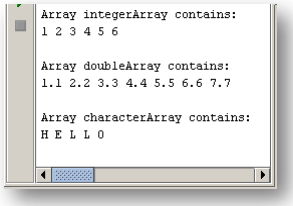
DE OVERLOADED METHODEN OPROEPEN

```
public class PrintArrayApplicatie
{
    public static void main( String args[] )
    {
    // creatie van arrays van Integer, Double en Character
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println( "Array integerArray contains:" );
    PrintArray.printArray( integerArray ); // een Integer-array wordt doorgegeven
    System.out.println( "\nArray doubleArray contains:" );
    PrintArray.printArray( doubleArray ); // een Double-array wordt doorgegeven
    System.out.println( "\nArray characterArray contains:" );
    PrintArray.printArray( characterArray ); // een Character-array wordt
                                             // doorgegeven
  } // einde main
} // einde klasse PrintArrayApplicatie
```

- Hetzelfde voorbeeld wordt herschreven door gebruik te maken van een **generieke methode**:

```
public class PrintArray
{
  public static < E > void printArray( E[] inputArray )
  {
      Arrays.stream(inputArray).forEach
              (element -> System.out.printf("%s ", element));
      System.out.println();
    } // einde methode printArray
} // einde klasse PrintArray
```

- Klasse PrintArrayApplicatie is identiek.



20.3 Generieke methoden: implementatie en "compile-time" vertaling

- Alle generieke methoden hebben een **"type parameter section"**, dat tussen < en > staat.
- Elke **"type parameter section"** bevat één of meer **"(formal) type parameters"**, gescheiden door komma's.
- Een "**type parameter"** is een identifier, die gebruikt kan worden als return-type, parameter-type en type voor lokale variabelen.
- Een "**type parameter"** kan enkel referenties bevatten, geen primitieve datatypes!


```
public static < E > void printArray( E[] inputArray ){
     Arrays.stream(inputArray).forEach
              (element -> System.out.printf("%s ", element));
    /*OF: for ( E element : inputArray )
       System.out.printf( "%s ", element ); */

    System.out.println();
} // einde methode printArray
```

- De generieke methode "printArray" heeft als **"type parameter section"** <E>.
- De **"type parameter section"** <E> bevat één **"type parameter"**, nl. E.
- E wordt hier gebruikt als parameter-type (**E[]** inputArray) en type voor de lokale variabele **element**.

public static < E, W > void voorbeeld(E[] anArray , W waarde)
{ … }

- De generieke methode “voorbeeld” heeft als **“type parameter section”** <E, W>.
- <E, W> bevat twee **“type parameters”**, nl. E, W.

- De methode kan bvb. opgeroepen worden als

```
Integer[] integerArray = { 1, 2, 3};
String test = “test”;
voorbeeld(integerArray, test);
```

---

public static < E> void printTwoArrays(E[] array1 , E[] array2)
{ … }

- De **type parameter** E mag meerdere keren als parameter-type worden gebruikt.

“compile-time” vertaling

- Bij het oproepen van een generieke methode wordt, **tijdens compilatie**, het type gecontroleerd.

Voorbeeld:
Integer getal = 20;
PrintArray.printArray(getal); ➔ **geeft een compileerfout**

public static **< E >** void printArray(**E[]** inputArray)

20.4 Generieke methode met een return-type

- **Voorbeeld**:
  Het maximum bepalen van drie waarden. De maximum waarde wordt teruggegeven.

  public static < T extends Comparable< T > > **T** maximum(T x, T y, T z)

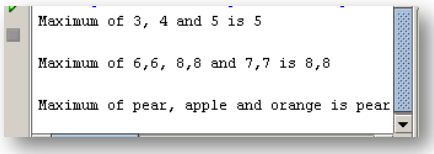
  Het is een generieke methode waarvan het parameter type een implementatieklasse van Comparable<T> moet zijn. De interface Comparable is zelf generiek.

GENERIEKE METHODE

```
public class Operation
{
  // geeft het grootste van de drie Comparable-objecten terug
  public static < T extends Comparable< T > > T maximum( T x, T y, T z )
  {  /* java7
     T max = x; // het eerste object is momenteel het grootste object.
     if ( y.compareTo( max ) > 0 )
        max = y; // het object y is momenteel het grootste object.
     if ( z.compareTo( max ) > 0 )
        max = z; // het object z is het grootste object.
     */

     T max=Arrays.asList(x,y,z).stream().max(T::compareTo).get();

     return max; // geeft het grootste object terug
  } // end method maximum
}
```

OPROEPEN VAN GENERIEKE METHODE

```
public class MaximumApplicatie
{
   public static void main( String args[] )
   {
      System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
         Operation.maximum( 3, 4, 5 ) );
      System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, Operation.maximum( 6.6, 8.8, 7.7 ) );
      System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
         "apple", "orange", Operation.maximum( "pear", "apple", "orange" ) );
   }

}
```



HoGent

20.5 Overloaded generieke methoden

- We mogen meerdere generieke methoden schrijven waarvan de parameterlijsten verschillen.

  voorbeeld:

  public static **< E >** void printArray(**E[]** inputArray){…}

  public static **< E >** void printArray(**E[]** inputArray , int beginIndex){…}

- Een generieke methode en een gewone methode mogen ook overloaded zijn.

  public static **< E >** void printArray(**E[]** inputArray){…}

  public static void printArray(**String[]** inputArray)

  { //de elementen worden in tabelvorm weergegeven

  … }

HoGent

20.6 Generieke klassen

- Het type van het attribuut (attributen) in een generieke klasse is nog niet bepaald.
- **Voorbeeld:** we maken een Stack. De stack bevat het attribuut: E[] elements
  "elements" is een array, waarvan het type nog niet bepaald is.
- De klasse wordt bewaard als **Stack.java** en niet als Stack<E>.java

GENERIEKE KLASSE "Stack"

```
public class Stack< E >
{
   private final int SIZE; // aantal elementen van een stack
   private int top; // locatie van de top van de stack
   private E[] elements; // array die de elementen van de stack zal bevatten

   // defaultconstructor; creëert een stack van 10 elementen
   public Stack()
   {
      this( 10 );
    }

   // constructor; creëert een stack van "s" of "10" elementen
   public Stack( int s )
   {
      SIZE = s > 0 ? s : 10; // set size of Stack
      top = -1; // stack is leeg
      elements = ( E[] ) new Object[ SIZE ]; // creatie van de array
   }
```

public class Stack< E >

- Alle generieke klassen hebben een **“type parameter section”**, dat tussen < en > staat.
- Elke **“type parameter section”** bevat één of meer **“(formal) type parameters”**, gescheiden door komma’s.
- Een “**type parameter”** is een identifier, die gebruikt wordt als attribuut-type. Het kan ook gebruikt worden als return-type, parameter-type en type voor lokale variabelen.
- ***Vb:*** public class FileVerwerking< E extends Serializable>
  De “**type parameter”** E moet een implementatieklasse van Serializable zijn.

```
public Stack( int s )
{ SIZE = s > 0 ? s : 10; // set size of Stack
  top = -1; // stack is leeg
  elements = ( E[] ) new Object[ SIZE ];
}
```

elements = new E[size]; → compileerfout

We kunnen geen array van type **type parameter** (in ons geval E) creëren, omdat E niet bestaat in runtime.

We dienen een array van Objecten te creëren en om te zetten naar een array van type E.

Wel krijgen we een waarschuwing tijdens compilatie: **“java uses unchecked or unsafe operations.”**. De array van type Object zou objecten verschillend van type E kunnen bevatten.

GENERIEKE KLASSE “Stack”

```
// voeg een element toe aan de stack; indien gelukt, return true;
// anders, throw FullStackException
public void push( E pushValue )
{
   if ( top == SIZE - 1 ) // de stack is vol
      throw new FullStackException( String.format(
         "Stack is full, cannot push %s", pushValue ) );

   elements[ ++top ] = pushValue; // place pushValue on Stack
} // end method push
```


GENERIEKE KLASSE “Stack”

```
   // indien de stack niet leeg is wordt het top element teruggegeven, anders //
    EmptyStackException
   public E pop()
   {
      if ( top == -1 ) // indien de stack leeg is
         throw new EmptyStackException( "Stack is empty, cannot pop" );

      return elements[ top-- ]; // verwijder en geef het top element terug
   } // einde methode pop
} // einde class Stack< E >
```

```
public class EmptyStackException extends RuntimeException
{

   public EmptyStackException()
   {
      this( "Stack is empty" );
   }

   public EmptyStackException( String exception )
   {
      super( exception );
   }

} // einde klasse EmptyStackException
```

HoGent

```
public class FullStackException extends RuntimeException
{

   public FullStackException()
   {
      this( "Stack is full" );
   }

   public FullStackException( String exception )
   {
      super( exception );
   }

} // end class FullStackException
```

HoGent

GENERIEKE KLASSE

“Stack” uittesten

```
public class StackApplicatie
{
   private double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
   private int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

   private Stack< Double > doubleStack;
   private Stack< Integer > integerStack;
```

- “doubleStack” is een object van Stack<Double>. In de generieke klasse Stack zal E vervangen worden door Double.
- Idem, behalve Integer i.p.v. Double.

HoGent

```
// Stack-objecten testen
public void testStacks()
{
   doubleStack = new Stack<>( 5 ); // Stack van Doubles
   integerStack = new Stack<>( 10 ); // Stack van Integers

   testPushDouble(); // waarden van type double in de doubleStack plaatsen
   testPopDouble(); // waarden van de doubleStack afhalen
   testPushInteger(); // waarden van type int in de integerStack plaatsen
   testPopInteger(); // waarden van de integerStack afhalen
} // einde methode testStacks
```

- Creatie van de array “doubleStack” en heeft als lengte 5.
- Creatie van de array “integerStack” en heeft als lengte 10.

HoGent

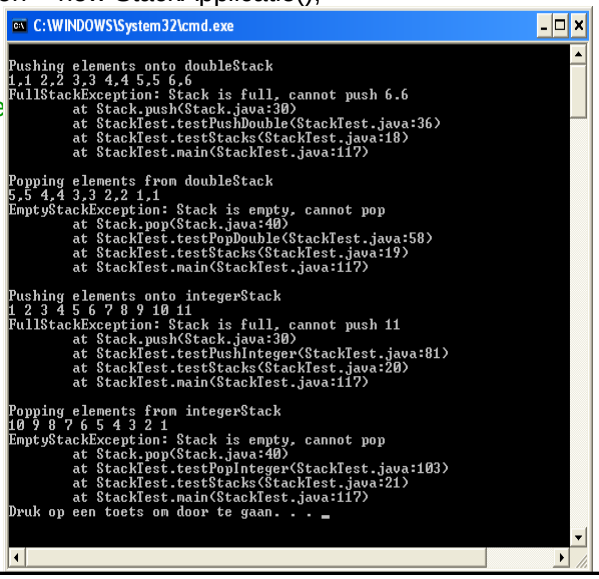
```
// waarden van type double in de doubleStack plaatsen
public void testPushDouble()
{
   try {
          System.out.println( "\nPushing elements onto doubleStack" );

         // doubles op de stack plaatsen
         for ( double element : doubleElements )
        {
           System.out.printf( "%.1f ", element );
           doubleStack.push( element );
        } // end for
   } // end try
   catch ( FullStackException fullStackException )
   {  System.err.println();
      fullStackException.printStackTrace();
   } // end catch FullStackException
} // end method testPushDouble
```

```
// waarden van de doubleStack afhalen
public void testPopDouble()
{
   try
   {  System.out.println( "\nPopping elements from doubleStack" );
      double popValue; // het element bijhouden dat van de stack werd
                       // verwijderd.
      // alle elementen van de stack afhalen
      while ( true )
      {
         popValue = doubleStack.pop();
         System.out.printf( "%.1f ", popValue );
      } // end while
   } // end try
   catch( EmptyStackException emptyStackException )
   {  System.err.println();
      emptyStackException.printStackTrace();
   } // end catch EmptyStackException
} // end method testPopDouble
```

```
// waarden van type int in de integerStack plaatsen
public void testPushInteger()
{
   try
   {  System.out.println( "\nPushing elements onto integerStack" );

      for ( int element : integerElements )
      {
         System.out.printf( "%d ", element );
         integerStack.push( element );
      } // end for
   } // end try
   catch ( FullStackException fullStackException )
   {
      System.err.println();
      fullStackException.printStackTrace();
   } // end catch FullStackException
} // end method testPushInteger
```

```
// waarden van de integerStack afhalen
public void testPopInteger()
{
   try
   {
      System.out.println( "\nPopping elements from integerStack" );
      int popValue; // het element bijhouden dat van de stack werd
                    // verwijderd.
      // alle elementen van de stack afhalen
      while ( true )
      {  popValue = integerStack.pop();
         System.out.printf( "%d ", popValue );
      } // end while
   } // end try
   catch( EmptyStackException emptyStackException )
   {  System.err.println();
      emptyStackException.printStackTrace();
   } // end catch EmptyStackException
} // end method testPopInteger
```

```
public static void main( String args[] )
  {
     StackApplicatie application = new StackApplicatie();
     application.testStacks();
  } // end main
} // end class StackApplicatie
```



HoGent

Generieke klasse Stack uittesten d.m.v. generieke methoden

In het vorig voorbeeld zien we dat de methoden testPopDouble() en testPopInteger() bijna identiek zijn.

Hetzelfde geldt voor testPushDouble() en testPushInteger().

```
public void testPushDouble()
{
 try {

 System.out.println("…doubleStack" );

 for ( double element : doubleElements )
 {
   System.out.printf( "%.1f ", element );
   doubleStack.push( element );
```

```
public void testPushInteger()
{
 try {

 System.out.println("…integerStack" );

 for ( int element : integerElements )
 {
   System.out.printf( "%d ", element );
   integerStack.push( element );
```

We kunnen dit vermijden door generieke methoden te gebruiken.

KLASSE “StackApplicatie2”

bevat generieke methoden

```
// Klasse “StackApplicatie2” om de klasse Stack<E> te testen.

public class StackApplicatie2
{
   private Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
   private Integer[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

   private Stack< Double > doubleStack; // stack stores Double objects
   private Stack< Integer > integerStack; // stack stores Integer objects
```

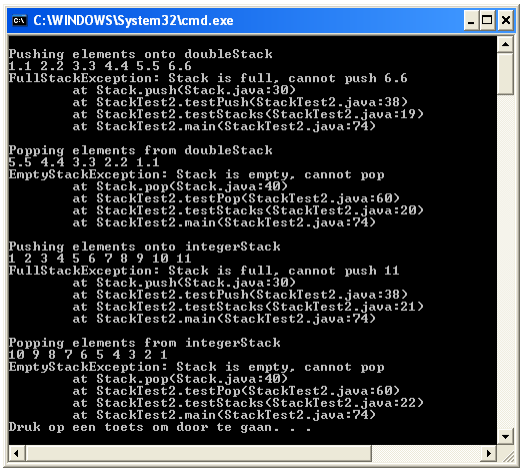


```
   // Stack-objecten testen
   public void testStacks()
   {
      doubleStack = new Stack<>( 5 ); // Stack of Doubles
      integerStack = new Stack<>( 10 ); // Stack of Integers

       //generieke methoden oproepen
       testPush( "doubleStack", doubleStack, doubleElements );
       testPop( "doubleStack", doubleStack );
       testPush( "integerStack", integerStack, integerElements );
       testPop( "integerStack", integerStack );

   } // end method testStacks
```

```
// generieke methode testPush
public < T > void testPush( String name, Stack< T > stack, T[] elements )
{
   try
   {  System.out.printf( "\nPushing elements onto %s\n", name );

      for ( T element : elements )
      {
         System.out.printf( "%s ", element );
         stack.push( element );
      }
   } // end try
   catch ( FullStackException fullStackException )
   {
      System.out.println();
      fullStackException.printStackTrace();
   } // end catch FullStackException
} // end method testPush
```

```
// generieke methode testPop
public < T > void testPop( String name, Stack< T > stack )
{
   // pop elements from stack
   try {  System.out.printf( "\nPopping elements from %s\n", name );
          T popValue; // store element removed from stack

          while ( true )
          {
             popValue = stack.pop();
             System.out.printf( "%s ", popValue );
          } // end while
       } // end try
       catch( EmptyStackException emptyStackException )
       {  System.out.println();
             emptyStackException.printStackTrace();
        }
 } // end method testPop
```

```
public static void main( String args[] )
  {
    StackApplicatie2 application = new StackApplicatie2();
    application.testStacks();
  } // end main
} // end class StackApplicatie2
```



HoGent

- ***Polymorfisme*: De klasse "StackApplicatie2" kan als volgt herschreven worden:**

```
public class StackApplicatie3
{ private Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
  private Integer[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
  private Object[][] elements = {doubleElements, integerElements};
  private String[] nameStack = {"doubleStack","integerStack" };
  private Stack[] stack = new Stack[2];

  public void testStacks()
  {       stack[0] = new Stack< Double >( 5 );
          stack[1] = new Stack<Integer >(10);
          for (int i=0; i<stack.length;i++)
          {       testPush( nameStack[i], stack[i], elements[i] );
          testPop( nameStack[i],  stack[i]);
          }
  } // end method testStacks
```

20.7 “Raw Types”

Voorbeeld van een “raw type” stack-variabele:

a) Stack rawTypeStack = new Stack<Double>(5);

b) Stack<Integer> integerStack = new Stack(10);

- in geval b) krijgen we een waarschuwing tijdens compilatie: **`“java uses unchecked or unsafe operations.”`**.

  → De “raw type” stack (Stack(10)) zou objecten verschillend van type Integer kunnen bevatten.
- geval a) kan dan ook weer zo’n foutmelding opleveren als deze referentie wordt doorgegeven als actueel argument waar als formeel argument niet een “raw type” verwacht wordt. De rawTypeStack garandeert niet dat de elementen Double’s zullen zijn.

HoGent

20.8 Wildcards

- Waarom Wildcards gebruiken? We gaan dit aantonen door middel van een voorbeeld.
- We zullen de generieke methode **`sum`** implementeren, die de elementen van een Collection sommeert.

HoGent

```
import java.util.*;
public class TotalNumbers
{ public static void main( String args[] )
  {
    Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers en Doubles
   //autoboxing : int 1 en 3 worden automatisch omgezet naar Integer
          //double 2.4 en double 4.1 worden automatisch omgezet naar Double
    Collection< Number > numberList =
              new ArrayList<>(Arrays.asList(numbers));


    System.out.printf( "numberList contains: %s\n", numberList );
    System.out.printf( "Total of the elements in numberList: %.1f\n",
      sum( numberList ) );
  } // end main
```

HoGent

```
  // sommeert de elementen van een collection
  public static double sum( Collection< Number > list )
  {
    double total = 0;

    for ( Number element : list )
      total += element.doubleValue();

    return total;
  } // end method sum
} // end class TotalNumbers
```

HoGent

- We wensen hetzelfde programma uit te testen voor een Collection, waarvan de elementen van type Integer zijn. We passen het volgende aan:

```
public class TotalNumbers
{
        public static void main( String args[] )
        {
                Integer[] numbers = { 1, 2, 3, 4 };
        Collection< Integer > numberList =
                new ArrayList<>(Arrays.asList(numbers));
                …
                System.out.printf( …, sum( numberList ) );
                …
```

- We krijgen een compileerfout **"sum(java.util.Collection<java.lang.Number>) in TotalNumbers cannot be applied to (java.util.Collection<java.lang.Integer>)"**

- Nochtans is Number de superklasse van Integer, maar de compiler beschouwt Collection<Integer> niet als een subtype van Collection<Number>.

- ➔ **oplossing: wildcards:**

```
public static double sum( Collection< ? extends Number > list )
```

HoGent

20.8 Wildcards

```
import java.util.ArrayList;

public class WildcardApplicatie
{
   public static void main( String args[] )
   {
      Integer[] integers = { 1, 2, 3, 4, 5 };
      Collection< Integer > integerList =
                  new ArrayList<>(Arrays.asList(integers));


      System.out.printf( "integerList contains: %s\n", integerList );
      System.out.printf( "Total of the elements in integerList: %.0f\n\n",
        sum( integerList ) );
```

HoGent

20.8 Wildcards

```
      Double[] doubles = { 1.1, 3.3, 5.5 };
      Collection< Double > doubleList =
                  new LinkedList<>(Arrays.asList(doubles));

      System.out.printf( "doubleList contains: %s\n", doubleList );
      System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
        sum( doubleList ) );

      Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
      Collection< Number > numberList =
                  new ArrayList<>(Arrays.asList(numbers));

      System.out.printf( "numberList contains: %s\n", numberList );
      System.out.printf( "Total of the elements in numberList: %.1f\n",
        sum( numberList ) );
   } // end main
```

HoGent

20.8 Wildcards

```
// calculate total of stack elements
public static double sum( Collection< ? extends Number > list )
{
   double total = 0; // initialize total

   // calculate sum
   for ( Number element : list )
      total += element.doubleValue();

   return total;
} // end method sum
} // end class WildcardApplicatie
```

HoGent

20.9 Overerving

- Een generieke klasse kan erven van een niet-generieke klasse.
- Een generieke klasse kan erven van een generieke klasse.
- Een niet-generieke klasse kan erven van een generieke klasse, vb. de klasse **Properties** erft van de klasse **Hashtable<K,V>**.
- Een generieke methode in een subklasse kan een generieke methode van de superklasse overschrijven indien ze dezelfde header hebben.

HoGent

20.10 Web resources

- Java Resource Center www.deitel.com/Java/
  Resource Center Contents
  Java Generics

HoGent