universität innsbruck

Generische Programmierung

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Motivation

Motivation (Beispiel – Pair-Klasse)

```java
public final class IntegerPair {
    private final Integer first;
    private final Integer second;
    public IntegerPair(Integer first, Integer second) {
        this.first = first;
        this.second = second;
    }
    public Integer getFirst() {
        return first;
    }
    public Integer getSecond() {
        return second;
    }
}
```

```java
public final class DoublePair {
    private final Double first;
    private final Double second;
    public DoublePair(Double first, Double second) {
        this.first = first;
        this.second = second;
    }
    public Double getFirst() {
        return first;
    }
    public Double getSecond() {
        return second;
    }
}
```

Fast identische Implementierung
(duplizierter Code)
- nur Double statt Integer.
bessere Lösung ?

Beispiel (Object-Pair)

```java
public final class Pair {
    private final Object first;
    private final Object second;

    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() {
        return first;
    }

    public Object getSecond() {
        return second;
    }
}
```

Wurzelklasse `Object` und Ersetzungsprinzip ausnutzen

Problem gelöst?

Typsicherheit?

Allgemeine Implementierung

- Es werden nur Objekte vom Typ `Object` verwaltet.
- Alle Objekte (durch Autoboxing auch primitive Datentypen) können verwaltet werden.
- Eine Implementierung für unterschiedliche Typen!
- Aber
  - Getter-Methoden liefern nur ein Objekt vom Typ `Object`, das für die weitere Benutzung gecastet werden muss.
  - Keine **Typsicherheit** und daher umständliche Programmierung notwendig → `ClassCastException` zur Laufzeit.
  - Zwei Instanzen für Typsicherheit: Compiler und JVM zur Laufzeit.
- Lösung?

Generics

Generics

- Generics (Generizität) erlaubt es, einen Programmteil (Klasse, Interface, Methode) mit Typen zu parametrisieren.
- Seit Java 5
- Unterscheidung
  - Generischer Typ
  - Parametrisierter Typ

Generischer Typ

- Ein generischer Typ wird durch die entsprechende generische Klasse oder das generische Interface definiert.
  - Definition wie eine normale Klasse bzw. ein normales Interface.
  - Zusätzlich folgt nach dem Namen eine Typparameter-Sektion:

    ```
    public class Stack<T> {…}
    ```

    - `<T>` ist die Typparameter-Sektion.
    - `T` wird als formaler Typparameter oder Typvariable bezeichnet.
    - Für den formalen Typparameter kann ein beliebiger Bezeichner gewählt werden. Empfehlung: *Großbuchstabe*
- Innerhalb der Klasse bzw. des Interfaces wird die Typvariable fast wie ein normaler Typ verwendet.
  - Bei Übergabe- bzw. Rückgabeparameter etc.
  - Einschränkungen werden noch erklärt.
- In der Typparameter-Sektion können, durch Beistriche getrennt, mehrere Typvariablen angegeben werden.

  ```
  public class Pair<T, U> {…}
  ```

Beispiel generischer Typ

```java
public final class Pair<T, U> {
    private final T first;
    private final U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }
}
```

src/at/ac/uibk/pm/generics/pair/Pair.java

Parametrisierter Typ (1)

- Ein parametrisierter Typ ist ein Typ in der Form `C<T1, …, TN>`, wobei `C` ein generischer Typ und `T1`, …, `TN` eine konkrete Parametrisierung des generischen Typs ist.
  - Die Typen `T1`, …, `TN` werden als aktuelle Typparameter oder Typargumente bezeichnet.
- Ein parametrisierter Typ kann fast wie jeder andere Typ verwendet werden.
  - Variablendeklaration
  - Methodenparameter
  - Rückgabeparameter
  - …
  - Einschränkungen werden noch erklärt.
- Beispiel:

```
Stack<Integer> integerStack;
Pair<Double, Double> doublePair;
```

Parametrisierter Typ (2)

- Für die Objekterzeugung mit `new` bei einem generischen Typ müssen nach dem Klassenname die Typargumente oder der Diamant <> angegeben werden.
- Der Diamant-Operator erlaubt eine kompaktere Schreibweise.
  - Die Typargumente der Deklaration müssen bei der Erzeugung nicht wiederholt werden.
  - Der Compiler wählt den korrekten Typ (durch Type-Inference).
- Beispiel (Pair):

```
Pair<Integer, Integer> pair1 = new Pair<Integer, Integer>(1, 2);
Pair<Integer, Integer> pair2 = new Pair<>(1, 2);  // Diamond-Operator
```

Parametrisierter Typ (3)

- `Pair<Integer, Integer>` und `Pair<Double, Double>` sind unterschiedliche (inkompatible) Typen.
  - Variablen dieser Typen können einander nicht zugewiesen werden.
- Beim Übersetzen überprüft der Compiler korrekte Zuweisungen, Verwendungen etc.
  - Statische Typprüfung durch den Compiler verhindert Übersetzung von Programmen mit Typfehlern.

```
public final class PairApplication {
    public static void main(String[] args) {
        Pair<Integer, Integer> integerPair = new Pair<>(1, 2);
        Pair<Double, Double> doublePair = integerPair; // Compile-Error!
    }
}
```

Ausgabe des Compilers:

```
java: incompatible types: Pair<Integer, Integer> cannot be converted to Pair<Double, Double>
```

Parametrisierter Typ (4)

- Methoden mit einem formalen Typparameter als Rückgabetyp liefern ein Ergebnis des entsprechenden aktuellen Typparameters.

```
public final class NoCastApplication {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>("EINS", 5);
        String v1 = pair.getFirst();
        int v2 = pair.getSecond();
    }
}
```

Typebounds (1)

- Durch Typebounds können die aktuellen Typparameter, welche für eine Typvariable eingesetzt werden können, eingeschränkt werden.
- Wird kein Typebound angegeben, können beliebige Referenztypen als Typargument eingesetzt werden.
- Ein Typebound wird nach der Typvariable mit dem Schlüsselwort `extends` angegeben.
  - Das Schlüsselwort `extends` wird bei Typebounds für Klassen und Interfaces verwendet.
- Beispiel:

```
public class ClassA<T extends Class1> {…}
```

  - Für die Parametrisierung der Klasse `ClassA` dürfen nur Subtypen von `Class1` eingesetzt werden.

```
public class ClassB<T extends Interface1> {…}
```

  - Für die Parametrisierung der Klasse `ClassA` dürfen nur Subtypen von `Interface1` eingesetzt werden.

Typebounds (2)

- Die Typvariable kann mit maximal einer Klasse und beliebig vielen Interfaces eingeschränkt werden
  - Der erste Typ kann ein beliebiger Referenztyp sein.
  - Alle weiteren Typen müssen Interfaces sein.
- Für das Verknüpfen von Typebounds wird das &-Zeichen verwendet.
- Beispiele:

```
public class ClassC<T extends Class1 & Interface1> {…}
```

  - Für die Parametrisierung der Klasse `ClassC` dürfen nur Typen eingesetzt werden, welche Subtypen von `Class1` und `Interface1` sind.

```
public class ClassD<T extends Interface1 & Interface2> {…}
```

  - Für die Parametrisierung der Klasse `ClassD` dürfen nur Typen eingesetzt werden, welche Subtypen von `Interface1` und `Interface2` sind.

Typebounds (3)

- Typargumente müssen zum Typebound kompatibel sein.
- Beispiel:

```
public class A { }
```

```
public class B extends A { }
```

```
public class C extends B { }
```

```
public class Stack<T extends B> {…}
```

```
public class TypeBoundsApplication {
    public static void main(String[] args) {
        Stack<C> cStack = new Stack<C>();
        Stack<B> bStack = new Stack<B>();
        Stack<A> aStack = new Stack<A>(); // Compile-Error!
    }
}
```

Ausgabe des Compilers:

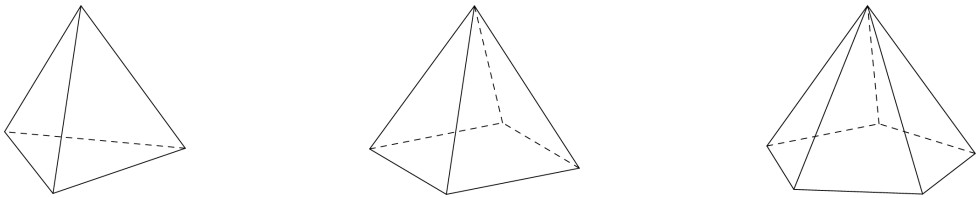
```
java: type argument A is not within bounds of type-variable T
```

Beispiel Regelmäßige Pyramide (1)

```
public class RegularPyramid<T> {
    private final T base;
    private final double height;

    public RegularPyramid(T base, double height) {
        this.base = base;
        this.height = height;
    }
}
```

- Die Klasse soll um Methoden zum Ermitteln der Anzahl an Ecken und Kanten erweitert werden.
  - Die Anzahl der Ecken und Kanten hängt von der gegeben Grundfläche ab.
- Bei der Parametrisierung kann für die Typvariable jedes Objekt eingesetzt werden.
  - Problem: Die Anzahl an Seiten des Objekts muss ermittelt werden können.
    - Ohne Typebound können nur die Methoden von Object verwendet werden.



Beispiel Regelmäßige Pyramide (2)

```java
public interface RegularPolygon {
    int getNumberOfSides();
    …
}
```

```java
public abstract class AbstractRegularPolygon implements RegularPolygon {
    private final double sideLength;

    protected AbstractRegularPolygon(double sideLength) {
        this.sideLength = sideLength;
    }
    …
}
```

```java
public final class Square extends AbstractRegularPolygon {
    public Square(double sideLength) {
        super(sideLength);
    }
    @Override
    public int getNumberOfSides() {
        return 4;
    }
}
```

Beispiel Regelmäßige Pyramide (3)

```
public class RegularPyramid<T extends RegularPolygon> {
    private final T base;
    private final double height;

    public RegularPyramid(T base, double height) {
        this.base = base;
        this.height = height;
    }

    public int getNumberOfEdges() {
        return 2 * base.getNumberOfSides();
    }

    public int getNumberOfVertices() {
        return base.getNumberOfSides() + 1;
    }
}
```

src/at/ac/uibk/pm/generics/pyramid/RegularPyramid.java

Typebounds mit Typvariablen

- Eine Typvariable kann auch bei der Formulierung von Typebounds verwendet werden.
- Dies wird als rekursiver Typebound bezeichnet.
- Rekursive Typebounds werden häufig in Kombination mit dem Interface `Comparable<T>` eingesetzt.
- Beispiel mit Interface `Comparable<T>`
  - Alle Elemente im `OrderedPair` sollten vergleichbar sein, d.h. das Interface `Comparable<T>` implementieren.
  - `T extends Comparable<T>` kann wie folgt gelesen werden: „Jeder Typ T, der mit sich selbst verglichen werden kann.“

```
public class OrderedPair<T extends Comparable<T>> {…}
```

```
OrderedPair<Integer> integerPair = new OrderedPair<>();
OrderedPair<Object> objectPair = new OrderedPair<>(); // Compile-Error!
```

Ausgabe des Compilers:

```
java: type argument Object is not within bounds of type-variable T
```

Generics und Vererbung (1)

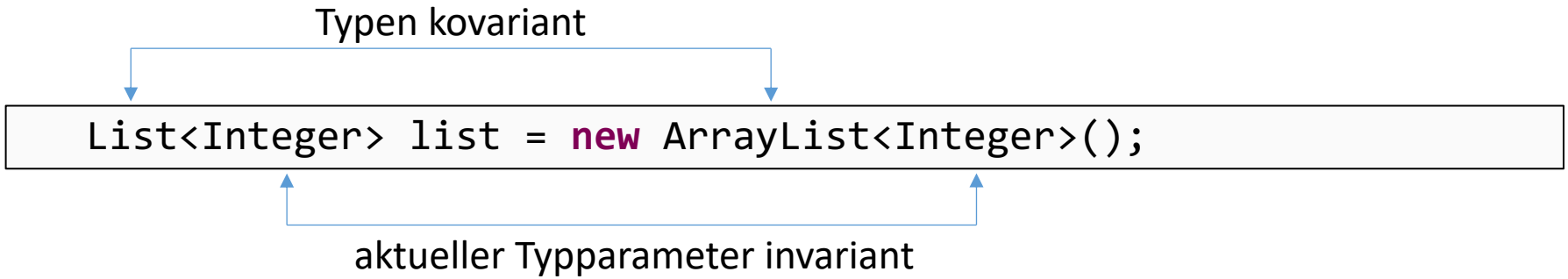
- Subtyping erstreckt sich nicht über Parametrisierung von generischen Typen.

```
ArrayList<Object> objectList = new ArrayList<String>(); // Compile-Error!
```

Ausgabe des Compilers:

```
java: incompatible types: ArrayList<String> cannot be converted to ArrayList<Object>
```

- Ein parametrisierter Typ P2 ist ein Subtyp von einem mit Referenztypen parametrisierten Typ P1, wenn
  - die Typargumente gleich sind
  - und der Rawtype (generische Klasse ohne Parametrisierung) von P2 ein Subtyp des Rawtypes von P1 ist.



Generics und Vererbung (2)

- Generische Klassen werden bei der Vererbung wie gewöhnliche Klassen behandelt (gilt auch für generische Interfaces).
- Eine generische Klasse darf erben:
  - von einem nicht generischen Typ

```
public class SubClass1<T> extends SuperClass {…}
```

  - von einem generischen Typ mit konkreter Parametrisierung

```
public class SubClass2<T> extends GenericSuperClass<Double> {…}
```

  - von einem generischen Typ mit gekoppelter Typvariable

```
public class SubClass3<T> extends GenericSuperClass<T> {…}
```

Generics und Vererbung (3)

- Überschreiben bei generischer Klasse mit konkreter Parametrisierung
  - Es werden die Typvariablen durch den konkreten Typen ersetzt

```
public class IntegerList implements List<Integer> {
    …
    @Override
    public boolean add(Integer x) {…}
}
```

- Überschreiben bei generischer Klasse mit gekoppelter Typvariable
  - Es bleiben die Typvariablen erhalten

```
public class GenericList<T> implements List<T> {
    …
    @Override
    public boolean add(T x) {…}
}
```

Beispiele - Zuweisungskompatibilität

```
public class SubClass1<T> extends SuperClass {…}
```

```
public class SubClass2<T> extends GenericSuperClass<Double> {…}
```

```
public class SubClass3<T> extends GenericSuperClass<T> {…}
```

```
SuperClass a1 = new SubClass1<String>();
```

Beliebige Parametrisierung

```
GenericSuperClass<Double> a2 = new SubClass2<String>();
```

Verpflichtend Double

Beliebige Parametrisierung

```
GenericSuperClass<Integer> a3 = new SubClass3<Integer>();
```

Gekoppelte Typargumente

Beispiel Invarianz Generics

```
public static void append(List<Vehicle> source, List<Vehicle> target) {
    for (Vehicle element : source) {
        target.add(source);
    }
}


public static void main(String[] args) {
    List<Vehicle> vehicles = new ArrayList<>();
    List<Truck> trucks = new ArrayList<>();
    …
    copy(trucks, vehicles); // Compile-Error!
}
```

Ausgabe des Compilers:

```
java: incompatible types: List<Truck> cannot be converted to List<Vehicle>
```

Wildcards (1)

- Typargumente können Referenztypen oder Wildcards sein.
- Die Wildcard ? steht für jeden beliebigen Referenztyp.

```
List<?> list;
list = new ArrayList<Integer>();
list = new ArrayList<Double>();
list = new ArrayList<String>();
```

- Für die Erzeugung eines Exemplars einer generischen Klasse können Wildcards nicht verwendet werden.

```
List<?> list = new ArrayList<?>();
```

Ausgabe des Compilers:

```
java: unexpected type
  required: class or interface without bounds
  found:    ?
```

Wildcards (2)

```
public void static printList(List<Object> list) {
    for (Object element : list) {
       System.out.println(element);
    }
}
```

Kann nur mit Listen, welche als Typargument `Object` aufweisen, aufgerufen werden.

```
public void static printList(List<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

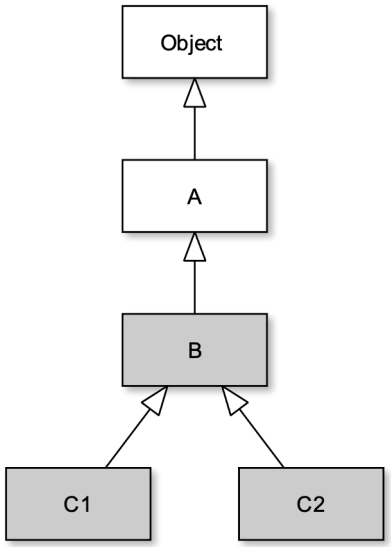
Listen mit beliebigen Typargumenten können übergeben werden.

Typebounds bei Wildcards

- Wildcards können auch eingeschränkt werden.
- `? extends UpperTypeBound`
  - Upper-Typebound
  - Alle Subtypen (inklusive des Typebounds) können für die Parametrisierung verwendet werden.
  - Einsetzen entlang der Vererbungshierarchie (Kovarianter Wildcardtyp)
  - `C<X>` ist kompatibel zu `C<? extends Y>`, wenn `X` zu `Y` kompatibel ist.
- `? super LowerTypeBound`
  - Lower-Typebound
  - Alle Supertypen (inklusive des Typebounds) können für die Parametrisierung verwendet werden.
  - Einsetzen entgegen der Vererbungshierarchie (Kontravarianter Wildcardtyp)
  - `C<X>` ist kompatibel zu `C<? super Y>`, wenn `Y` zu `X` kompatibel ist
- Die Wildcard `? extends Object` ist äquivalent zur unbounded Wildcard `?`.

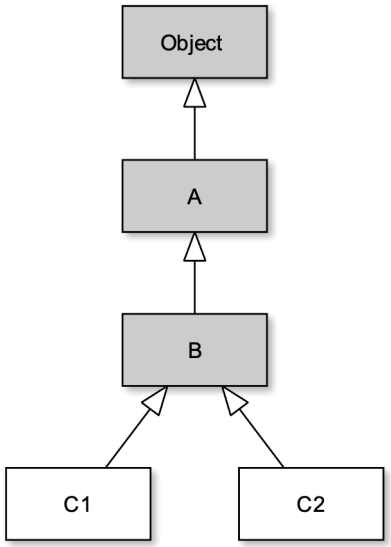
Beispiel Upper-Typebound

```
List<? extends B> list;
list = new ArrayList<Object>(); // Compile-Error!
list = new ArrayList<A>();      // Compile-Error!
list = new ArrayList<B>();
list = new ArrayList<C1>();
list = new ArrayList<C2>();
```



Beispiel Lower-Typebound

```
List<? super B> list;
list = new ArrayList<Object>();
list = new ArrayList<A>();
list = new ArrayList<B>();
list = new ArrayList<C1>();     // Compile-Error!
list = new ArrayList<C2>();     // Compile-Error!
```



Einsatz von Typebounds mit Wildcards

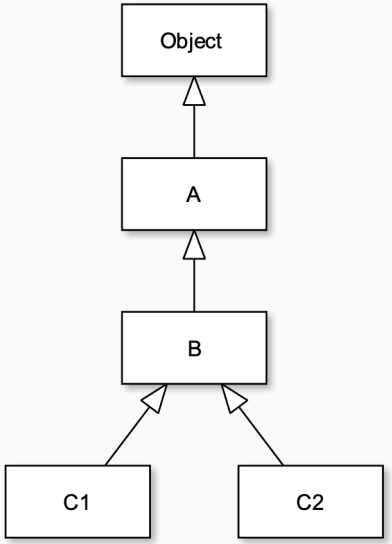
- Die Verwendung von Wildcardtypen bei Parametern erhöht die Flexibilität von Methoden.
- Merkhilfe:
  - **PECS** → **p**roducer-**e**`xtends`, **c**onsumer-**s**`uper`
    - Soll von einem parametrisierten Typ ein `T` produziert werden, soll `? extends T` verwendet werden.
    - Soll von einem parametrisierter Typ ein `T` konsumiert werden, soll `? super T` verwendet werden.
- Upper- und Lower-Typebounds sollten niemals für Rückgabetypen verwendet werden!
  - Beim Aufruf muss der Wildcardtyp in den Anwendungscode übernommen werden!
- Sofern konsumiert und produziert werden soll, kann keine Wildcard verwendet werden.

Beispiele Producer - extends

```
List<B> bList = new ArrayList<>(Arrays.asList(new B(), new B()));
List<? extends B> listExtends = bList;


listExtends.add(new Object());          // Compile-Error!
listExtends.add(new A());               // Compile-Error!
listExtends.add(new B());               // Compile-Error!
listExtends.add(new C1());              // Compile-Error!
listExtends.add(new C2());              // Compile-Error!

// Producer - extends
Object object = listExtends.get(0);
A a = listExtends.get(0);
B b = listExtends.get(0);
C1 c1 = listExtends.get(0);             // Compile-Error!
C2 c2 = listExtends.get(0);             // Compile-Error!
```



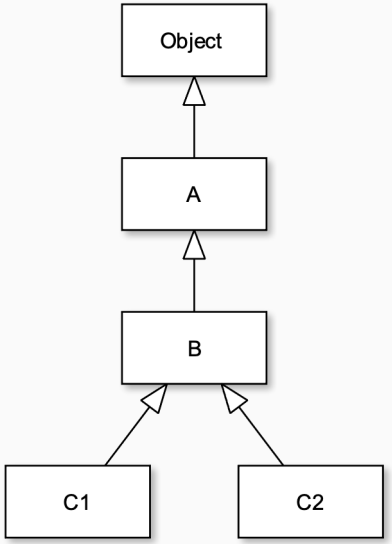
Beispiel Consumer - super

```
List<B> bList = new ArrayList<>(Arrays.asList(new B(), new B()));
List<? super B> listSuper = bList;


// Consumer = super
listSuper.add(new Object());          // Compile-Error!
listSuper.add(new A());               // Compile-Error!
listSuper.add(new B());
listSuper.add(new C1());
listSuper.add(new C2());



Object object = listSuper.get(0);
A a = listSuper.get(0);               // Compile-Error!
B b = listSuper.get(0);               // Compile-Error!
C1 c1 = listSuper.get(0);             // Compile-Error!
C2 c2 = listSuper.get(0);             // Compile-Error!
```



Beispiel Stack

```
public void pushAll(Collection<? extends T> elements) {
    for (T element : elements) {
        push(element);
    }
}

public void popAll(Collection<? super T> elements) {
    while (!isEmpty()) {
        elements.add(pop());
    }
}
```

produziert Elemente vom Typ T

konsumiert Elemente vom Typ T

```
public static void main(String[] args) {
    Stack<Number> arrayStack = new ArrayStack<>();
    arrayStack.pushAll(Arrays.asList(0.0, 1.0, 2.0));
    arrayStack.pushAll(Arrays.asList(3, 4, 5));

    Collection<Number> poppedElements = new ArrayList<>();
    arrayStack.popAll(poppedElements);
    System.out.println(poppedElements);
}
```

src/at/ac/uibk/pm/generics/stack/StackApplication.java

Beispiel Wildcardtyp

```
public static void append(List<? extends Vehicle> source,
                          List<? super Vehicle> target) {
    for (Vehicle element : source) {
        target.add(element);
    }
}


public static void main(String[] args) {
    List<Vehicle> vehicles = new ArrayList<>();
    List<Truck> trucks = new ArrayList<>();
    …
    append(trucks, vehicles);
    …
    List<Truck> trucksNew = new ArrayList<>();
    …
    append(trucks, trucksNew); // Compile-Error!
}
```

Ausgabe des Compilers:

```
java: incompatible types: List<Truck> cannot be converted to List<Vehicle>
```

Polymorphe Methoden

Polymorphe Methoden

- Generische Methoden werden als polymorphe Methoden bezeichnet.
- Sie sind unabhängig von generischen Typen.
  - Können in nicht-generischen Klassen definiert und aufgerufen werden.
- Objektmethoden, statische Methoden und Konstruktoren können polymorph sein.
- Die formalen Typparameter werden vor dem Rückgabetyp der Methode bzw. vor dem Klassennamen des Konstruktors angegeben.
- Die formalen Typparameter beziehen sich nur auf die polymorphe Methode.

Beispiel polymorphe Methode

```
public final class PolymorphicMethods {

    public static <T> T determineMajority(T x, T y, T z) {
        if (x.equals(y) || x.equals(z)) {
            return x;
        }
        if (y.equals(z)) {
            return y;
        }
        return null;
    }

    public static void main(String[] args) {
        String decision = determineMajority("yes", "yes", "no");
        System.out.println(decision);

        Integer option = determineMajority(1, 1, 2);
        System.out.println(option);
    }
}
```

Typebounds

- Auch bei polymorphen Methoden können Typebounds verwendet werden.
- Beispiel: Methode zum Berechnen des Medians von drei Werten.

```
public <T extends Comparable<? super T>> T median(T x, T y, T z) {
    …
}
```

  - Erlaubter Aufruf auf dem Objekt `instance`:

    ```
    Integer integerMedian = instance.<Integer>median(3, 2, 5);
    ```

    - Integer implementiert das Interface `Comparable`.
    - Das Typargument kann durch Typinferenz ermittelt werden und muss in diesem Fall nicht angegeben werden.
  - Nicht erlaubt:

    ```
    Object objectMedian = instance.<Object>median(3, 2, 5);
    ```

    - `Object` implementiert das Interface `Comparable` nicht.

Beispiel polymorphe Methode mit Typebounds

```
public static <T extends Vehicle> void append(List<T> source,
                                              List<? super T> target) {
    for (T element : source) {
        target.add(element);
    }
}


public static void main(String[] args) {
    final List<Vehicle> vehicles = new ArrayList<>();
    final List<Truck> trucks = new ArrayList<>();
    …
    append(trucks, vehicles);
    …
    List<Truck> trucksNew = new ArrayList<>();
    …
    append(trucks, trucksNew);
}
```

Type Erasure

Homogene bzw. Heterogene Übersetzung

- Es gibt zwei Realisierungsmöglichkeiten für generische Datentypen.
- Heterogene Variante
  - Für jeden Typ welcher als Typparameter zum Einsatz kommt, wird individueller Code erzeugt.
    - Werden beispielsweise die Typen `String`, `Integer` und `Pair` als Typparameter für eine Klasse verwendet, werden drei Klassen für diese Typen erzeugt.
  - Wird beispielsweise bei C++ verwendet.
- Homogene Übersetzung
  - Für jede generische Klasse wird nur eine Klasse erzeugt, die anstelle des generischen Typs eine allgemeine Klasse (wie z.B. `Object`) verwendet.
  - Für eine konkrete Parametrisierung werden entsprechende Casts eingefügt.
- Java benutzt die **homogene** Übersetzung.

Type-Erasure (1)

- Der Mechanismus hinter der Übersetzung generischer Klassen etc. wird als *Type-Erasure* bezeichnet.
- Generics werden nur vom Compiler verarbeitet.
- Laufzeitsystem und JVM sehen nichts davon!

Type-Erasure (2)

- Auflösen von Generics durch den Compiler:
  - Typvariablen mit Typebounds werden durch den einzigen bzw. erstgenannten Typebound ersetzt.
  - Typvariablen ohne Typebound werden mit `Object` ersetzt.
  - Spitzklammern und Typparameter werden gelöscht.
  - Bei Bedarf:
    - Einfügen von Casts, um Typsicherheit zu wahren.
    - Generieren von Brückenmethoden für Typen, die generische Typen erweitern.

Typparameter-Sektion	**Type-Erasure**
`<T>`	`Object`
`<T extends Number>`	`Number`
`<T extends Comparable<? super T>>`	`Comparable`
`<T extends Number & Cloneable>`	`Number`

Beispiel generische Klasse

```
public final class Pair<T, U> {
    private final T first;
    private final U second;
    public Pair(T first, U second) {…}
    …
}
```

```
public final class Pair {
    private final Object first;
    private final Object second;
    public Pair(Object first, Object second) {…}
    …
}
```

Type-Erasure bei parametrisierten Typen

- Zuerst statische Typprüfung:
  - Typargumente müssen, falls angegeben, allen Typebounds genügen.
  - Die Regeln für Subtyping müssen bei parametrisierten Typen eingehalten werden.
- Danach Type-Erasure:
  - Typargumente, inklusive Wildcards und spitzen Klammern werden gelöscht.
  - Typecasts werden eingeschoben, wo ein Wert eines Typparameters benutzt wird:

```
Pair<Integer, Integer> pair = new Pair<>(1, 1);
Integer value = pair.getFirst();
```

```
Pair pair = new Pair(1, 1);
Integer value = (Integer) pair.getFirst();
```

Type-Erasure (Zusammenfassung)

- Alle parametrisierten Typen einer generischen Klasse, eines generischen Interfaces bzw. einer polymorphen Methode teilen sich eine einzige Implementierung.
- Zur Laufzeit existieren **keine**
  - Typvariablen
  - Typparameter
  - Typebounds
  - Wildcards

Rawtypes

- Eine durch Type-Erasure reduzierte Definition liefert den sogenannten **Rawtype** des generischen Typs.
  - Es gibt genau einen Rawtype zu einer generischen Klasse.
  - Ist eine normale, nicht-generische Klasse.
  - Kann verwendet werden, z.B.

```
Pair p = new Pair(10, 10);
```

- Vorteil
  - Alte Applikationen (vor Java 5) können mit generischen Klassen zusammenarbeiten.
- Nachteil
  - Einführung von Non-Reifiable Types

Reifiable und Non-Reifiable Types

- Reifiable Types
  - Typen, für die zur Laufzeit die vollständigen Typinformationen vorhanden sind.
  - Type-Erasure hat auf diese Typen keinen Einfluss.
  - Beispiele:
    - Primitive Datentypen
    - Nicht generische Klassen oder Interfaces
    - Rawtypes

- Non-Reifiable Types
  - Typen für die, aufgrund von Type-Erasure, zur Laufzeit nicht alle Typinformationen vorhanden sind.
  - Beispiele:
    - Parametrisierung eines generischen Typs mit mindestens einem Typargument
    - Parametrisierung eines generischen Typs mit mindesten einer gebundenen Wildcard

Grenzen von Generics in Java (1)

- Kein Erzeugen von Arrays aus Non-Reifiable Types möglich.
- Keine dynamische Typprüfung für Non-Reifiable Types möglich.

```
x instanceof List<Integer>
```

- Keine dynamische Typprüfung der Typvariable.
  - Der Typparameter kann nicht mit dem Operator `instanceof` überprüft werden.

    ```
    if (x instanceof T) …
    ```
  - Der Compiler kann beim Übersetzen noch nicht den Typ kennen.
  - Type-Erasure würde die Bedingung auf den sinnlosen Test

    ```
    x instanceof Object
    ```

    reduzieren.
- Kein direktes Erzeugen von Arrays mit dem formalen Typparameter möglich.

Grenzen von Generics in Java (2)

- Primitive Datentypen können nicht als Typparameter verwendet werden.

```
Pair<int, int> intPair = new Pair<>(1, 4); // Compile-Error!
```

Ausgabe des Compilers:

```
java: unexpected type
  required: reference
  found:    int
```

Grenzen von Generics in Java (3)

- Statische Variablen und statische Methoden einer generischen Klasse können keine Typvariablen benutzen.
  - Formale Typparameter gehören zu den Exemplaren und nicht zur Klasse.
- Generische Klassen können nicht direkt oder indirekt die Klasse `Throwable` erweitern.
- Methoden, die nach Type-Erasure die gleiche Signatur haben, können nicht überladen werden.

```
public static void print(Pair<Short, Short> pair) {…} // Compile-Error!
public static void print(Pair<String, String> pair) {…}
```

Ausgabe des Compilers:

```
java: name clash: print(Pair<String,String>) and print(Pair<Short,Short>) have the same erasure
```

Grenzen von Generics in Java (4)

- Konstruktoren:
  - Wegen der fehlenden Information über spätere Typparameter können in einer generischen Klasse keine Konstruktoren von Typvariablen aufgerufen werden (T kann auch nicht als Basisklasse verwendet werden).

```
public class Store<T> {

    private T info;

    public Store() {
        info = new T(); // Compile-Error!
    }

    …
}
```

Ausgabe des Compilers:

```
java: unexpected type
  required: class
  found:    type parameter T
```

Brückenmethoden

- Spezielle Situationen müssen vom Compiler selbst aufgelöst werden.
- Programmierende müssen sich keine Gedanken machen.
- Ein besonderes Beispiel sind Brückenmethoden.
- Brückenmethoden werden eingefügt, wenn eine Klasse eine generische Klasse bzw. ein generisches Interface erweitert bzw. implementiert und durch Type-Erasure eine Signatur einer vererbten Methode verändert wird.

Beispiel (Original)

```
public class X<E> {
    private Integer member = 5;

    public Integer get(E param) {
        return member;
    }
}
```

```
public class Y extends X<String> {
    @Override
    public Integer get(String param) {
        return Integer.valueOf(10);
    }
}
```

```
X<String> a = new Y();
a.get("Hello");
```

Was passiert bei der Übersetzung?
Hat die Methode `get` nach der Übersetzung (und Type-Erasure) die gleiche Signatur?
Anscheinend nicht.

Wird hier die Methode noch überschrieben?
Ja! → Brückenmethode

Beispiel (was der Compiler macht)

```
public class X {

    private Integer member = 5;

    public X() {}

    public Integer get(Object param) {
       return member;
    }
}
```

Der Code sieht in etwa so aus
(im Bytecode, nach Decompilierung)

```
public class Y extends X {

    public Y() {}

    public Integer get(String param) {
        return Integer.valueOf(10);
    }

    public Integer get(Object param) {
        return get((String) param);
  }
}
```

Brückenmethode

Brückenmethode im vorherigen Beispiel

- Die Brückenmethode (2. get-Methode) wurde vom Compiler eingefügt.
- Sie überschreibt die Methode der Klasse X und ruft jetzt die theoretisch überschreibende Methode der Klasse Y auf.
- Die Klasse Y verhält sich so, als wäre die Methode überschrieben worden.
- Beim Programmieren ist dies nicht direkt zu erkennen.
  - Nur im Bytecode ersichtlich!
- Die Brückenmethoden werden auch bei Methoden verwendet, die sich nach der Type-Erasure nur im Rückgabewert unterscheiden.

Zusammenfassung Genercis-Begriffe

Begriff	Beispiel
Generischer Typ	`List<E>`
Typvariable oder formaler Typparameter	`E`
Gebundener Typparameter	`E extends Number`
Raw-Type	`List`
Parametrisierter Typ	`List<Integer>`
Typargument oder aktueller Typparameter	`Integer`
Ungebundener Wildcardtyp	`List<?>`
Gebundener Wildcardtyp	`List<? extends Number>`
Wildcard	`?`
Upper-Typebound	`? extends Number`
Lower-Typebound	`? super Number`

Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (***Java SE 17 Edition***)**, Oracle, 2021
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018