Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 08

Bounded Types

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter → Map → Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

TECHNISCHE HOCHSCHULE LÜBECK

Unit 08

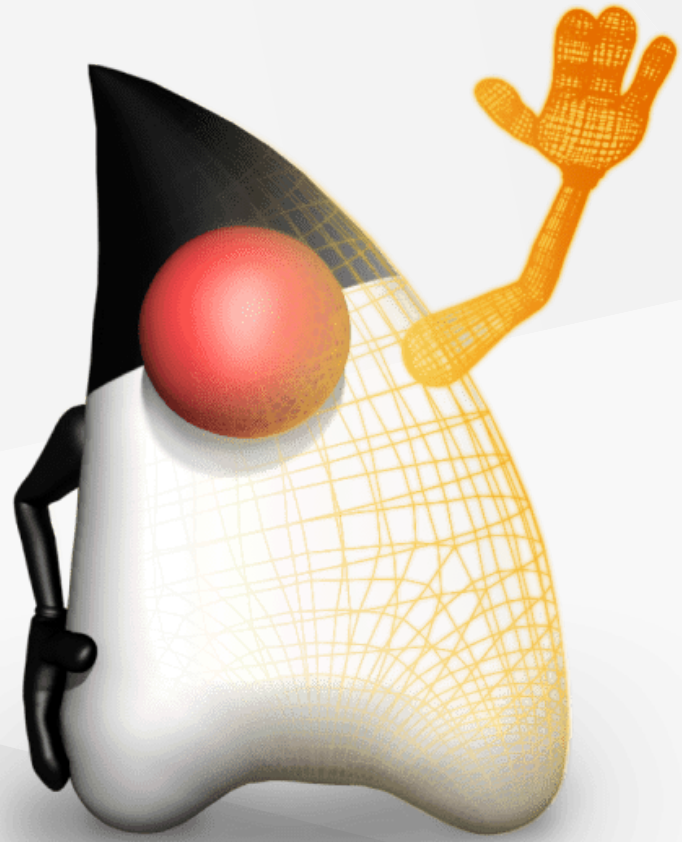
Generische Datentypen - Bounded Types

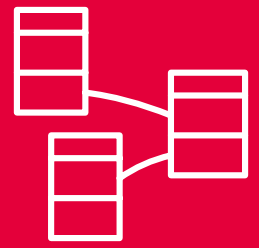
Inhalte:

- Bounded Types
- Type Erasure mit Bounded Types
- Bounded Types mit mehreren Schnittstellen

Übungsaufgaben

- Generische Bestimmung eines größten Elements
  - klassisch
  - rekursiv
  - Lambdas + Streams
- Generische Implementierung von Quicksort





class<T> → class<T extends Type>

Wenn generisch zu generisch ist ...

Das Problem mit dem Type Erasure

Der **Type Erasure** ist ein zentraler Mechanismus, der dafür sorgt, dass alle Informationen über generische Typen zur Compile-Zeit entfernt werden. Dies bedeutet, dass generische Typen zur Laufzeit nicht mehr als solche existieren und stattdessen durch den allgemeinsten Referenztyp `Object` ersetzt werden.

Dadurch können allerdings auch keine Operatoren oder Methoden auf diese Objekte mehr angewendet werden, die der generischsten aller Klassen in Java `Object` nicht bekannt sind (siehe Tabelle).

Die Fähigkeiten (Methoden) der `Object`-Klasse sind sehr eingeschränkt, wie diese Aufstellung zeigt. Bis auf `equals()`, `toString()` sowie ggf. `finalize()` kann über Polymorphie keinerlei Typspezifische Logik in generischen Klassen angeprochen werden.

Einzig über `getClass()` *könnte* über Class Introspection (programatisch **aufwändig**) spezifische Logik aufgerufen werden.

Methode	Rückgabetyp	Beschreibung
`equals()`	`boolean`	Vergleich des Inhalts.
`hashCode()`	`int`	Gibt einen hash-Code für das Objekt zurück.
`toString()`	`String`	Gibt eine String-Darstellung des Objekts zurück.
`getClass()`	`Class<?>`	Gibt die Klasse des Objekts zurück.
`finalize()`	`void`	Wird aufgerufen, bevor Objekt gelöscht wird.
`notifyAll()`	`void`	Weckt alle Threads, die auf diesem Objekt warten.
`wait()`	`void`	Lässt den aktuellen Thread warten.

Methoden der Klasse `Object`

In welchen Situationen kann dies problematisch werden?

Angenommen wir wollten folgende Klasse erweitern,

```java
public class Values<T> {
  private List<T> values = new ArrayList<>();

  public Values(T... vs) {
    for (T v : vs) values.add(v);
  }

  public void add(T e) {  this.values.add(e); }

  public T get(int i) {
    return this.values.get(i);
  }


  [...]

}
```

z. B., um das kleinste Elemente zu bestimmen:

```java
public T min() {
  T m = this.values.get(0);
  for (T e : this.values) { m = e < m ? e : m; }
  return m
}
```

Das Problem hier ist allerdings, dass alle Elemente der Liste nach dem Type Erasure vom Typ `Object` sind.

Der `<`-Operator ist allerdings nicht auf `Object` definiert.

Der Type Erasure mit `Object` ist also nur dann unproblematisch, wenn die Generische Klasse in ihrer Logik nichts mit dem Objekt an sich macht (wie das bspw. bei Collections der Fall ist).

Im hier gezeigten Fall müssten wir von allen Elementen die Eigenschaft verlangen, dass diese vergleichbar sind (`Comparable`).

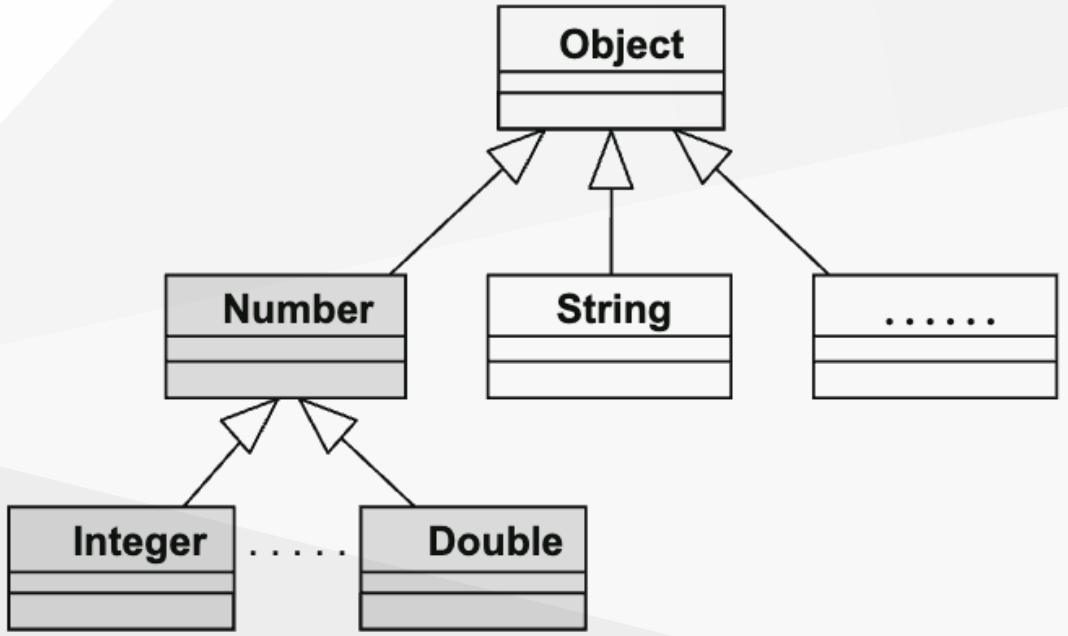
Was sind Bounded Types?

In Fällen wie dem gezeigten, kann man auf sogenannte Bounded Types zurückgreifen (und die Generizität damit einschränken).

Bounded Types beziehen sich auf den Mechanismus, mit dem mögliche Typen für generische Klassen und Methoden eingeschränkt werden. Dies ermöglicht eine genauere Typprüfung zur Kompilierzeit und trägt dazu bei, die **Typsicherheit** zu erhöhen.

- Ein Bounded Type-Parameter wird durch die Syntax `T extends Base` definiert, wobei `Base` eine Klasse oder ein Interface ist.
- Dies bedeutet, dass der Typ `T` auf `Base` oder einen Subtyp von `Base` beschränkt ist.
- Diese Einschränkung hilft, die Verwendung einer generischen Klasse oder Methode auf einen Teilbaum einer Klassenhierarchie zu fokussieren.

Beispiel: Selektion eines Teilbaums einer Klassenhierarchie durch Angabe von `T extends Number`.



Bildquelle: Java als erste Programmiersprache, Springer, 2016

Type Erasure bei Bounded Types

Generische Klassendefinition:

```
public class Values<T extends Comparable<T>> {
  private List<T> values = new ArrayList<>();

  public Values(T... vs) {
    for (T v : vs) values.add(v);
  }

  public void add(T e) {  this.values.add(e); }

  public T get(int i) { return values.get(i); }

  public T min() {
    T m = this.values.get(0);
    for (T e : this.values) {
      m = e.compareTo(m) < 0 ? e : m;
    }
    return m;
  }
}
```

Nach Type Erasure:

```
public class Values {
  private List<Comparable<?>> values = new ArrayList<>();

  public Values(Comparable<?>... vs) {
    for (Comparable<?> v : vs) values.add(v);
  }

  public void add(Comparable<?> e) {  this.values.add(e); }

  public Comparable<?> get(int i) { return values.get(i); }

  public Comparable<?> min() {
    Comparable<?> m = this.values.get(0);
    for (Comparable<?> e : this.values) {
      m = e.compareTo(m) < 0 ? e : m;
    }
    return m;
  }
}
```

Bounded Types mit mehreren Schnittstellen

Eine Ente kann fliegen und schwimmen oder schwimmen und fliegen …

Bounded Types mit mehreren Schnittstellen

Wo wir herkommen:

In Java können wir Generics mit Bounded Types einschränken.

Wir sagen dem Compiler: *"Dieser Typ `T` darf nur ein bestimmtes Interface oder eine bestimmte Klasse sein – oder eine Unterklasse davon."*

```
class Box<T extends Number> {
  private T value;

  public Box(T value) {
    this.value = value;
  }
}
```

Was auch geht:

In Java ist es aber auch möglich, einen Typ gleichzeitig auf mehrere Interfaces einzuschränken. Das geht mit dem `&` -Operator.

Wir sagen dem Compiler damit: *" `T` muss jetzt sowohl `Comparable` als auch `Serializable` sein."*

```
class Box<T extends Comparable<T> & Serializable> {
  private T value;

  public Box(T value) {
    this.value = value;
  }
}
```

💡 **Wichtig**: Wenn eine Klasse + Interfaces eingeschränkt werden sollen, muss die Klasse (falls vorhanden) immer zuerst stehen. Mehrere Interfaces sind kein Problem, aber es darf maximal eine Klasse geben.

📌 Wofür braucht man das?

Sichere API-Designs:

Man will, dass generische Typen nur verwendet werden können, wenn sie bestimmte Fähigkeitskombinationen haben, z. B. vergleichbar (`Comparable`) und speicherbar (`Serializable`) sind.

Generische Algorithmen mit Anforderungen:

Sortieralgorithmen könnten verlangen, dass die Objekte `Comparable` sind.
Falls sie zusätzlich noch persistiert werden sollen, müssen sie auch `Serializable` sein.
Beispiel:

```
public static <T extends Comparable<T> & Serializable> void process(T item) {
  System.out.println("Item: " + item);
}
```

Frameworks und Libraries:

Viele Frameworks wie Spring oder Hibernate nutzen solche Bounded Types, um sicherzustellen, dass die Objekte alle benötigten Fähigkeiten besitzen. **Beispiel**: Ein Datenbankobjekt, das sowohl `Comparable` für die Sortierung als auch `Serializable` für die Speicherung sein muss.

🦢 Beispiel mit Tieren und Interfaces

```java
public interface Fly {
  void fly();
}

public interface Swim {
  void swim();
}
```

Angenommen wir hätten in einer Tiershow folgende Fähigkeiten

- Schwimmen
- Fliegen

und Tiere:

- Ente
- Adler
- Delphin

```java
public class Duck implements Fly, Swim {
  public void fly() {
    System.out.println("Duck is flying!");
  }

  public void swim() {
    System.out.println("Duck is swimming!");
  }
}

public class Eagle implements Fly {
  public void fly() {
    System.out.println("Eagle is high flying!");
  }
}

public class Dolphin implements Swim {
  public void swim() {
    System.out.println("Flipper is swimming gracefully!");
  }
}
```

Trainer-Klassen (Teil I)

🌊 SeaShowTrainer - Schwimmende Tiere erforderlich

Hier wird ein Bounded Type verwendet, der nur Tiere zulässt, die das Swim-Interface implementieren.

✅ Zulässig: Duck, Dolphin
❌ Nicht erlaubt: Eagle (kann nicht schwimmen)

```
class SeaShowTrainer<T extends Swim> {
  private T animal;

  public SeaShowTrainer(T animal) { this.animal = animal; }

  public void performSeaShow() {
    System.out.println("Sea show is starting!");
    animal.swim();
  }
}
```

🛩️ AirShowTrainer - Fliegende Tiere erforderlich

Hier wird ein Bounded Type verwendet, der nur Tiere zulässt, die das Fly-Interface implementieren.

✅ Zulässig: Duck, Eagle
❌ Nicht erlaubt: Dolphin (kann nicht fliegen)

```
class AirShowTrainer<T extends Fly> {
  private T animal;

  public AirShowTrainer(T animal) { this.animal = animal; }

  public void performAirShow() {
    System.out.println("Air show is starting!");
    animal.fly();
  }
}
```

Trainer-Klassen (Teil II)

🌍 MegaShowTrainer - Tiere erforderlich, die fliegen UND schwimmen können

Hier kommt der spannende Teil: Bounded Types mit mehreren Interfaces!

Wir verwenden daher ein Bounded Type, der nur Tiere zulässt, die das Fly-Interface und das Swim-Interface implementieren.

✅ Zulässig: Duck (kann fliegen & schwimmen)
❌ Nicht erlaubt: Eagle (kann nur fliegen), Dolphin (kann nur schwimmen)

```
class MegaShowTrainer<T extends Fly & Swim> {
  private T animal;

  public MegaShowTrainer(T animal) { this.animal = animal; }

  public void performMegaShow() {
    System.out.println("Mega show is starting!");
    animal.fly();
    animal.swim();
  }
}
```

💡 Warum ist das nützlich?

- Man kann den Compiler zwingen, die Anforderungen direkt zu überprüfen.
- Es gibt keine Laufzeitfehler – wenn die Bounded Types nicht passen, wird der Code gar nicht kompiliert.
- Dadurch entstehen flexible, aber typsichere APIs, die für verschiedene Tiere (oder Klassen allgemein) funktionieren.

Übungsaufgaben

Größtes Element • Quicksort

1. Aufgabe: Größtes Element

Entwickeln Sie **eine** Methode `biggest()`, die aus einer Liste von Werten beliebigen Typs den größten Wert bestimmt.
Implementieren Sie dabei `biggest()` jeweils einmal:

- "klassisch"
- rekursiv
- und als Streams mit Lambda-Funktionen.

Vorgabe:

```
int max = biggest(List.of(1,2,3,4,5,6,7,8,9,0));
System.out.println(max); // => 9

String tooLong = biggest(List.of("Welt", "lass", "Dich", "umarmen"));
System.out.println(tooLong); // => umarmen

Character nearZ = biggest(new ArrayList<Character>());
System.out.println(nearZ); // => null
```

Frag KIRA: Wie funktioniert `min()` und `max()` auf Streams?
Frag KIRA: Was bitte soll ein `Optional` sein? Welche Methoden bietet `Optional` an?

2. Aufgabe: Quicksort

Recherchieren Sie wie Quicksort funktioniert und implementieren Sie das Verfahren für beliebige Datentypen in Java.

Vorgabe:

```
List<Integer> values = quicksort(Arrays.asList(42, -24, 17, 38, 12, 23, 23, 89, -100));
System.out.println(values); // => [-100, -24, 12, 17, 23, 23, 38, 42, 89]

List<String> sorted = quicksort(Arrays.asList("Dies", "ist", "mal", "wieder", "nur", "unser", "Beispiel"));
System.out.println(sorted); // => [Beispiel, Dies, ist, mal, nur, unser, wieder]
```

Lösung:

>>> YOUR-TURN <<<

Frag KIRA: Hilf mir dabei Quicksort zu implementieren.

Zusammenfassung

- Einschränkung der Generizität durch Bounds
  Mit Bounded Types lassen sich generische Typen auf bestimmte Klassen oder Interfaces einschränken, um die zulässigen Typen klar zu definieren.
- Effekt auf den Type Erasure
  Beim Type Erasure wird der Bound (z. B. `T extends Number`) als Ersatztyp verwendet, wenn der generische Typ durch den Compiler entfernt wird.
- Bounded Types mit Klassen und mehreren Interfaces
  Ein Bounded Type kann maximal eine Klasse (als Oberklasse) enthalten, die immer zuerst stehen muss, gefolgt von beliebig vielen Interfaces. Mehrere Interfaces im Bound erlauben es, generische Typen mit mehreren Fähigkeiten zu kombinieren, sodass alle Methoden aus diesen Interfaces im generischen Code nutzbar sind.

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe: Größtes Element

```java
// Klassisch
public static <T extends Comparable<T>> T biggest(List<T> values) {
  if (values.isEmpty()) return null;
  T big = values.get(0);
  for (T v : values) { big = v.compareTo(big) < 0 ? big : v; }
  return big;
}

// Rekursiv
public static <T extends Comparable<T>> T biggestRecursive(List<T> values) {
  if (values.isEmpty()) return null;
  if (values.size() == 1) return values.get(0);
  T head = values.get(0);
  T rest = biggestRecursive(values.subList(1, values.size()));
  return head.compareTo(rest) < 0 ? rest : head;
}

// Streams und Funktional
public static <T extends Comparable<T>> T biggestLambda(List<T> values) {
  return values.stream().max((a, b) -> a.compareTo(b)).orElse(null);
}
```

Lösungsvorschlag zur 2. Aufgabe: Quicksort

```java
public static <T extends Comparable<T>> List<T> quicksort(List<T> list) {
  if (list.size() <= 1) return list;

  T pivot = list.get(list.size() - 1); // Wähle das Pivot-Element (z. B. letztes Element)

  // Listen zur Partitionierung
  List<T> less = new ArrayList<>();
  List<T> greater = new ArrayList<>();

  // Partitioniere die Liste
  for (int i = 0; i < list.size() - 1; i++) {
    if (list.get(i).compareTo(pivot) <= 0) less.add(list.get(i));
    else greater.add(list.get(i));
  }

  // Rekursiv sortieren und zusammenführen
  List<T> sortedList = quicksort(less);
  sortedList.add(pivot);
  sortedList.addAll(quicksort(greater));

  return sortedList;
}
```