Karlsruher Institut für Technologie

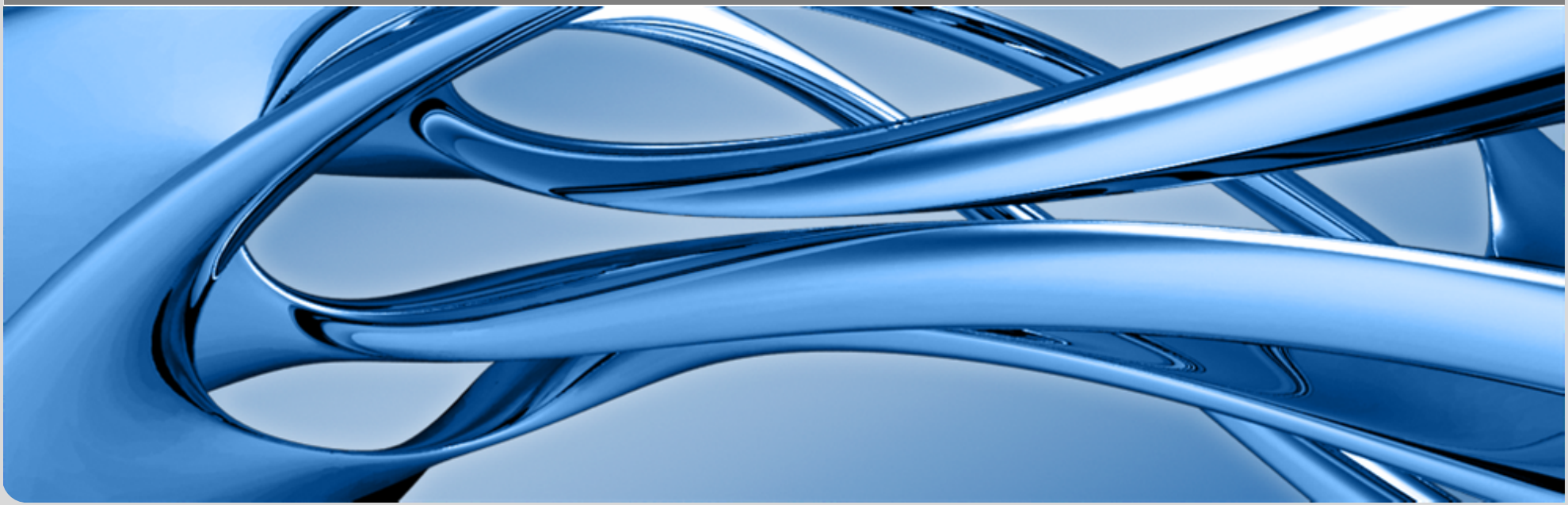
SDQ Software Design and Quality

Tutorium 8

Interfaces und Generics

Christian Zielke | 18. Dezember 2018

CHAIR FOR SOFTWARE DESIGN AND QUALITY



Gliederung

Gliederung

Allgemeines

Kurzer Hinweis

Am 8.1. findet kein Tutorium statt!

Vererbung

- **ist-ein** Beziehung zwischen Unter- und Oberklasse
- Unterklasse kann an Stelle der Oberklasse verwendet werden
- Es kann nur von einer Klasse geerbt werden
- Geerbte Methoden können überschrieben werden
- Es wird immer die in der Vererbungslinie unterste Methode ausgeführt
- Alle Klassen erben von `Object`
- Modifikatoren **final** und **abstract**

Interfaces

- Sammlung von Methodensignaturen ohne Implementierung
- Eine Klasse kann mehrere Interfaces implementieren
- Definieren Schnittstellen und Fähigkeiten

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Interfaces - Konzept

- Viele Dinge können auf unterschiedliche Art umgesetzt werden
  $\rightarrow$ Ein Fahrzeug kann ein Auto, ein Boot ovm. sein
- Es existieren grundlegende Funktionalitäten, die alle Umsetzungen enthalten müssen
- Abstrakte Klasse? Nachteile:
  - Keine Mehrfachvererbung $\rightarrow$ vergleichbar & sortierbar?
- Lösung: **Interface**
  - Sammlung von Methoden ohne Implementierung
  - Konstanten erlaubt, **keine Attribute**
  - Eine Klasse **implementiert** ein Interface und **muss** dabei alle Methoden des Interfaces implementieren
  - Interfaces können nicht selbst instanziiert werden

Syntax

- Interface-Deklaration (in eigener Datei!):
  `interface` *`InterfaceName`* `{Konstanten/Methoden}`
- Implementierung (in Klasse):
  `class` *`ClassName`* `implements InterfaceName {}`
- Mehrfachimplementierung:
  `class` *`ClassName`* `implements IName1, IName2 {}`
- Implementierungen müssen public sein!

Syntax

- Interface-Deklaration (in eigener Datei!):
  `interface` *`InterfaceName`* `{Konstanten/Methoden}`
- Implementierung (in Klasse):
  `class` *`ClassName`* `implements InterfaceName {}`
- Mehrfachimplementierung:
  `class` *`ClassName`* `implements IName1, IName2 {}`
- Implementierungen müssen public sein!

Syntax

- Interface-Deklaration (in eigener Datei!):
  `interface` *`InterfaceName`* `{Konstanten/Methoden}`
- Implementierung (in Klasse):
  `class` *`ClassName`* `implements InterfaceName {}`
- Mehrfachimplementierung:
  `class` *`ClassName`* `implements IName1, IName2 {}`
- Implementierungen müssen public sein!

Syntax

- Interface-Deklaration (in eigener Datei!):
  `interface` *`InterfaceName`* `{Konstanten/Methoden}`
- Implementierung (in Klasse):
  `class` *`ClassName`* `implements InterfaceName {}`
- Mehrfachimplementierung:
  `class` *`ClassName`* `implements IName1, IName2 {}`
- Implementierungen müssen public sein!

Beispiel

```java
interface Drivable (
boolean start();
void stop();
void accelerate(float acc);
void turn (int degree);
}
```

```java
class Car implements Drivable (
public boolean start() {...}
public void stop() {...}
public void accelerate(float acc) {...}
public void turn (int degree){...}
}
```

```java
class Boat implements Drivable {
public boolean start() {...}
public void stop() {...}
public void accelerate(float acc) {...}
public void turn (int degree){...}
}
```

Beispiel

```
1   interface Drivable (
2   boolean start();
3   void stop();
4   void accelerate(float acc);
5   void turn (int degree);
6   }
```

```
1   class Car implements Drivable (
2   public boolean start() {...}
3   public void stop() {...}
4   public void accelerate(float acc) {...}
5   public void turn (int degree){...}
6   }
```

```
1   class Boat implements Drivable {
2   public boolean start() {...}
3   public void stop() {...}
4   public void accelerate(float acc) {...}
5   public void turn (int degree){...}
6   }
```

Beispiel

```
1   interface Drivable (
2   boolean start();
3   void stop();
4   void accelerate(float acc);
5   void turn (int degree);
6   }
```

```
1   class Car implements Drivable (
2   public boolean start() {...}
3   public void stop() {...}
4   public void accelerate(float acc) {...}
5   public void turn (int degree){...}
6   }
```

```
1   class Boat implements Drivable {
2   public boolean start() {...}
3   public void stop() {...}
4   public void accelerate(float acc) {...}
5   public void turn (int degree){...}
6   }
```

Beispiel

```
1   void simulate(Drivable d) {
2   d.start();
3   d.accelerate();
4   d.stop();
5   }
```

- Beim Aufruf einer Interface-Methode wird geprüft, welches Objekt `d` gerade hält
- `simulate(new Car());`
  → Methoden von `Car` werden verwendet
- `simulate(new Boat());`
  → Methoden von `Boat` werden verwendet

Beispiel

```
1   void simulate(Drivable d) {
2   d.start();
3   d.accelerate();
4   d.stop();
5   }
```

- Beim Aufruf einer Interface-Methode wird geprüft, welches Objekt `d` gerade hält
- `simulate(new Car());`
  → Methoden von `Car` werden verwendet
- `simulate(new Boat());`
  → Methoden von `Boat` werden verwendet

Beispiel

```
1   void simulate(Drivable d) {
2   d.start();
3   d.accelerate();
4   d.stop();
5   }
```

- Beim Aufruf einer Interface-Methode wird geprüft, welches Objekt `d` gerade hält
- `simulate(new Car());`
  → Methoden von `Car` werden verwendet
- `simulate(new Boat());`
  → Methoden von `Boat` werden verwendet

Beispiel

```
1   void simulate(Drivable d) {
2   d.start();
3   d.accelerate();
4   d.stop();
5   }
```

- Beim Aufruf einer Interface-Methode wird geprüft, welches Objekt `d` gerade hält
- `simulate(new Car());`
  → Methoden von `Car` werden verwendet
- `simulate(new Boat());`
  → Methoden von `Boat` werden verwendet

Generics - Konzept

- Will man eine Datenstruktur implementieren, so braucht man für jeden Datentyp eine eigene Implementierung
  $\rightarrow$ Doppelter Code, unpraktisch
- Lösung: **Generics**
- Verwendet einen Datentyp als “Klassenparameter”
- Ermöglicht generisches, einmaliges Implementieren einer Datenstruktur
- Einer oder mehrere Typ-Parameter möglich
- Bereits durch die Listen der Java-API bekannt

Generics - Konzept

- Will man eine Datenstruktur implementieren, so braucht man für jeden Datentyp eine eigene Implementierung
  $\rightarrow$ Doppelter Code, unpraktisch
- Lösung: **Generics**
- Verwendet einen Datentyp als “Klassenparameter”
- Ermöglicht generisches, einmaliges Implementieren einer Datenstruktur
- Einer oder mehrere Typ-Parameter möglich
- Bereits durch die Listen der Java-API bekannt

Generics - Konzept

- Will man eine Datenstruktur implementieren, so braucht man für jeden Datentyp eine eigene Implementierung
  $\rightarrow$ Doppelter Code, unpraktisch
- Lösung: **Generics**
- Verwendet einen Datentyp als “Klassenparameter”
- Ermöglicht generisches, einmaliges Implementieren einer Datenstruktur
- Einer oder mehrere Typ-Parameter möglich
- Bereits durch die Listen der Java-API bekannt

Generics - Konzept

- Will man eine Datenstruktur implementieren, so braucht man für jeden Datentyp eine eigene Implementierung
  $\rightarrow$ Doppelter Code, unpraktisch
- Lösung: **Generics**
- Verwendet einen Datentyp als “Klassenparameter”
- Ermöglicht generisches, einmaliges Implementieren einer Datenstruktur
- Einer oder mehrere Typ-Parameter möglich
- Bereits durch die Listen der Java-API bekannt

Generics - Konzept

- Will man eine Datenstruktur implementieren, so braucht man für jeden Datentyp eine eigene Implementierung
  $\rightarrow$ Doppelter Code, unpraktisch
- Lösung: **Generics**
- Verwendet einen Datentyp als “Klassenparameter”
- Ermöglicht generisches, einmaliges Implementieren einer Datenstruktur
- Einer oder mehrere Typ-Parameter möglich
- Bereits durch die Listen der Java-API bekannt

Generics - Konzept

- Will man eine Datenstruktur implementieren, so braucht man für jeden Datentyp eine eigene Implementierung
  $\rightarrow$ Doppelter Code, unpraktisch
- Lösung: **Generics**
- Verwendet einen Datentyp als “Klassenparameter”
- Ermöglicht generisches, einmaliges Implementieren einer Datenstruktur
- Einer oder mehrere Typ-Parameter möglich
- Bereits durch die Listen der Java-API bekannt

Syntax

- Generische Klasse:
  ```
  class Name<DatentypParameter> {}
  ```
- Generisches Interface:
  ```
  interface Name<DatentypParameter> {}
  ```
- Erzeugen eines generischen Objekts:
  ```
  Name<Datentyp> varName = new Name<Datentyp>();
  ```
- Generische Klasse, mehrere Typ-Parameter:
  ```
  class Name<param1, param2> {}
  ```
- Generische Methode
  ```
  <Typ-Parameter> Typ Name(Parameterliste) {...}
  ```

Syntax

- Generische Klasse:
  ```
  class Name<DatentypParameter> {}
  ```
- Generisches Interface:
  ```
  interface Name<DatentypParameter> {}
  ```
- Erzeugen eines generischen Objekts:
  ```
  Name<Datentyp> varName = new Name<Datentyp>();
  ```
- Generische Klasse, mehrere Typ-Parameter:
  ```
  class Name<param1, param2> {}
  ```
- Generische Methode
  ```
  <Typ-Parameter> Typ Name(Parameterliste) {...}
  ```

Syntax

- Generische Klasse:
  ```
  class Name<DatentypParameter> {}
  ```
- Generisches Interface:
  ```
  interface Name<DatentypParameter> {}
  ```
- Erzeugen eines generischen Objekts:
  ```
  Name<Datentyp> varName = new Name<Datentyp>();
  ```
- Generische Klasse, mehrere Typ-Parameter:
  ```
  class Name<param1, param2> {}
  ```
- Generische Methode
  ```
  <Typ-Parameter> Typ Name(Parameterliste) {...}
  ```

Syntax

- Generische Klasse:
  ```
  class Name<DatentypParameter> {}
  ```
- Generisches Interface:
  ```
  interface Name<DatentypParameter> {}
  ```
- Erzeugen eines generischen Objekts:
  ```
  Name<Datentyp> varName = new Name<Datentyp>();
  ```
- Generische Klasse, mehrere Typ-Parameter:
  ```
  class Name<param1, param2> {}
  ```
- Generische Methode
  ```
  <Typ-Parameter> Typ Name(Parameterliste) {...}
  ```

Syntax

- Generische Klasse:
  ```
  class Name<DatentypParameter> {}
  ```
- Generisches Interface:
  ```
  interface Name<DatentypParameter> {}
  ```
- Erzeugen eines generischen Objekts:
  ```
  Name<Datentyp> varName = new Name<Datentyp>();
  ```
- Generische Klasse, mehrere Typ-Parameter:
  ```
  class Name<param1, param2> {}
  ```
- Generische Methode
  ```
  <Typ-Parameter> Typ Name(Parameterliste) {...}
  ```

Syntax

- Einschränkung durch **extends** (auch für Interfaces)
  `class C<T` **extends** `I>`
- z.B. zum Sicherstellen von Fähigkeiten
- Wildcard **?** als anonymen Parameter
  - `<?` **extends** `C>`
    Lesen mit Typ C möglich
  - `<?` **super** `D>`
    Zuweisen mit Typ D möglich

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - = 0 -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - $= 0$ -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - $= 0$ -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - $= 0$ -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - $= 0$ -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - $= 0$ -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Vergleiche von Objekten

Vergleiche von Objekten möglich mit:

- == für Referenzvergleich
- .equals() für Objektinhaltsgleichheit
- Interface Comparable<T>

Comparable<T>

- Vergleich wie bei equals()
- implementierende Klassen müssen compareTo(T o) implementieren
- Rückgabe:
  - $= 0$ -> beide sind gleich
  - < 0 -> Objekt ist kleiner als das Argument
  - > 0 -> Objekt ist größer als das Argument
- Collections.sort(liste) bzw Collections.sort(liste, comparator)

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Wrapper

- primitive Datentypen sind keine Objekte
- dadurch keine Vorteile von OOP
- Wrapper, die einen primitiven Typ halten und Funktionen bereitstellen
  - Wert ist immutable (nicht mehr änderbar)
  - Number als Oberklasse der numerischen Wrapper
- Erstellung:
  - Integer i = new Integer(1337);
  - Integer i2 = Integer.valueOf(1337);
  - Vorsicht: (Un-)Gleichheit, da nun Objekte
- Autoboxing
  - primitive Typen und Wrapper können (autom.) ineinander umgewandelt werden
  - manuell: wrapper = Integer.valueOf(val); val = wrapper.intValue()
  - int i = 1337; Integer j = i; int k = j;

Übungsaufgaben

Interfaces

Definiere ein Interface StandardList, welches dann von der SingleLinkedList Klasse implementiert werden soll.

Comparable

Füge zu den Klassen MotorizedVehicle, Car, usw. sinnvolle compareTo Methoden hinzu.

Übungsaufgaben

Generics

Ändere die Klasse SingleLinkedList so ab, dass sie für verschiedene Datentypen funktioniert.