TEKO
SCHWEIZERISCHE FACHSCHULE

Objektorientiertes Programmieren 2

Abstrakte Klassen und Schnittstellen

Stephan Kessler, BSc in Systems Engineering

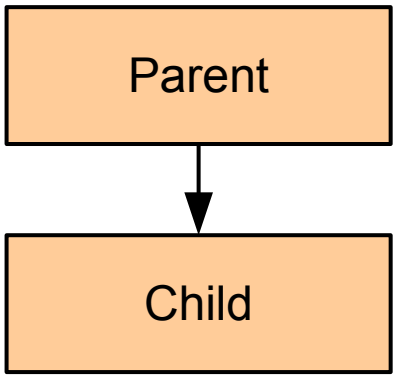
Kontakt: stephan.kessler@edu.teko.ch

Vorbereitung

Lesen Sie folgende Abschnitte im Buch ***Sprechen Sie Java?*** als Vorbereitung für dieses Thema. Beachten Sie, dass bei Angaben von Unterkapiteln nur diese auch gelesen werden müssen:

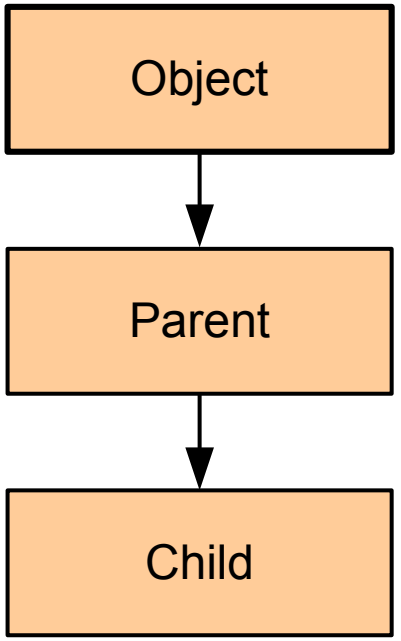
- super-Aufrufe (S. 186)
- Methoden und Klassen (Fokus auf this-Schlüsselwort) (S. 146)
- Abstrakte Klassen (S. 191-192)
- Interfaces (S. 193-195)
- Anonyme Klassen (S.195-197)

Klassenhierarchie



Zuvor haben Sie gelernt, dass Klassen von anderen Klassen erben können. Dadurch entsteht eine Hierarchie. Übergeordnete Klassen sind **Superklassen**. Von *Child* aus gesehen ist *Parent* eine Superklasse.

Klassenhierarchie Object



In Java ist es allerdings so, dass **jede** Klasse von der „Super“-Klasse ***Object*** erbt. Diese Klasse steht dadurch an der obersten Stelle der Hierarchie.

Klassenhierarchie Object

javax.swing

Class JFrame

java.lang.Object
    java.awt.Component
        java.awt.Container
            java.awt.Window
                java.awt.Frame
                    javax.swing.JFrame

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, Accessible, RootPaneContainer, WindowConstants

Dieses Schema wird auch in der Java-Doc ersichtlich. Wenn man z.B. die Klasse ***JFrame*** anschaut, sieht man die Klassenhierarchie im oberen Bereich. Dabei erkennt man, dass *JFrame* von der Klasse *Frame* erbt, diese wiederum von der Klasse *Window* usw. ***Object*** ist auch hier an der obersten Hierarchiestelle.

Klassenhierarchie Object

int	**hashCode**() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes the current thread to wait until another thread invokes the **notify()**

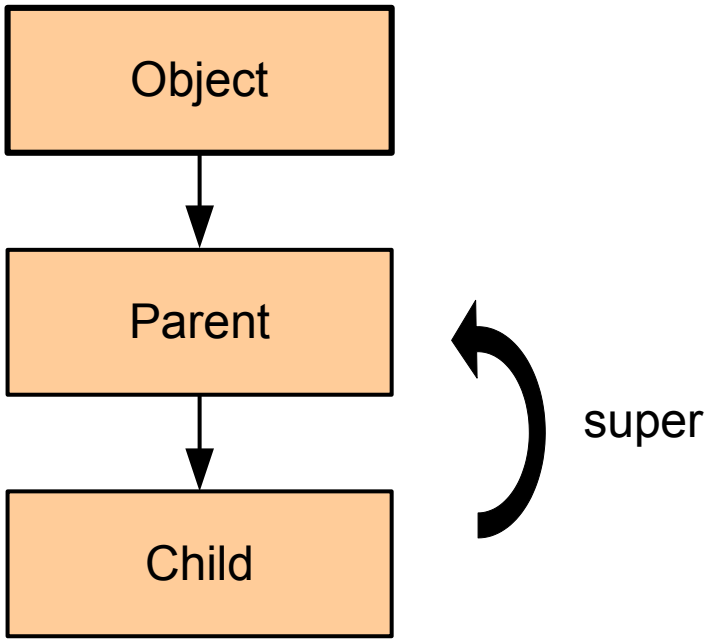
Da jede Klasse mindestens von der Klasse *Object* erbt, besitzt auch jede Klasse die zugänglichen Methoden von *Object*. In der Java-Doc-Seite von *Object* können diese entnommen werden. Als Beispiel ist die Methode *toString* in jedem Objekt aufrufbar. Diese Methode gibt (unverändert) einen Namen des instanziierten Objekts zurück.

toString() wird zB. Von der Methode *System.out.println()* verwendet. Jedes Objekt, welches dort übergeben wird, wird mit dessen *toString()*-Methode aufgerufen.

Aufgabe

- Schreiben Sie eine Klasse und überschreiben Sie die Methode *toString()* so, dass bei einem Aufruf ein “Hallo Welt“ ausgegeben wird.
- Schreiben Sie eine Hauptklasse mit einer Main-Methode und instanziieren Sie die zuvor programmierte Klasse. Rufen Sie die Methode System.out.println() auf und übergeben Sie das instanziierte Objekt.

Klassenhierarchie super



Wie bereits erwähnt sind übergeordnete Klassen Superklassen. Mit dem Schlüsselwort ***super*** können Sie dadurch auf die übergeordnete Klasse zugreifen.

Klassenhierarchie super

```java
public class Parent
{
    public void doSomething(){
        System.out.println("Hello");
    }
}
```

```java
public class Child extends Parent
{
    public void doSomething(){
        super.doSomething();
        System.out.println("Welt");
    }
}
```

Das Schlüsselwort kann z.B. beim Überschreiben von Methoden verwendet werden. Dabei werden diese Methoden nicht mehr überschrieben, sondern **erweitert**. Sie können die Position von ***super*** in einer Methode beliebig wählen (z.B. am Schluss als letzter Befehl (vor return)).

In diesem Beispiel wird „HalloWelt“ ausgegeben.

Klassenhierarchie super

```java
public class Parent
{
    public String name;

    public Parent(String surname){
        name = "Joe " + surname;
    }
}
```

```java
public class Child extends Parent
{
    public Child(String child_name){
        super(child_name);
        System.out.println(name);
    }
}
```

super können Sie auch für die Verbindung von Konstruktoren verwenden. Hierbei muss jedoch der ***super***-Aufruf am Anfang des Konstruktors stehen.

Würde ein Objekt *Child* mit dem String-Parameter „Pete“ instanziiert, würde bei diesem Beispiel der Text „Joe Pete“ ausgegeben.

Aufgabe

```java
public class Parent
{
     int parent_result;
     public Parent(int value){
          value = value + 2;
          parent_result = increase(value);
     }

     public int increase(int input){
          return input + 5;
     }

     public int decrease(int input){
          return input – 9;
     }
}
```

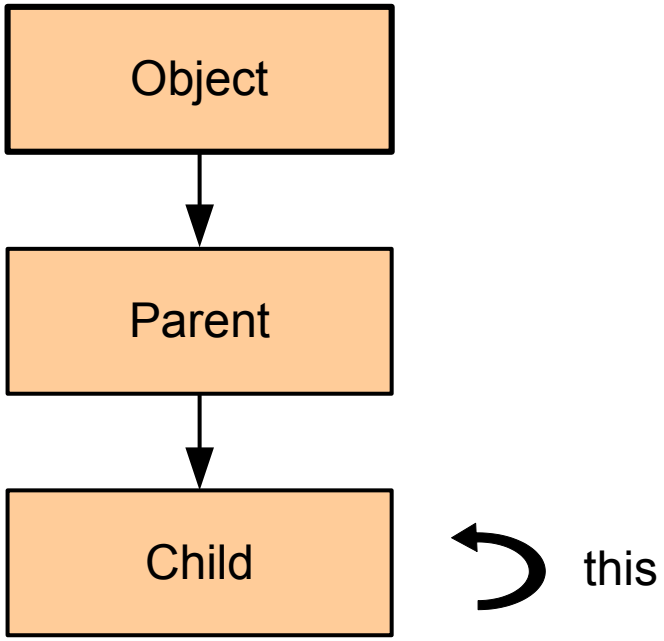
```java
public class Child extends Parent
{
     public Child(){
          super(3);
          parent_result = decrease(parent_result);
          System.out.println(parent_result);
     }

     public int increase(int input){
          input = input * 2;
          return super.increase(input);
     }

     public int decrease(int input){
          return input + 2;
     }
}
```

- Was wird ausgegeben, wenn *Child* instanziiert wird?

Klassenhierarchie this



In Java haben Sie auch die Möglichkeit von einem Objekts aus auf sich selber zuzugreifen. Dazu steht Ihnen das Schlüsselwort ***this*** zur Verfügung.

Klassenhierarchie this

```
public class Child extends Parent
{
    public String name;
    public int age;
    public float height;

    public Child(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
```

Dies ist vor allem dann Hilfreich, wenn Sie spezifisch auf die Eigenschaften des Objektes zugreifen wollen. Im Beispiel sehen Sie, dass der Konstruktor 3 Parameter aufnimmt, die den gleichen Namen wie die drei Objekt-Variablen haben.

Klassenhierarchie this

```
public class Child extends Parent
{
    public String name;
    public int age;
    public float height;

    public Child(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
```

In Java haben die Parameter Vorrang. Daher wird z.B. mit *name* (ohne *this)* im Konstruktor auf den Parameter zugegriffen.

Durch das Schlüsselwort *this* greift man jedoch auf das Objekt wie von aussen zu, wodurch der Zugriff sich auf die Objekt-Variablen (Eigenschaften) des Objektes beziehen.

Klassenhierarchie this

```
public class Child extends Parent
{
    public String name;
    public int age;
    public float height;

    public Child(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }

    public Child(){
        this("Hans", 27, 1.72f);
    }
}
```

Das Schlüsselwort *this* kann auch für den Aufruf von «eigenen» Konstruktoren verwendet werden. Im oberen Beispiel kann die Klasse *Child* mit dem Konstruktor ohne Parameter instanziiert werden. Dadurch wird zugleich der zweite Konstruktor mit vorgegebenen Argumenten aufgerufen (*Hans, 27, 1.72f*). Dieser *this*-Aufruf kann nur in einem Konstruktor statt finden!

Klassenhierarchie this

```java
public class Creature
{
     public Creature(){
          CreatureHandler.setCreature(this);
     }
}
```

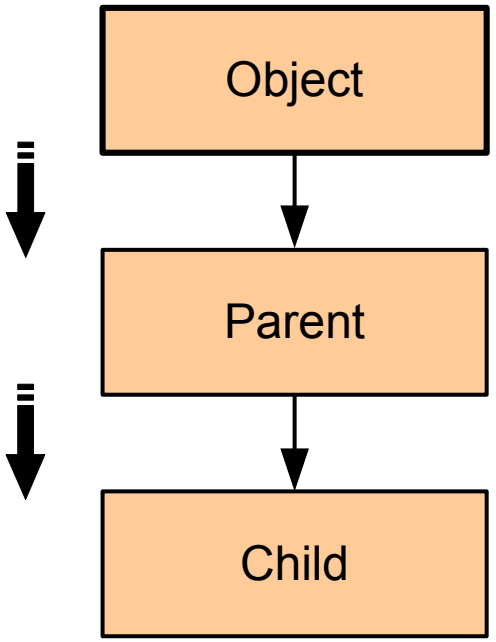
```java
public class CreatureHandler
{
     public static Creature creature;

     public static void setCreature(Creature new_creature){
          creature = new_creature;
     }
}
```

Auch können Sie *this* als Argument für einen Methodenaufruf verwenden. Dabei übergeben Sie das aktuelle Objekt der Methode.

In diesem Beispiel „registriert“ sich das instanziierte Objekt *Creature* während dem Konstruktoraufruf *im CreatureHanlder.*

Klassenhierarchie Typen



Bis anhin ist bekannt, dass ein instanziiertes Objekt aus *Child* auch den Typ *Child* besitzt. Durch die Hierarchie besitzt dieses Objekt jedoch auch den Typ der Eltern- / Super-Klassen.

Klassenhierarchie Typen

```
public class Hauptklasse
{
    public static void main(String[] args){
        Parent parent_object;
        parent_object = new Child();
    }
}
```

Daher ist diese Schreibweise möglich. Für das instanziierte Objekt aus *Child* kann eine Variable mit dem Typ *Parent* verwendet werden.

Klassenhierarchie Typen

```
public class Hauptklasse
{
    public static void main(String[] args){
        Parent parent_object;
        parent_object = new Child();

        parent_object.doParentStuff();
        parent_object.doChildStuff();
    }
}
```

Dabei können Sie alle Methoden und Eigenschaften auf der Stufe von *Parent* verwenden. Ein Zugriff auf Methoden und Eigenschaften die in *Child* abgelegt sind, funktionieren, wie hier geschrieben, nicht.

Das Objekt besitzt jedoch diese Methoden. Java denkt jedoch, dass es sich um ein *Parent*-Objekt handelt und verhindert dadurch den Zugriff auf unbekannte Elemente.

Klassenhierarchie Typen

```
public class Hauptklasse
{
    public static void main(String[] args){
        Parent parent_object;
        parent_object = new Child();

        parent_object.doParentStuff();

        Child child_object = (Child) parent_object;
        parent_object.doChildStuff();
    }
}
```

Sie können dies jedoch mit einem **Type-Cast** umgehen. Das Objekt bleibt dabei unverändert. Sie speichern dies lediglich noch in eine weitere Varibale des Types *Child* ab. Damit erhalten Sie wieder Zugriff auf die Eigenschaften und Methoden des *Child*-Objektes.

Klassenhierarchie instanceof

```
public class Hauptklasse
{
    public static void main(String[] args){
        Parent parent_object;
        parent_object = new Child();

        parent_object.doParentStuff();

        if(parent_object instanceof Child){
            Child child_object = (Child) parent_object;
            parent_object.doChildStuff();
        }
    }
}
```

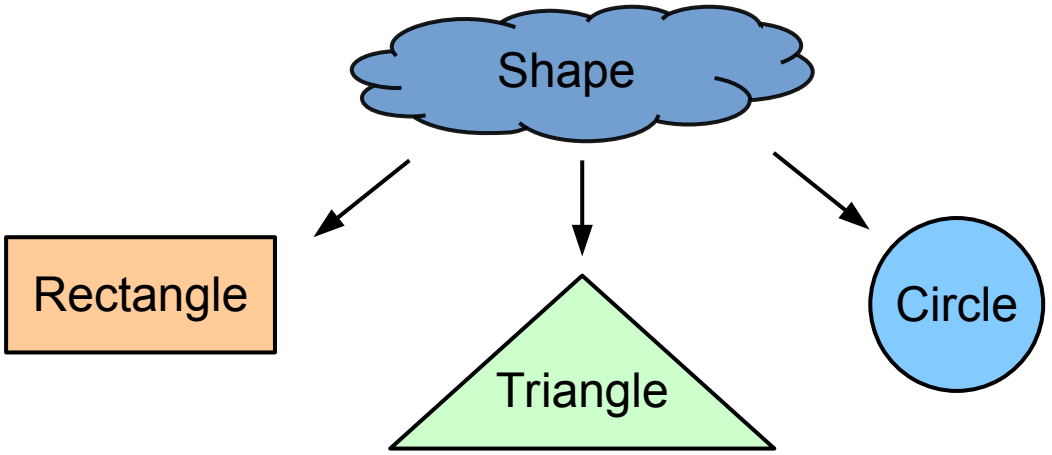
Bei einem Type-Cast können Sie jedoch in einen Fehler laufen, da Sie nicht immer sicher sein können, dass es sich hier auch um das zu wandelnde Objekt auch handelt.

Java bietet Ihnen hier das Schlüsselwort ***instanceof*** an. Damit können Sie die Hierarchiestruktur abfragen und überprüfen ob eine Stufe auch vorkommt. Das Schlüsselwort gibt entsprechen *true* oder *false* zurück.

Aufgabe

- Schreiben Sie drei Tier-Klassen: *Dog*, *Cat*, *Bird.*
- Schreiben Sie in jeder Tierklasse eine nichtstatische Methode die jeweils das Geräusch der Tiere auf die Konsole ausgibt. Versuchen Sie dabeid die Methoden-Namen unter den Tieren unterscheidlich zu wählen. (z.B *doDogSound*, *doCatSound*, *doBirdSound*)
- Schreiben Sie in einer Main-Methode ein Array welches 6 *Object*-Elemente aufnimmt. Füllen Sie diese Elemente frei mit *Dog*-, *Cat*-, *Bird*- Objekte (jeweils neu instanziiert).
- Schreiben Sie in der Main-Methode eine for-Schleife, welche alle Objekte durcharbeitet und herausfindet, ob es sich um ein *Dog*-, *Cat*- oder *Bird*-Objekt handelt und die entsprechende Methode startet.

Abstrakte Klassen



Abstrakte Klassen sind Strukturelemente in der Objektorientierten Programmierung. Sie können nicht direkt instanziiert werden. Eine Vererbung mit diesen ist jedoch möglich. Abstrakte Klassen können grundlegende Eigenschaften ihrer Unterklassen somit festlegen bzw. vorgeben.

Abstrakte Klassen

```java
public abstract class Animal
{
    public abstract void makeNoise();

    public void getType(){
        System.out.println("I am an Animal");
    }
}
```

Um in Java eine abstrakte Klasse zu erstellen, muss als Schlüsselwort ***abstract*** in der Klassenbeschreibung hinterlegt werden.

Abstrakte Klassen

```java
public abstract class Animal
{
    public abstract void makeNoise();

    public void getType(){
        System.out.println("I am an Animal");
    }
}
```

Der Unterschied zwischen einer abstrakten und einer „normalen“ Klasse in Java ist, dass bei einer abstrakten Klasse gewisse Methoden nur dessen Signatur beschrieben werden und dabei der Inhalt („Rumpf“ / Logik) ausgelassen wird.

Diese Methoden erhalten auch das Schlüsselwort ***abstract***.

Da die Logik der Methode fehlt, kann in diesem Zustand kein Objekt aus dieser Klasse direkt instanziiert werden.

Sie haben die Möglichkeit auch Variablen (Eigenschaften) in einer abstrakten Klasse zu setzten.

Abstrakte Klassen

```java
public class Dog extends Animal
{
    public void makeNoise(){
        System.out.println("Woof");
    }
}
```

```java
public class Cat extends Animal
{
    public void makeNoise(){
        System.out.println("Meow");
    }
}
```

Die abstrakte Klasse lässt sich jedoch vererben. Der Programmierer **muss** aber nun die abstrakte Klasse „überschreiben“ und dessen Logik programmieren.

Abstrakte Klassen

```
public class Hauptklasse
{
    public static void main(String[] args){
        Animal dog = new Dog();
        Animal cat = new Cat();

        dog.makeNoise();
        cat.makeNoise();
    }
}
```

Das Instanziieren der Objekte ist dabei gleich, wie bei einem „normalen“ Vererbungsprozess.

Abstrakte Klassen garantieren jedoch, dass gewisse Methoden, dessen Logik bei der Erstellung der Abstrakten Klasse noch unbekannt sind, bei der Implementierung durch den Programmierer bestimmt werden (der Programmierer wird dazu aufgefordert).

Polymorphismus

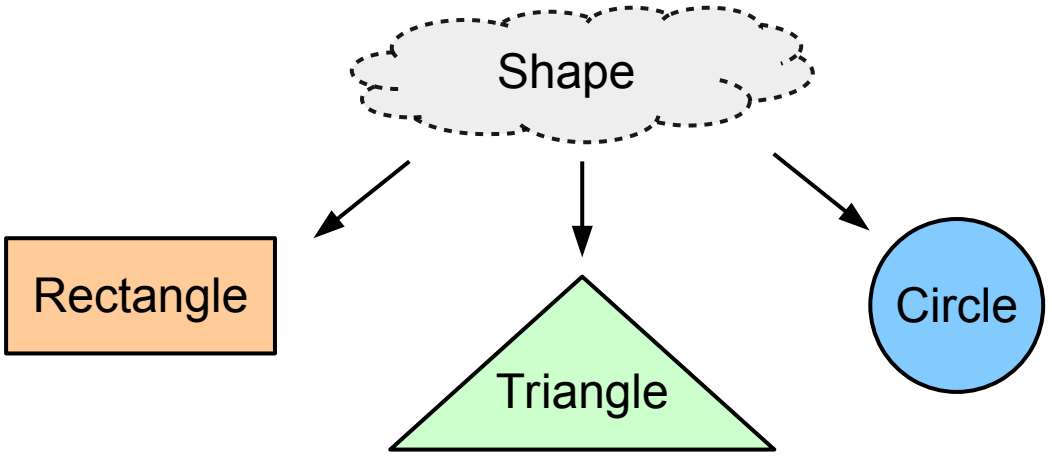
```
public class Hauptklasse
{
    public static void main(String[] args){
        Animal dog = new Dog();
        Animal cat = new Cat();

        dog.makeNoise();
        cat.makeNoise();
    }
}
```

Die Möglichkeit, dass eine Klasse mehrere Formen (in Java durch Vererbung und Überschreibung von Methoden) annehmen kann, nennt man in der Objektorientierten Programmierung **Polymorphismus** („mehrere Formen“).

Die beiden *Animals*-Objekte in diesem Beispiel besitzen weiterhin die Methode *makeNoise()*, jedoch führt der Aufruf dieser Methode jeweils etwas anderes aus.

Schnittstellen



In Java existieren neben Klassen auch Schnittstellen (**interfaces**). Diese haben eine sehr ähnlichen Aufbau wie Klassen und ähneln den abstrakten Klassen an Funktionalität. Bei abstrakten Klassen können Sie abstrakte sowie „normale“ Methoden kombinieren. In einer Schnittstelle können Sie nur abstrakte Methoden, sprich nur Methodenköpfe, hinterlegen.

Schnittstellen

```java
public interface Resettable
{
    public void reset();
}
```

Für den Aufbau einer Schnittstelle wird das Schlüsselwort ***interface*** verwendet.

Die Methodenköpfe werden hierbei ohne ein weiteres Schlüsselwort deklariert. Beachten Sie, dass Sie in einer Schnittstelle **keine** Methode mit einer Logik (Rumpf) programmieren können! Diese Kombination ist nur in einer abstrakten Klasse möglich.

Die einzige weitere Möglichkeit eine andere Arten von Element in einem Interface zu verwenden, sind statische, unveränderbare (final) Variablen, welche als Konstante betrachtet werden können.

Schnittstellen

```
public class Child extends Parent implements Resettable
{
    int x = 0;

    public void doChildStuff(){
        x = x + 2;
    }

    public void reset(){
        x = 0;
    }
}
```

Eine Schnittstelle kann in einer Klasse implementiert werden. Dieser Vorgang ist dem Vererben einer Klasse ähnlich. Als Schlüsselwort wird dazu ***implements*** anstelle *extends* verwendet.

Schnittstellen

```
public class Child extends Parent implements Resettable
{
    int x = 0;

    public void doChildStuff(){
        x = x + 2;
    }

    public void reset(){
        x = 0;
    }
}
```

Wie Sie bereits gelernt haben, ist es in Java nicht möglich in einer Klasse von mehreren Klassen gleichzeitig zu erben (Mehrfachvererbung). Daher können Sie nach dem Schlüsselwort *extends* nur eine Klasse angeben.

Java versucht damit Engpässe zu verhindern: z.B. Zwei vererbende Klassen besitzen eine Methode mit gleichem Methodenkopf. Es ist dabei unklar, welche Methode bei einem Super-Aufruf ausgeführt werden soll.

Schnittstellen

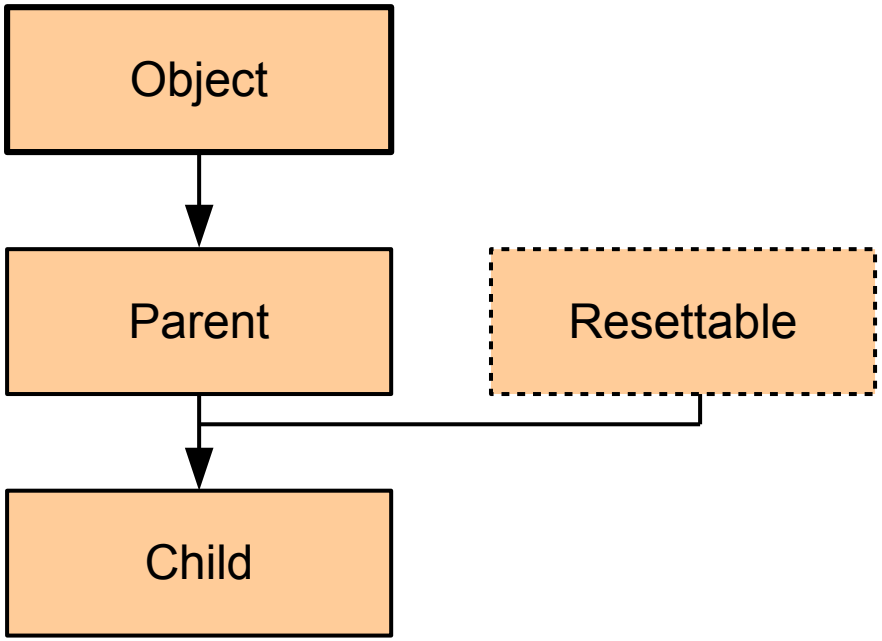
```
public class Child extends Parent implements Resettable, Disposable
{
    int x = 0;

    public void doChildStuff(){
        x = x + 2;
    }

    public void reset(){
        x = 0;
    }
}
```

Hingegen Schnittstellen können Sie so oft „vererben“ bzw. implementieren wie Sie wollen, da diese keine Methoden mit einer Logik anbieten. Dessen Logik (Rumpf) wird erst bei der Implementierung bestimmt. Nach dem Schlüsselwort ***implements*** können Sie durch Komma-Abtrennung weitere Schnittstellen angeben.

Schnittstellen



Schnittstellen vergeben auch ihren eigenen Typ bei einer Implementierung an die Klasse.

Dadurch besitzt *Child* folgenden Typen-Bezeichnungen: *Child*, *Parent, Object* und auch *Resettable*

Anonyme Klassen

```
public class Hauptklasse
{
    public static void main(String[] args){
        Parent parent = new Parent(){
            public void doParentStuff(){
                System.out.println("Parent Stuff");
            }
        };

        Animal animal = new Animal(){
            public void makeNoise(){
                System.out.println("sound...");
            }
        };
    }
}
```

Die Logik der Methoden von normalen und abstrakte Klassen, sowie auch Schnittstellen, können Sie auch direkt bei der Instanziierung bestimmen. Dazu schreiben Sie nach den Parametern-Klammern für den Konstruktoraufruf jeweils ***{}***. Sie erhalten dadurch wieder einen Bereich wo Sie entsprechen die Methoden überschreiben und auch weitere Eigenschaften (Variablen) setzten können.

Anonyme Klassen

```
public class Hauptklasse
{
    public static void main(String[] args){
        Parent parent = new Parent(){
            public void doParentStuff(){
                System.out.println("Parent Stuff");
            }
        };

        Animal animal = new Animal(){
            public void makeNoise(){
                System.out.println("sound...");
            }
        };
    }
}
```

Das erspart Ihnen das Erstellen einer weiteren Child-Klasse (Klasse die von diesem Objekt erbt und dessen Methoden überschreibt). Da es sich hier um eine neue Klasse handelt, diese jedoch keinen direkten Namen besitzt, nennt man diese auch **anonyme Klassen**.

In diesem Beispiel ist *Parent* eine normale und *Animal* eine abstrakte Klasse.