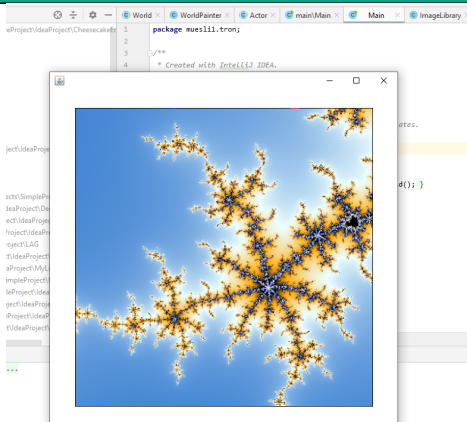


# FOP Recap #4



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Vererbung mit extends





---

# Hier könnte Ihre Werbung stehen



Hinweise

Vererbung

Vorgriff: Statischer und Dynamischer Typ

Zugriffsmodifikatoren

`final` und `this`

Live-Coding: Algomon

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Hinweise

- Beheben von häufigen Fehlern
- Bevor Sie eine Sprechstunde besuchen ...
- Dokumentation von Java-QuellCode
- Übungsblatt

## Vererbung

## Vorgriff: Statischer und Dynamischer Typ

## Zugriffsmodifikatoren

## ~~final und this~~

# Hinweise

## Beheben von häufigen Fehlern



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



Abbildung: Wiki-Seite zum Fehler beheben

<https://wiki.tudalgo.org/exercises/fix-errors/>

# Hinweise

Bevor Sie eine Sprechstunde besuchen ...



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

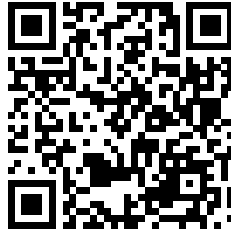
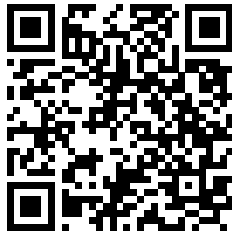


Abbildung: Wiki-Seite zu Sprechstunden-Fragen

<https://wiki.tudalgo.org/support/good-bad-questions/>



**Abbildung:** Wiki-Seite für Dokumentation mit JavaDoc  
<https://wiki.tudalgo.org/exercises/documentation/>



# Kleine Hinweise





Hinweise

Vererbung

Idee

Keyword extends

Definitionen

Verwendung im Detail

Lösung des Anfangsproblems

Vorgriff: Statischer und Dynamischer Typ

Zugriffsmodifikatoren



```
1 public class Car {  
2     public double speed;  
3  
4     public void accelerate(double a, double time) {  
5         speed += a*time;  
6     }  
7  
8     public void honk() {  
9         System.out.println("toot toot");  
10    }  
11 }
```



```
1 public class Truck {
2     public double speed;
3     public String company;
4
5     public void accelerate(double a, double time) {
6         speed += a*time;
7     }
8
9     public void honk() {
10        System.out.println("TOOT TOOT");
11    }
12 }
```

# Vererbung

## Idee



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 Car car = new Car();  
2 car.accelerate(5.2, 10);  
3 car.honk();
```

\$ T00T T00T

```
1 Truck truck = new Truck();  
2 truck.accelerate(5.2, 10);  
3 truck.honk();
```

\$ T00T T00T



```
1 public void doTheThing(???? vehicle) {  
2     vehicle.accelerate(5.2, 10);  
3     vehicle.honk();  
4 }
```

```
1 Car car = new Car();  
2 doTheThing(car);  
3 Truck truck = new Truck();  
4 doTheThing(truck);
```



Was wollen wir mit Vererbung erreichen?

- Doppelten Code vermeiden
- Trotzdem flexibel Code ersetzen oder ergänzen können
- Gleichen Code zum Verwenden ermöglichen

# Vererbung

## Keyword extends

```
1 public class A {  
2     .....  
3 }
```

```
1 public class B extends A {  
2     .....  
3 }
```

### Syntax extends:

*Zugriffsmodifikatoren* class *Klassen-Name* extends *Basis-Klassen-Name*



### Vererbung mit extends:

Jede\* Klasse **erbt direkt** von *einer* anderen Klasse. Diese Klasse nennt man dann ihre **Basis-Klasse** oder auch **Super-Klasse**. Falls von dem Programmierer keine Basis-Klasse mittels `extends` spezifiziert wird, wird automatisch `java.lang.Object` als Basis-Klasse verwendet.

Jede Klasse erbt **indirekt** von der Basis-Klasse seiner Basis-Klasse. Ebenso von der Basis-Klasse der Basis-Klasse seiner Basis-Klasse... und so weiter. Somit erbt **jede** Klasse letztendlich **indirekt** oder **direkt** von `java.lang.Object`.

\*`java.lang.Object` hat als einzige Klasse keine Basis-Klasse.



### Vererbung im Detail:

Es werden alle (*nicht-statischen*) **Methoden** sowie alle (*nicht-statischen*) **Attribute** von **direkten** sowie **indirekten Super-Klassen** geerbt. Auf die vererbten Attribute und Methoden kann je nach **Zugriffsmodifikatoren** zugegriffen werden. Die vererbten Methoden können **überschrieben** werden (solange Zugriff auf sie möglich ist).

Jeder **Konstruktor** einer Klasse **muss** entweder:

- einen Konstruktor seiner **Basis-Klasse**
- oder einen anderen Konstruktor seiner eigenen Klasse

als **erste** Anweisung aufrufen.

Falls seine Basis-Klasse einen **Standardkonstruktor** (Konstruktor ohne Parameter) hat und kein anderer eigener Konstruktor aufgerufen wird, wird ein parameterloser Konstruktoraufruf seiner Basis-Klasse implizit automatisch ergänzt.



```
1 public class MyCoolClass {  
2  
3     public MyCoolClass(int a, int b) {  
4         System.out.println(a + " & " + b);  
5     }  
6  
7 }
```



```
1 public class MyCoolClass extends Object {  
2  
3     public MyCoolClass(int a, int b) {  
4         super();  
5         System.out.println(a + " & " + b);  
6     }  
7  
8 }
```

# Vererbung

## Verwendung im Detail – Direkte und indirekte Vererbung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A { ..... }  
2 public class B extends A { ..... }  
3 public class C1 extends B { ..... }  
4 public class C2 extends B { ..... }  
5 public class D extends C2 { ..... }
```

- A erbt **direkt** von Object
- B erbt **direkt** von A sowie **indirekt** von Object
- C1 und C2 erben **direkt** von B sowie **indirekt** von Object und A
- D erbt **direkt** von C2 sowie **indirekt** von Object, A und B

# Vererbung

## Verwendung im Detail – Erben von Attributen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public int myNumber = 5;  
3 }
```

```
1 public class B extends A {  
2     public String coolString = "HappyThoughts";  
3  
4     public void print() {  
5         System.out.println(myNumber);  
6     }  
7 }
```

# Vererbung

## Verwendung im Detail – Erben von Attributen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B b = new B();  
2 b.print();  
3 b.myNumber = 42;  
4 b.print();
```

# Vererbung

## Verwendung im Detail – Erben von Attributen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B b = new B();  
2 b.print();  
3 b.myNumber = 42;  
4 b.print();
```

\$ 5

\$ 42

# Vererbung

## Verwendung im Detail – Erben von Attributen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public int myNumber = 5;  
3 }
```

```
1 public class B extends A {  
2     public String coolString = "HappyThoughts";  
3 }
```

```
1 public class C extends B {  
2     public double otherStuff = 0.2;  
3 }
```



# Vererbung

## Verwendung im Detail – Erben von Attributen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 C myC = new C();  
2 myC.myNumber = 42;  
3 myC.coolString = "EvenHappierThoughts";  
4 myC.otherStuff = -0.8;
```

# Vererbung

## Verwendung im Detail – Erben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public void greet() {  
3         System.out.println("Hello!");  
4     }  
5 }
```

```
1 public class B extends A {  
2     public void sayGoodbye() {  
3         System.out.println("Bye-bye!");  
4     }  
5 }
```



```
1 B anyName = new B();  
2 anyName.greet();  
3 anyName.sayGoodbye();
```

# Vererbung

## Verwendung im Detail – Erben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B anyName = new B();  
2 anyName.greet();  
3 anyName.sayGoodbye();
```

\$ Hello!

\$ Bye-bye!

# Vererbung

## Verwendung im Detail – Erben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public void greet() {  
3         System.out.println("Hello!");  
4     }  
5 }
```

```
1 public class B extends A {  
2     public void sayGoodbye() { ..... }  
3     public void test() {  
4         greet();  
5     }  
6 }
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public void greet() {  
3         System.out.println("Hello!");  
4     }  
5 }
```

```
1 public class B extends A {  
2     @Override // Optional  
3     public void greet() {  
4         System.out.println("Bonjour!");  
5     }  
6 }
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B localB = new B();  
2 localB.greet();
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B localB = new B();  
2 localB.greet();
```

```
$ Bonjour!
```



# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public void greet() {  
3         System.out.println("Hello!");  
4     }  
5 }
```

```
1 public class B extends A {  
2     public void greet(int a) {  
3         System.out.println("Bonjour!");  
4     }  
5 }
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B localB = new B();  
2 localB.greet();
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 B localB = new B();  
2 localB.greet();
```

```
$ Hello!
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public void greet() {  
3         System.out.println("Hello!");  
4     }  
5 }
```

```
1 public class B extends A {  
2     @Override // ERROR!  
3     public void greet(int a) {  
4         System.out.println("Bonjour!");  
5     }  
6 }
```

# Vererbung

## Verwendung im Detail – Überschreiben von Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public void greet() {  
3         System.out.println("Hello!");  
4     }  
5 }
```

```
1 public class B extends A {  
2     @Override // ERROR!  
3     public void greeta() {  
4         System.out.println("Bonjour!");  
5     }  
6 }
```



- @Override ist zu empfehlen, weil hiermit ein Error beim Kompilieren entsteht, wenn keine Methoden überschrieben wird
- Dies hilft solche Fehler direkt zu bemerken (Tippfehler etc)
- Es ist jedoch optional und ändert nichts an der Ausführung



```
1 public class A {  
2  
3 }
```

```
1 public class B extends A {  
2     public B() {  
3         // super(); automatisch ergänzt  
4     }  
5 }
```

# Vererbung

## Verwendung im Detail – Konstruktoren



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public A() {  
3  
4     }  
5 }
```

```
1 public class B extends A {  
2     public B() {  
3         // super(); automatisch ergänzt  
4     }  
5 }
```



# Vererbung

## Verwendung im Detail – Konstruktoren



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public A(int num) {  
3  
4     }  
5 }
```

```
1 public class B extends A {  
2     public B() {  
3         // ERROR!  
4     }  
5 }
```



```
1 public class A {  
2     public A(int num) { .... }  
3     public A(boolean flag) { .... }  
4 }
```

```
1 public class B extends A {  
2     public B() {  
3         // ERROR!  
4     }  
5 }
```



```
1 public class A {  
2     public A(int num) { .... }  
3     public A(boolean flag) { .... }  
4 }
```

```
1 public class B extends A {  
2     public B() {  
3         super(5);  
4     }  
5 }
```



```
1 public class A {  
2     public A(int num) { .... }  
3     public A(boolean flag) { .... }  
4 }
```

```
1 public class B extends A {  
2     public B() {  
3         super(false);  
4     }  
5 }
```

# Vererbung

## Verwendung im Detail – Konstruktoren



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class A {  
2     public A(int num) { .... }  
3     public A(boolean flag) { .... }  
4 }
```

```
1 public class B extends A {  
2     public B(int k) {  
3         super(k == 5);  
4     }  
5 }
```



```
1 public class B extends A {  
2     public B(int k) {  
3         boolean temp = k == 5; // ERROR!  
4         super(temp);  
5     }  
6 }
```



```
1 public class B extends A {  
2     public B(int k) {  
3         this(true, 0.2);  
4     }  
5  
6     public B(boolean myBool, double myDouble) {  
7         super(42);  
8     }  
9 }
```



```
1 public class Tree {  
2     public String message;  
3  
4     public Tree(String s, int strangeNumber) {  
5         System.out.println(s + "! " + strangeNumber);  
6         message = s;  
7     }  
8 }
```





```
1 public class LemonTree {  
2     public int numberOfLemons;  
3  
4     public LemonTree(int k) {  
5         super("All that I can see", k + 20);  
6         numberOfLemons = k;  
7     }  
8 }
```



```
1 LemonTree tree = new LemonTree(5);  
2 System.out.println(tree.message);  
3 System.out.println(tree.numberOfLemons);
```



```
1 LemonTree tree = new LemonTree(5);  
2 System.out.println(tree.message);  
3 System.out.println(tree.numberOfLemons);
```

```
$ All that I can see! 25  
    $ All that I can see  
    $ 5
```



```
1 public class Car {  
2     public double speed;  
3  
4     public void accelerate(double a, double time) {  
5         speed += a * time;  
6     }  
7  
8     public void honk() {  
9         System.out.println("toot toot");  
10    }  
11 }
```



```
1 public class Truck {
2     public double speed;
3     public String company;
4
5     public void accelerate(double a, double time) {
6         speed += a * time;
7     }
8
9     public void honk() {
10        System.out.println("TOOT TOOT");
11    }
12 }
```

# Vererbung

Lösung des Anfangsproblems – Nochmal wiederholt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 Car car = new Car();  
2 car.accelerate(5.2, 10);  
3 car.honk();
```

\$ T00T T00T

```
1 Truck truck = new Truck();  
2 truck.accelerate(5.2, 10);  
3 truck.honk();
```

\$ T00T T00T



```
1 public void doTheThing(???? vehicle) {  
2     vehicle.accelerate(5.2, 10);  
3     vehicle.honk();  
4 }
```

```
1 Car car = new Car();  
2 doTheThing(car);  
3 Truck truck = new Truck();  
4 doTheThing(truck);
```



Was wollen wir mit Vererbung erreichen?

- Doppelten Code vermeiden
- Trotzdem flexibel Code ersetzen oder ergänzen können
- Gleichen Code zum Verwenden ermöglichen





```
1 public class Vehicle {  
2     public double speed;  
3  
4     public void accelerate(double a, double time) {  
5         speed += a * time;  
6     }  
7  
8     public void honk() {  
9         // Leer, nur zum überschreiben gedacht  
10    }  
11 }
```



```
1 public class Car extends Vehicle {  
2     @Override  
3     public void honk() {  
4         System.out.println("toot toot");  
5     }  
6 }
```



```
1 public class Truck extends Vehicle {  
2     public String company;  
3  
4     @Override  
5     public void honk() {  
6         System.out.println("TOOT TOOT");  
7     }  
8 }
```



```
1 public void doTheThing(Vehicle vehicle) {  
2     vehicle.accelerate(5.2, 10);  
3     vehicle.honk();  
4 }
```

```
1 Car car = new Car();  
2 doTheThing(car);  
3 Truck truck = new Truck();  
4 doTheThing(truck);
```



```
1 public void doTheThing(Vehicle vehicle) {  
2     vehicle.accelerate(5.2, 10);  
3     vehicle.honk();  
4 }
```

```
1 Vehicle car = new Car();  
2 Vehicle truck = new Truck();  
3 doTheThing(car);  
4 doTheThing(truck);
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Hinweise

Vererbung

Vorgriff: Statischer und Dynamischer Typ

- Statischer Typ

- Dynamischer Typ

- Downcasting – Vorgriff

- Arrays

Zugriffsmodifikatoren

~~final und this~~

# Vorgriff: Statischer und Dynamischer Typ

## Statischer Typ



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Tree {  
2     public void grow() {  
3         // ...  
4     }  
5 }
```

```
1 public class LemonTree extends Tree {  
2     // Overwrite grow ...  
3 }
```

```
1 Tree happyTree = new Tree();  
2 happyTree.grow();
```

# Vorgriff: Statischer und Dynamischer Typ

## Statischer Typ



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 Tree happyTree = new Tree();  
2 happyTree.grow();
```

- In der Variable happyTree lassen sich Objekte von folgenden Typen speichern:
  - ▣ Tree
  - ▣ Jegliche Unterklasse von Klasse Tree, z.B. LemonTree





- Der statische Typ reicht für die Ausführung nicht aus!

```
1 Tree happyTree = new LemonTree();  
2 happyTree.grow();
```

- Je nach *dynamischen* Typen wird unterschiedliche Methode aufgerufen
- Hier ist der dynamische Typ `LemonTree`
- Dieser ist erst während der Laufzeit bekannt

# Vorgriff: Statischer und Dynamischer Typ

## Dynamischer Typ – instanceof



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Man kann mit instanceof eingrenzen, welcher dynamischer Typ vorliegt

```
1 Tree happyTree = ....
2 happyTree.grow();
3
4 if(happyTree instanceof LemonTree) {
5     // happyTree ist vom Typ LemonTree oder Unterklasse
6 }
7 if(happyTree instanceof Tree) {
8     // Gilt immer
9 }
10 if(happyTree instanceof AppleTree) {
11     // Könnte gelten, wenn AppleTree Unterklasse von Tree ist
12 }
```

# Vorgriff: Statischer und Dynamischer Typ

## Downcasting – Vorgriff



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Man kann nur Methoden aufrufen, die im statischen Typen definiert sind
- Mit Casting kann man den statischen Typen ändern

```
1 Tree happyTree = ...
2 happyTree.grow();
3
4 if(happyTree instanceof LemonTree) {
5     // happyTree ist vom Typ LemonTree oder Unterklasse
6     LemonTree lemonTree = (LemonTree) happyTree;
7     lemonTree.countLemons();
8 }
```

# Vorgriff: Statischer und Dynamischer Typ

## Arrays



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Tree { .... }
2 public class LemonTree extends Tree { .... }
3 public class AppleTree extends Tree { .... }
4 public class GoldenDeliciousAppleTree extends AppleTree { .... }
```

# Vorgriff: Statischer und Dynamischer Typ

## Arrays



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 Tree[] allTrees = new Tree[3];  
2 allTrees[0] = new LemonTree();  
3 allTrees[1] = new AppleTree();  
4 allTrees[2] = new GoldenDeliciousAppleTree();  
5  
6 ??? myTree = allTrees[1];
```

- Statischer Typ: ???
- Dynamischer Typ: ???

# Vorgriff: Statischer und Dynamischer Typ

## Arrays



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 Tree[] allTrees = new Tree[3];
2 allTrees[0] = new LemonTree();
3 allTrees[1] = new AppleTree();
4 allTrees[2] = new GoldenDeliciousAppleTree();
5
6 ??? myTree = allTrees[1];
```

- Statischer Typ: Tree
- Dynamischer Typ: AppleTree

# Vorgriff: Statischer und Dynamischer Typ

## Arrays



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 Tree[] allTrees = new Tree[3];
2 allTrees[0] = new LemonTree();
3 allTrees[1] = new AppleTree();
4 allTrees[2] = new GoldenDeliciousAppleTree();
5
6 Tree myTree = allTrees[1];
7 bool flag = myTree instanceof AppleTree; // true
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Hinweise

Vererbung

Vorgriff: Statischer und Dynamischer Typ

**Zugriffsmodifikatoren**

`final` und `this`

Live-Coding: Algomon





Möglicher Zugriff auf Attribute/Methoden einer Klasse:

- `private`
  - ▣ Überall **innerhalb** der Klasse
- (default)
  - ▣ Zusätzlich im gesamten Package
- `protected`
  - ▣ Zusätzlich in Klassen die erben
- `public`
  - ▣ Überall



## Modifier: public

```
1 public class Tree {  
2     public void grow() {  
3         // ...  
4     }  
5 }
```


```
1 public class LemonTree extends Tree ....
```

```
1 public class AppleTree extends Tree ....
```

## Access:

 fop

- ✓  Tree
- ✓  LemonTree
- ✓  Toast

 user

- ✓  Fruit
- ✓  AppleTree

## Modifier: (default)

```
1 public class Tree {  
2     void grow() {  
3         // ...  
4     }  
5 }
```


```
1 public class LemonTree extends Tree ....
```

```
1 public class AppleTree extends Tree ....
```

## Access:

 fop

- ✓  Tree
- ✓  LemonTree
- ✓  Toast

 user

- ✗  Fruit
- ✗  AppleTree



## Modifier: protected

```
1 public class Tree {  
2     protected void grow() {  
3         // ...  
4     }  
5 }
```


```
1 public class LemonTree extends Tree ....
```

```
1 public class AppleTree extends Tree ....
```

## Access:

 fop

- ✓  Tree
- ✓  LemonTree
- ✓  Toast

 user

- ✗  Fruit
- ✓  AppleTree



## Modifier: private


```
1 public class Tree {  
2     private void grow() {  
3         // ...  
4     }  
5 }
```


```
1 public class LemonTree extends Tree ....
```

```
1 public class AppleTree extends Tree ....
```

## Access:

 fop

- ✓  Tree
- ✗  LemonTree
- ✗  Toast

 user

- ✗  Fruit
- ✗  AppleTree

Modifier	Class	Package	Subclass	Global
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
(Default)	✓	✓	✗	✗
private	✓	✗	✗	✗



Hinweise

Vererbung

Vorgriff: Statischer und Dynamischer Typ

Zugriffsmodifikatoren

**final und this**

Als Modifikator für lokale Variablen

Als Modifikator für Attribute

Für Zugriff auf aktuelle Instanz

Zur Lösung von Shadowing

## Bedeutung vom Modifikator `final`:

- Für Variablen/Attribute: Es muss genau einmal ein Wert zugewiesen werden
- Für Klassen: Von `final` Klassen kann nicht geerbt werden



# final

## Als Modifikator für lokale Variablen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 final int i = 5;  
2 i = 7; // Nicht erlaubt!  
3 i++; // Nicht erlaubt!
```

# final

## Als Modifikator für lokale Variablen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 final int i;  
2 i = 8;  
3 i = 7; // Nicht erlaubt!
```

# final

## Als Modifikator für lokale Variablen



```
1 final int i;  
2 if(complexCondition() == true) {  
3     i = 5;  
4 }  
5 else {  
6     i = 3;  
7 }  
8 i = 7; // Nicht erlaubt!  
9  
10 int k = i;
```

# final

## Als Modifikator für lokale Variablen



```
1 final int i;  
2 if(complexCondition() == true) {  
3     i = 5;  
4 }  
5 else {  
6  
7 }  
8 // FEHLER! Fehlende Zuweisung von i!  
9  
10 int k = i;
```

# final

## Als Modifikator für Attribute



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class CuteZombie {  
2     public final int maxHealth = 5;  
3  
4     public void a() {  
5         maxHealth = 7; // ERROR!  
6     }  
7 }
```

# final

## Als Modifikator für Attribute



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class CuteZombie {  
2     public final int maxHealth;  
3  
4     // ERROR! Fehlende Zuweisung!  
5  
6     public void a() {  
7         maxHealth = 7; // ERROR!  
8     }  
9 }
```

# final

## Als Modifikator für Attribute



```
1 public class CuteZombie {  
2     public final int maxHealth;  
3  
4     public CuteZombie() {  
5         maxHealth = 8; // Erlaubt und notwendig!  
6     }  
7  
8     public void a() {  
9         maxHealth = 7; // ERROR!  
10    }  
11 }
```

# final

## Als Modifikator für Attribute



```
1 public class CuteZombie {  
2     public final int maxHealth;  
3  
4     public CuteZombie(int m) {  
5         maxHealth = m - 5;  
6  
7         maxHealth = 256; // ERROR!  
8     }  
9  
10    public void a() {  
11        maxHealth = 7; // ERROR!  
12    }  
13 }
```



# this

Für Zugriff auf aktuelle Instanz



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Toast {  
2     public void toastWith(Toaster toaster) {  
3         // ....  
4     }  
5 }
```

```
1 public class Toaster {  
2     public void process(Toast toast) {  
3         toast.toastWith(this);  
4     }  
5 }
```



```
1 public class Toast {  
2     private int timeToasted;  
3  
4     public void toastFor(int time) {  
5         timeToasted += time;  
6     }  
7 }
```



```
1 public class Toast {  
2     private int timeToasted;  
3  
4     public void toastFor(int timeToasted) {  
5         timeToasted += timeToasted; // Änderung des Attributs!  
6     }  
7 }
```



```
1 public class Toast {  
2     private int timeToasted;  
3  
4     public void toastFor(int timeToasted) {  
5         this.timeToasted += timeToasted;  
6     }  
7 }
```



```
1 public class Toast {  
2     private int timeToasted;  
3  
4     public Toast(int initialTimeToasted) {  
5         timeToasted = initialTimeToasted;  
6     }  
7 }
```

# this

## Zur Lösung von Shadowing



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Toast {  
2     private int timeToasted;  
3  
4     public Toast(int initialTimeToasted) {  
5         this.timeToasted = initialTimeToasted; // Redundantes this  
6     }  
7 }
```



```
1 public class Toast {  
2     private int timeToasted;  
3  
4     public Toast(int timeToasted) {  
5         this.timeToasted = timeToasted;  
6     }  
7 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Hinweise

Vererbung

Vorgriff: Statischer und Dynamischer Typ

Zugriffsmodifikatoren

`final` und `this`

Live-Coding: Algomon





# Live-Coding!