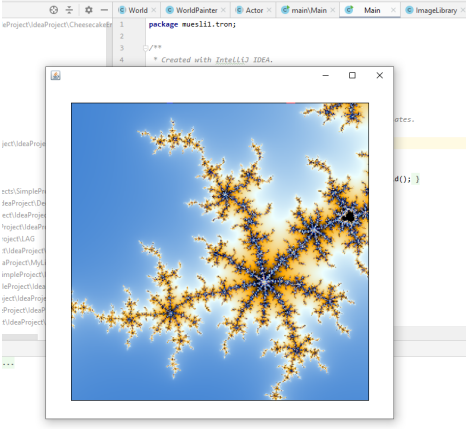
FOP Recap #5

Abstrakte Klassen, Interfaces und noch mehr

TECHNISCHE UNIVERSITÄT DARMSTADT



Hier könnte Ihre Werbung stehen

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Organisatorisches
- Methoden
- Interfaces
- Abstrakte Klassen
- Statische Methoden und Attribute
- Statischer und Dynamischer Typ
- Casting
- Scopes
- `String`

Organisatorisches

TECHNISCHE UNIVERSITÄT DARMSTADT

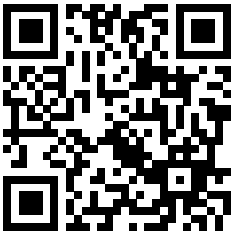


Abbildung: Fragen und Abstimmung
`https://participate.tudalgo.org/p/83215145`

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches

Methoden
- Überladen von Methoden
- Überladen von Konstruktoren
- `super` in Methoden

Interfaces

Abstrakte Klassen

Statische Methoden und Attribute

Statischer und Dynamischer Typ

Casting

Scopes

`String`

Methoden

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Eine Methode heißt *überladen*, wenn für den selben Typ[1] (mindestens) eine weitere gleichnamige Methode existiert

- keine zwei gleichnamigen Methoden mit selben Parametern in selber Reihenfolge
- aufrufbar wie „normale“ Methoden
- es wird immer die Methode gewählt, deren formale Parameter die aktualen Parameter am besten darstellen

> Überladen $\neq$ Überschreiben

[1] Klasse, Interface, ...

Methoden

Überladen von Methoden — Beispiel Klasse `MyPrinter`

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
public void print(String message) {
    System.out.println("string: " + message);
}

public void print(Object object) {
    System.out.println("object: " + object);
}

public void print(int number) {
    System.out.println("number: " + number);
}
```

Methoden

Überladen von Methoden — Beispiel Klasse `MyPrinter`

TECHNISCHE UNIVERSITÄT DARMSTADT

Was liefern folgende Aufrufe?

```
1  MyPrinter printer = new MyPrinter();
2  printer.print(1337);
3  printer.print("Hello Darmstadt!");
4  printer.print(4.2);
```

Methoden

Überladen von Methoden — Beispiel Klasse `MyPrinter`

TECHNISCHE UNIVERSITÄT DARMSTADT

Was liefern folgende Aufrufe?

```
1  MyPrinter printer = new MyPrinter();
2  printer.print(1337);
3  printer.print("Hello Darmstadt!");
4  printer.print(4.2);
```

```
$ number: 1337
$ string: Hello Darmstadt!
$ object: 4.2
```

Konstruktoren sind nur *spezielle* Methoden

- beim Überladen gleiche Eigenschaften wie bei „normalen“ Methoden

Methoden

Überladen von Konstruktoren — Beispiel Klasse `MyRobot`

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
public MyRobot(int x, int y, int numberOfCoins) {
    this.x = x;
    this.y = y;
    this.numberOfCoins = numberOfCoins;
}

public MyRobot(int x, int y) {
    this.x = x;
    this.y = y;
    this.numberOfCoins = 0;
}

public MyRobot() {
    this(0, 0); // calls constructor with two parameters
}
```

Methoden
super in Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

`super` bereits von Konstruktoren bekannt

- `super`-Aufruf ruft überschiebene Methode aus Basis-Klasse auf

Syntax

`super.`*Methodenname*`(`*aktuale Parameter*`);`

Methoden

super in Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
class A {

    void a() {
        println("A.a()");
    }

    void b() {
        println("A.b()");
    }
}
```

```java
class B extends A {

    @Override
    void a() {
        super.a();
        println("B.a()");
    }

    @Override
    void b() {
        println("B.b()");
    }
}
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Was liefern folgende Aufrufe?

```
1  B b = new B();
2  b.a(); // B overrides a with super
3  b.b(); // B overrides b without super
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Was liefern folgende Aufrufe?

```
1  B b = new B();
2  b.a(); // B overrides a with super
3  b.b(); // B overrides b without super
```

```
$ A.a()
$ B.a()
$ B.b()
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches

Methoden

Interfaces

- `implements` und `extends`
- Deklaration
- Beispiel
- Weitere Vorteile

Abstrakte Klassen

Statische Methoden und Attribute

Statischer und Dynamischer Typ

Casting

Scopes

`String`

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

> „An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface.“
>
> – Oracle

Idee

Trennung zwischen *Deklaration* und *Implementation*

- Interfaces *deklarieren* Methoden
- Klassen *implementieren* Methoden

Interfaces

Deklaration — Interface

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntax

<Modifiers> `interface` *<Interface Name>* `extends` *<Parent Interfaces>*

```java
interface A {
  // ...
}
```

```java
interface B {
    // no content
}
```

```java
interface C extends A, B {
  // ...
}
```

- Interfaces sind automatisch *immer* `public` – muss nicht angegeben werden

Interfaces

Deklaration — Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntax

<Modifiers> `interface` *<Interface Name>* `extends` *<Parent Interfaces>* `{ ... }`

- enthalten Methoden-*Deklarationen*
- Methoden in Interfaces sind auch *immer* `public`
- Methoden aus abgeleiteten Interfaces (*Parent Interfaces*) müssen nicht neu deklariert werden

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel

```java
interface A {
    void doMagic(int n);
    int doBad();
}
```

```java
interface C extends A, B {
    String getBehavior();
}
```

- folgende Deklarationen sind identisch:

```java
public int doMagic(int n);
int doMagic(int n);
```

Interfaces

Deklaration — `default`-Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Syntax

`default` <*Modifiers*> <*Return Type*> <*Method Name*>(<Parameters>) { `...` }

- können *immer* in implementieren Klassen überschrieben werden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassiker

- Hinweis auf fehlende Implementation

```
1  default void setZ() {
2      throwError();
3  }
```

- Unabhängigkeit von Implementation

```
1  default int getXPlusY() {
2      return getX() + getY();
3  }
```

Interfaces

Deklaration — Attribute

TECHNISCHE UNIVERSITÄT DARMSTADT

- Interface können keine *Objektattribute* haben → Implementation
- Interfaces können *nur public-final* Klassenattribute (`public` + `final` + `static`) haben
- müssen *direkt* initialisiert werden

- folgende Deklarationen + Initialisierungen sind identisch

```
1  public static final int MAGIC_NUMBER = 42;
2  int MAGIC_NUMBER = 42;
```

Interfaces

Deklaration — Attribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
interface Connectable {

    int DEFAULT_MAX_NUMBER_OF_CONNECTIONS = 42;

    void connect();

    default int maxNumberOfConnections() {
        return DEFAULT_MAX_NUMBER_OF_CONNECTIONS;
    }
}
```

Interfaces
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Zwei Klassen mit Position

```java
public class Person {

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
    // ...
}
```

```java
public class Car {

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
    // ...
}
```

Klasse zum Berechnen des Abstands zwischen zwei Personen

```
1  public PersonEuclidianDistanceCalculator {
2
3      public double calcDistance(Person p1, Person p2) {
4          int dx = p2.getX() - p1.getX();
5          int dy = p2.getY() - p1.getY();
6          return Math.sqrt(dx * dx + dy * dy);
7      }
8      // ...
9  }
```

Interfaces
Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

Der `PersonEuclidianDistanceCalculator` berechnet zwischen zwei *Personen* die *euklidische* Distanz.

Was ist, wenn weitere Klassen (und Distanzen) unterstützt werden sollen?

- je weiterer Klasse (und weiterer Distanz) *doppelt* so viele `Calculator`-Klassen
- für Kombinationen werden es noch mehr ...

Lösung

- alle Klassen, die Position haben, implementieren gemeinsames Interface
- alle Klassen, die Distanz berechnen können, implementieren gemeinsames Interface

TECHNISCHE UNIVERSITÄT DARMSTADT

Interface für Klassen mit Position

```
1  interface WithPosition {
2
3      int getX();
4      int getY();
5  }
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Interface für Distanzberechnung

```
1  interface DistanceCalculator {
2
3      double calcDistance(WithPosition p1, WithPosition p2);
4  }
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Interface `WithPosition` deklariert und implementiert Methode `getXPlusY()`

```
1  interface WithPosition {
2
3      int getX();
4      int getY();
5
6      default int getXPlusY() {
7          return getX() + getY();
8      }
9  }
```

Interfaces

Beispiel — Klasse zum Berechnen des Abstands zwischen zwei Objekten mit Position

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
public class Person implements
    ↪ WithPosition {

    @Override
    public int getX() {
        return x;
    }

    @Override
    public int getY() {
        return y;
    }
    ...
```

```java
public class Car implements
    ↪ WithPosition {

    @Override
    public int getX() {
        return x;
    }

    @Override
    public int getY() {
        return y;
    }
    ...
```

Interfaces

Beispiel — `DistanceCalculator` mit Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  public EuclidianDistanceCalculator {
2
3      public double calcDistance(WithPosition p1, WithPosition p2) {
4          int dx = p2.getX() - p1.getX();
5          int dy = p2.getY() - p1.getY();
6          return Math.sqrt(dx * dx + dy * dy)
7      }
```

Interfaces
Weitere Vorteile

TECHNISCHE UNIVERSITÄT DARMSTADT

Reminder: Jede Klasse kann nur *eine* Klasse erweitern.

- jede Klasse kann *mehrere* Interfaces implementieren

- „Module“ (hier: *DistanceCalculator*) können an zentraler Stelle ausgetauscht werden
- Implementation können einfach ausgetauscht werden

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Organisatorisches
- Methoden
- Interfaces
- Abstrakte Klassen
  - Syntax
  - Wichtige Eigenschaften
- Statische Methoden und Attribute
- Statischer und Dynamischer Typ
- Casting
- Scopes
- `String`

Abstrakte Klassen
Syntax

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  public abstract class MyClassName {
2      ....
3  }
```

Syntax abstrakte Klasse:

Zugriffsmodifikatoren `abstract class` *Klassen-Name*
`extends` *Basis-Klassen-Name*
`implements` *Interface-Namen*

Abstrakte Klassen

Wichtige Eigenschaften

TECHNISCHE UNIVERSITÄT DARMSTADT

- Abstrakte Klassen können abstrakte Methoden deklarieren
- Es kann kein Objekt von einer abstrakten Klasse erstellt werden
- Abstrakte Klassen müssen nicht alle abstrakten Methoden ihrer Basis-Klassen implementieren
  - Weder die abstrakten Methoden ihrer abstrakten Basisklassen
  - Noch die Methoden der Interfaces, von denen sie direkt oder indirekt erben

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Organisatorisches
- Methoden
- Interfaces
- Abstrakte Klassen
- Statische Methoden und Attribute
  - Allgemein
  - Klassenmethoden
  - Klassenattribute
- Statischer und Dynamischer Typ
- Casting
- Scopes
- `String`

Statische Methoden und Attribute

Allgemein

TECHNISCHE UNIVERSITÄT DARMSTADT

- Methoden und Attribute gehören immer zu einer Klasse
- `static` Methoden und Attribute sind jedoch unabhängig von Objekten dieser Klasse
- Man kann auch ohne Objekte auf `static` Methoden und Attribute zugreifen
- Objektmethoden lassen sich jedoch weiterhin nur mit einem Objekt aufrufen

Statische Methoden und Attribute

Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
public class MyClassName {
    public static void test() {
        ....
    }
    public void omnom() {
        test();
    }
}
```

```java
MyClassName.test();
```

Statische Methoden und Attribute

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

- Haben programm-weit denselben Wert
- Unabhängig von jeglichen Objekten

Statische Methoden und Attribute

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
public class MyClassName {
    public static int myValueName = 5;
    public void omnom() {
        myValueName = -1;
    }
}
```

```java
MyClassName.myValueName = -24;
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Organisatorisches
- Methoden
- Interfaces
- Abstrakte Klassen
- Statische Methoden und Attribute
- Statischer und Dynamischer Typ
  - Statischer Typ
  - Dynamischer Typ
  - Downcasting
  - Arrays
- Casting
- Scopes
- `String`

Statischer und Dynamischer Typ

Statischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
public class Tree {
    public void grow() {
        // ...
    }
}
```

```java
public class LemonTree extends Tree {
    // Overwrite grow ...
}
```

```java
Tree happyTree = new Tree();
happyTree.grow();
```

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`

Statischer und Dynamischer Typ

Dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

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 }
```

Statischer und Dynamischer Typ

Downcasting

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  }
```

Statischer und Dynamischer Typ

Downcasting

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
Tree happyTree = ...
happyTree.grow();

if(happyTree instanceof LemonTree) {
    // happyTree ist vom Typ LemonTree oder Unterklasse
    LemonTree lemonTree = (LemonTree) happyTree;
    lemonTree.countLemons();
}

if(happyTree instanceof LemonTree lemonTree) {
    // lemonTree ist nun vom Typ LemonTree
    lemonTree.countLemons();
}
```

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 { .... }
```

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: ???

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

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

- Organisatorisches
- Methoden
- Interfaces
- Abstrakte Klassen
- Statische Methoden und Attribute
- Statischer und Dynamischer Typ
- Casting
  - Bei primitiven Datentypen
  - Bei Objekttypen
- Scopes
- `String`

Casting

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei primitiven Datentypen
  - um einen Zahlenwert in einem anderen Typen zu speichern
  - passiert implizit oder explizit
- Bei Objekttypen
  - um den statischen Typen zu ändern
  - kann jedoch je nach dynamischen Typen fehlschlagen

Casting

Bei primitiven Datentypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jeder Zahltyp hat einen Bereich an Zahlen, den er repräsentieren kann
- `long`, `int`, `char`, `short`, `byte` für Ganzzahlen
- `double`, `float` für Komma-Zahlen
- Hierbei erkennt man folgende Rangordnung der Bereiche:
- `double` > `float` > `long` > `int` > `char` > `short` > `byte`

Casting

Bei primitiven Datentypen — Widening Casting

TECHNISCHE UNIVERSITÄT DARMSTADT

- Passiert implizit, automatisch

```
1  int a = 5;
2  int b = 27;
3  long c = a - b;
```

Casting

Bei primitiven Datentypen — Narrowing Casting

TECHNISCHE UNIVERSITÄT DARMSTADT

- Muss explizit angegeben werden, da Verlust von Präzision stattfinden kann!

```
1  int a = 288;
2  char c = a; // ERROR!
3  char c2 = (char) a; // OK!
4
5  char c3 = a + 5; // ERROR!
6  char c4 = ((char) a) + 5; // ERROR!
7  char c5 = (char) (a + 5); // OK!
```

Casting
Bei Objekttypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktioniert nur ohne Fehler bei passenden dynamischen Typen
- Sonst wird eine Exception geworfen

```
1  class A { .... }
2  class B extends A { .... }
3  class C extends B { .... }
4
5  A a = ....;
6  C castedC = (C) a; // Nur möglich,wenn a dynamischen Typen von C
   ↪  oder Subtypen hat
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Organisatorisches
- Methoden
- Interfaces
- Abstrakte Klassen
- Statische Methoden und Attribute
- Statischer und Dynamischer Typ
- Casting
- Scopes
  - Klassen/Interfaces/Enums
  - Attribute/Methoden
  - Parameter/Lokale Variablen
  - Shadowing
- `String`

Scopes

TECHNISCHE UNIVERSITÄT DARMSTADT

- Definieren in welchem Bereich Identifier sichtbar sind
- Möglicher Zugriff lässt sich dann (teils) durch Access Modifiers weiter einschränken

- Identifier sind zum Beispiel:
  - Klassen/Interfaces/Enums
  - Attribute
  - Variablen
  - Methoden
  - ...

Scopes
Klassen/Interfaces/Enums

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind bei uns normalerweise alle `public`
- Können dann überall importiert und genutzt werden

Scopes
Klassen/Interfaces/Enums

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sind bei uns normalerweise alle `public`
- Können dann überall importiert und genutzt werden

- Ergänzung:
  - In jeder Datei gibt es genau eine Top-Level Klasse, die den Dateinamen trägt
  - Kann jedoch beliebig viele nicht `public` Klassen geben

Scopes
Attribute/Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

- Können je nach Access Modifier genutzt werden

Scopes
Parameter/Lokale Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Können nur in ihrer Methode genutzt werden
- Sind nur innerhalb ihres „geschweiften Klammerpaares“ zulässig

```
1  public String global = "global";
2  public void foo() {
3      String local = "local";
4      System.out.println(global); // ->"global"
5      System.out.println(local); // ->"local"
6  }
7  public void bar() {
8      System.out.println(global); // ->"global"
9      System.out.println(local); // ->Compiler-Error
10 }
```

Scopes
Shadowing

TECHNISCHE UNIVERSITÄT DARMSTADT

- Falls zwei Variable im Scope mit demselben Namen vorliegen
- Lässt sich dann mit `this` und `super` lösen

```
1  public class Auto {
2      public double maxSpeed;
3
4      public Auto(double maxSpeed) {
5          this.maxSpeed = maxSpeed;
6      }
7  }
```

Das steht heute auf dem Plan

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches
Methoden
Interfaces
Abstrakte Klassen
Statische Methoden und Attribute
Statischer und Dynamischer Typ
Casting
Scopes

`String`
- Interner Aufbau
- `char`
- Beispiele

`String`
Interner Aufbau

TECHNISCHE UNIVERSITÄT DARMSTADT

- Im Prinzip nur ein `char[]`
- Jeder `String` is unveränderbar

String
char

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ist wie `int` ein primitiver Datentyp
- Belegt 2 Bytes Speicher statt wie ein `int` 4 Bytes
- Normale Verwendung:
  - Repräsentiert (im Normalfall) genau einen Buchstaben/Zeichen
  - Jedes Zeichen hat nach Unicode einen festen Zahlenwert zugeordnet

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  String s = "ABC";
2  char c0 = s.charAt(0); // == 'A'
3  char c1 = s.charAt(1); // == 'B'
4  char c2 = s.charAt(2); // == 'C'
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
String s = "ABC";
char[] arr = s.toCharArray();
char c0 = arr[0]; // == 'A'
char c1 = arr[1]; // == 'B'
char c2 = arr[2]; // == 'C'
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
String s = "ABC";
String result = "";
result += s.charAt(2);
result += s.charAt(1);
result += s.charAt(0);
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
String s = "ABC";
String result = "";
for(int i = s.length() - 1; i >= 0; i--) {
    result += s.charAt(i);
}
System.out.println(result);
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
String s = "ABC";
String result = "";
for(int i = s.length() - 1; i >= 0; i--) {
    result += s.charAt(i);
}
System.out.println(result);
```

```
$ CBA
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für Groß und Kleinschreibung!

```
1  char c = 'a';
2  char c2 = Character.toUpperCase(c); // == 'A'
3  boolean check = Character.isUpperCase(c2); // == true
4  char c3 = Character.toLowerCase(c2);
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
1  String message = "hihi";
2  String upperCaseMessage = message.toUpperCase();
3  System.out.println(upperCaseMessage);
```

String
Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```java
String message = "hihi";
String upperCaseMessage = message.toUpperCase();
System.out.println(upperCaseMessage);
```

```
$ HIHI
```

Live-Coding!