Algorithmen und Datenstrukturen

Vorlesung #01 – Einführung in Java Teil 1

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

10 · Apr · 2019

Themen der heutigen Vorlesung

- Informationen zu Java
- Überlick der Unterschiede C – Java

Themen der heutigen Vorlesung

- Informationen zu Java
- Überlick der Unterschiede C – Java
- Primitive Datentypen
- Arrays

Themen der heutigen Vorlesung

- Informationen zu Java
- Überlick der Unterschiede C – Java
- Primitive Datentypen
- Arrays
- Objektorientierte Programmierung
- Klassenkonzept: Attribute und Methoden

Java

- Entstanden aus einem Software Projekt für Unterhaltungselektronik wird ...
- Java als Programmiersprache für das Internet entwickelt.

  [1991 *Green Project*, Programmiersprache *Oak*, 1992 in *Java* umbenannt]
- Durchbruch durch Implementation im *Netscape Navigator 2.0* 1995.
- Entwickelt von *Sun Microsystems*, 2010 übernommen von *Oracle*

Java

- Entstanden aus einem Software Projekt für Unterhaltungselektronik wird ...
- Java als Programmiersprache für das Internet entwickelt.

  [1991 *Green Project*, Programmiersprache *Oak*, 1992 in *Java* umbenannt]
- Durchbruch durch Implementation im *Netscape Navigator 2.0* 1995.
- Entwickelt von *Sun Microsystems*, 2010 übernommen von *Oracle*

Durch die Orientierung Richtung Internet ergeben sich folgende Anforderungen:

- Programmcode, der über das Internet empfangen wird, sollte keinen Schaden anrichten
- Dies schließt eine Adaption von C oder C++ aus (ungültige Zeiger können unkontrolliert Speicher beschreiben)

Vorzüge von Java

- Objektorientierung
- Das Konzept der *Java Virtual Machine* (JVM)
  - Java Byte Code kann überall ausgeführt werden, wo eine JVM vorhanden ist, z.B. im Web Browser.
  - Unabhängig von Hardware und Betriebssystem

Vorzüge von Java

- Objektorientierung
- Das Konzept der *Java Virtual Machine* (JVM)
  - Java Byte Code kann überall ausgeführt werden, wo eine JVM vorhanden ist, z.B. im Web Browser.
  - Unabhängig von Hardware und Betriebssystem
- *Just-in-time-Compilation*, um die Laufzeit zu verbessern
- Referenzen an Stelle von Zeigern (mehr Sicherheit)
- Gute Speicherverwaltung - automatische Freigabe
- *Open Source*

Vorzüge von Java

- Objektorientierung
- Das Konzept der *Java Virtual Machine* (JVM)
  - Java Byte Code kann überall ausgeführt werden, wo eine JVM vorhanden ist, z.B. im Web Browser.
  - Unabhängig von Hardware und Betriebssystem
- *Just-in-time-Compilation*, um die Laufzeit zu verbessern
- Referenzen an Stelle von Zeigern (mehr Sicherheit)
- Gute Speicherverwaltung - automatische Freigabe
- *Open Source*

Zur Installation: Hinweise auf ISIS und in den Tutorien (Java Entwicklungsumgebung Intellij IDEA, Community Edition)

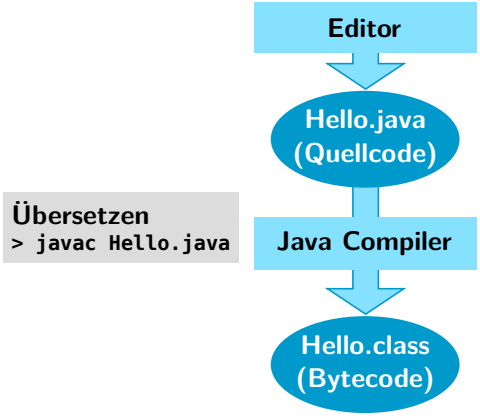
Ablauf des Programmierens in Java (ohne IDE)

Ablauf des Programmierens in Java (ohne IDE)

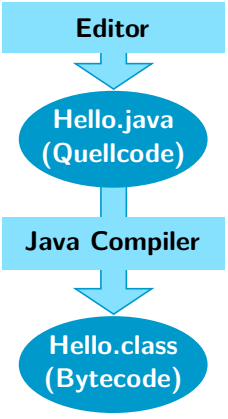
Übersetzen

```
> javac Hello.java
```

Ablauf des Programmierens in Java (ohne IDE)



Ablauf des Programmierens in Java (ohne IDE)



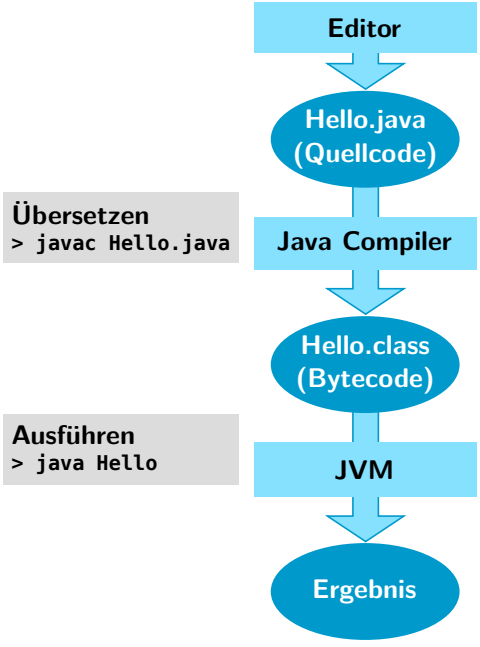
Übersetzen

```
> javac Hello.java
```

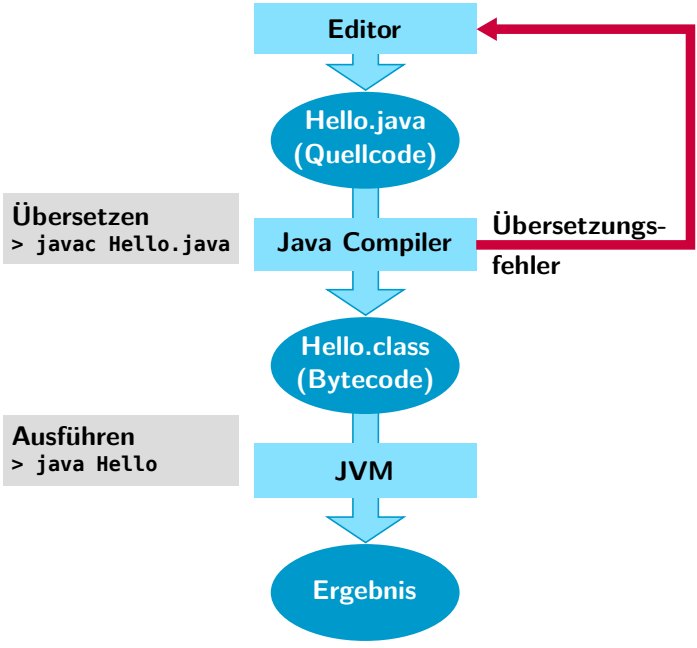
Ausführen

```
> java Hello
```

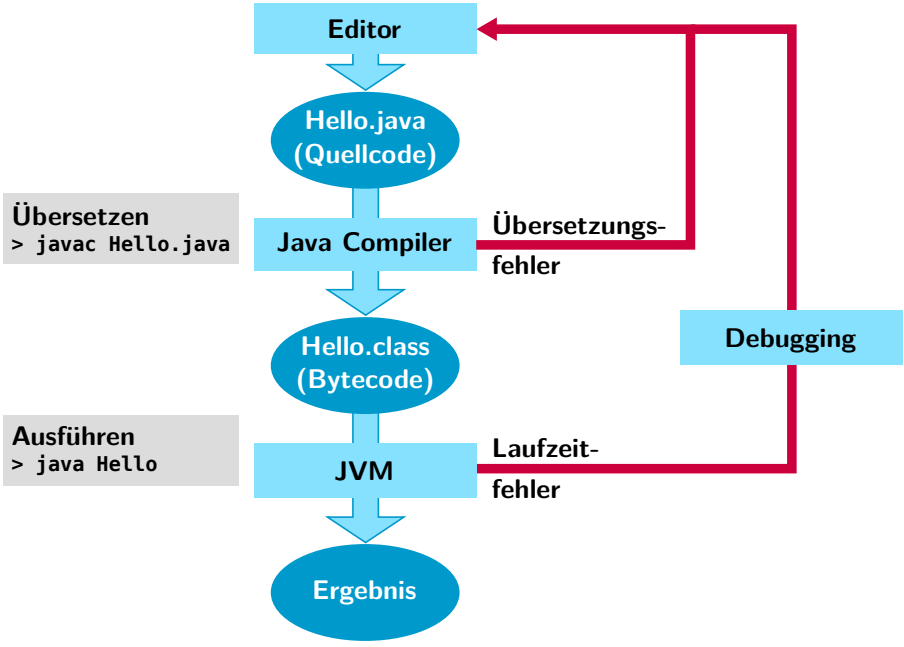
Ablauf des Programmierens in Java (ohne IDE)



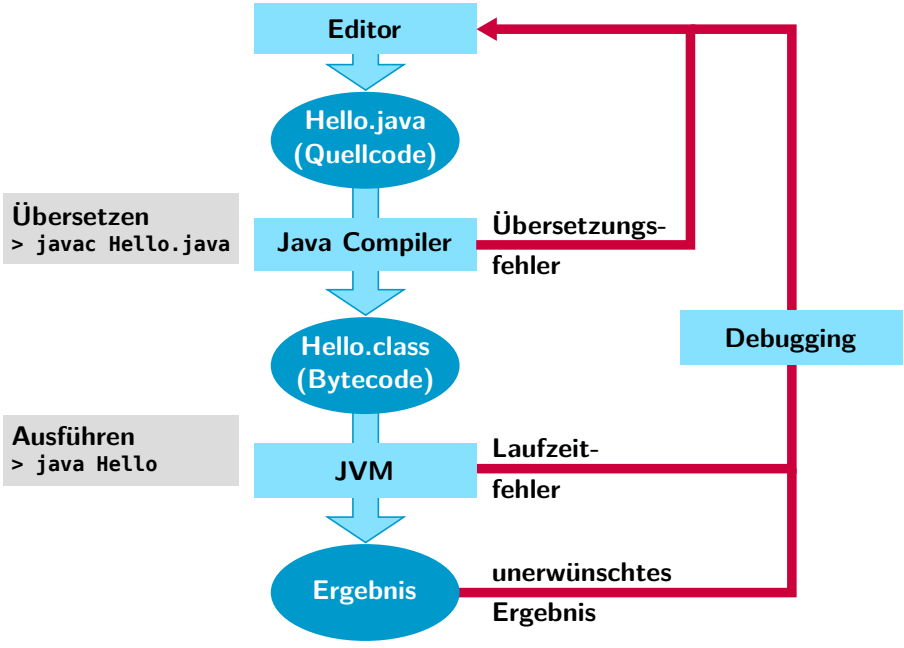
Ablauf des Programmierens in Java (ohne IDE)



Ablauf des Programmierens in Java (ohne IDE)



Ablauf des Programmierens in Java (ohne IDE)



Übersicht über die Unterschiede C – Java

	C	Java
Ausführung	Compiler erzeugt Maschinencode	Byte Code, muss interpretiert werden
Boolean	`int` mit 0 für *false* und $!=0$ für *true*	`boolean` mit Werten `true` und `false`
Array Deklaration	`int *x = malloc(N*sizeof(*x));`	`int[] x = new int[N];`
Array Größe	unbekannt für das Array	`x.length`
Zeichenketten	'\0' terminiertes `char` Array	Datentyp `String`

Übersicht über die Unterschiede C – Java

	C	Java
Ausführung	Compiler erzeugt Maschinencode	Byte Code, muss interpretiert werden
Boolean	`int` mit 0 für *false* und $!=0$ für *true*	`boolean` mit Werten `true` und `false`
Array Deklaration	`int *x = malloc(N*sizeof(*x));`	`int[] x = new int[N];`
Array Größe	unbekannt für das Array	`x.length`
Zeichenketten	'\0' terminiertes `char` Array	Datentyp `String`
Datenstruktur definieren	`struct`	`class` - Klassen mit Methoden

Übersicht über die Unterschiede C – Java

	C	Java
Ausführung	Compiler erzeugt Maschinencode	Byte Code, muss interpretiert werden
Boolean	`int` mit 0 für *false* und $!=0$ für *true*	`boolean` mit Werten `true` und `false`
Array Deklaration	`int *x = malloc(N*sizeof(*x));`	`int[] x = new int[N];`
Array Größe	unbekannt für das Array	`x.length`
Zeichenketten	'\0' terminiertes `char` Array	Datentyp `String`
Datenstruktur definieren	`struct`	`class` - Klassen mit Methoden
Bibliotheken laden	`#include <stdio.h>`	`import java.io.File;`

Bibilotheksfunktionen nutzen	`#include "math.h"`
	`x = sqrt(3.14);`
	Funktionen sind global
		`x = Math.sqrt(3.14);`
		Funktionen haben *namespaces*

Übersicht über die Unterschiede C – Java

	C	Java
Ausführung	Compiler erzeugt Maschinencode	Byte Code, muss interpretiert werden
Boolean	`int` mit 0 für *false* und $!=0$ für *true*	`boolean` mit Werten `true` und `false`
Array Deklaration	`int *x = malloc(N*sizeof(*x));`	`int[] x = new int[N];`
Array Größe	unbekannt für das Array	`x.length`
Zeichenketten	'\0' terminiertes `char` Array	Datentyp `String`
Datenstruktur definieren	`struct`	`class` - Klassen mit Methoden
Bibliotheken laden	`#include <stdio.h>`	`import java.io.File;`

Bibilotheksfunktionen nutzen	`#include "math.h"`
	`x = sqrt(3.14);`
	Funktionen sind global
		`x = Math.sqrt(3.14);`
		Funktionen haben *namespaces*

Speicher referenz.	Zeiger (*, &, +)	Referenzen
Speicher reservieren	`malloc`	`new`
Speicher freigeben	`free`	automatische Speicherbereinigung

Unterschiede C – Java (2)

	C	Java
Generischer Datentyp	`void *`	`Object`
Null	`NULL`	`null`

Unterschiede C – Java (2)

	C	Java
Generischer Datentyp	`void *`	`Object`
Null	`NULL`	`null`
Variablen automatisch initialisiert	nicht garantiert	Instanzvariablen und Array Elemente initialisiert mit `0`, `null`, bzw. `false`
Variablen deklarieren	am Anfang eines Blocks	irgendwo, vor der Benutzung
Variablennamen Konvention	`mean_square_error`	`meanSquareError`

Unterschiede C – Java (2)

	C	Java
Generischer Datentyp	`void *`	`Object`
Null	`NULL`	`null`
Variablen automatisch initialisiert	nicht garantiert	Instanzvariablen und Array Elemente initialisiert mit `0`, `null`, bzw. `false`
Variablen deklarieren	am Anfang eines Blocks	irgendwo, vor der Benutzung
Variablennamen Konvention	`mean_square_error`	`meanSquareError`
Dateinamen	`stack.c`, `stack.h`	`Stack.java` übereinstimmend mit Klassennamen
Bildschirmausgabe	`#include<stdio.h>` `printf("I am C\n");`	`System.out.println("I am Java\n");`
Kommentare	`/* ... */`	`/* ... */` oder vorangestelltes `//`

Siehe die vollständigere Liste auf der unten angegebenen Webseite.

Primitive Datentypen

In Java gibt es die folgenden **primitiven Datentypen** (*primitive data types*):

Datentyp	Inhalt	Wertebereich
`boolean`	1 Bit Wahrheitswert (Größe undefiniert)	`true`, `false`
`char`	16 Bit Unicode	Unicode Characters
`byte`	vorzeichenbehaftete ganze Zahl in 8 Bit	$-2^7 \cdots 2^7-1 = 127$
`short`	vorzeichenbehaftete ganze Zahl in 16 Bit	$-2^{15} \cdots 2^{15}-1 = 32767$
`int`	vorzeichenbehaftete ganze Zahl in 32 Bit	$-2^{31} \cdots 2^{31}-1 = 2147483647$
`long`	vorzeichenbehaftete ganze Zahl in 64 Bit	$-2^{63} \cdots 2^{63}-1 = 9223372036854775807$
`float`	Gleitkommazahl in 32 Bit	$\pm\ 2^{127} \approx 10^{38}$, 7 signifikante Stellen
`double`	Gleitkommazahl in 64 Bit	$\pm\ 2^{1023} \approx 10^{308}$, 15 signifikante Stellen

Primitive Datentypen

In Java gibt es die folgenden **primitiven Datentypen** (*primitive data types*):

Datentyp	Inhalt	Wertebereich
`boolean`	1 Bit Wahrheitswert (Größe undefiniert)	`true`, `false`
`char`	16 Bit Unicode	Unicode Characters
`byte`	vorzeichenbehaftete ganze Zahl in 8 Bit	$-2^7 \cdots 2^7-1 = 127$
`short`	vorzeichenbehaftete ganze Zahl in 16 Bit	$-2^{15} \cdots 2^{15}-1 = 32767$
`int`	vorzeichenbehaftete ganze Zahl in 32 Bit	$-2^{31} \cdots 2^{31}-1 = 2147483647$
`long`	vorzeichenbehaftete ganze Zahl in 64 Bit	$-2^{63} \cdots 2^{63}-1 = 9223372036854775807$
`float`	Gleitkommazahl in 32 Bit	$\pm\ 2^{127} \approx 10^{38}$, 7 signifikante Stellen
`double`	Gleitkommazahl in 64 Bit	$\pm\ 2^{1023} \approx 10^{308}$, 15 signifikante Stellen

Am häufigsten werden `boolean`, `int` und `double` verwendet, sowie der abstrakte Datentyp `String`.

Vorsicht bei der Wahl der Datentypen

- Datentypen sollten mit Bedacht gewählt werden, siehe
- YouTube *number of views* als `int` (Bereich bis 2.147.483.647)

Vorsicht bei der Wahl der Datentypen

- Datentypen sollten mit Bedacht gewählt werden, siehe
- YouTube *number of views* als `int` (Bereich bis 2.147.483.647)
- bis Dez. 2014!

YouTube ▸ Öffentlich 01.12.2014

We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!

Hover over the counter in PSY's video to see a little math magic and stay tuned for bigger and bigger numbers on YouTube.

Übersetzen

Arrays

Drei Schritte, um in Java ein Feld (*array*) oder Array anzulegen:

- Deklaration mit Angabe von Namen und Typ
- Speicher reservieren (Array anlegen)
- Elemente des Arrays mit Werten initialisieren

```
double[] x;                    // Variable als Array deklarieren
x = new double[K];             // und erzeugen (Speicher reservieren)
for (int k = 0; k < K; k++)
  x[k] = 0.0;                  // initialisieren
```

Arrays

Drei Schritte, um in Java ein Feld (*array*) oder Array anzulegen:

- Deklaration mit Angabe von Namen und Typ
- Speicher reservieren (Array anlegen)
- Elemente des Arrays mit Werten initialisieren

```
double[] x;                  // Variable als Array deklarieren
x = new double[K];           // und erzeugen (Speicher reservieren)
for (int k = 0; k < K; k++)
  x[k] = 0.0;                // initialisieren
```

Alle drei Schritte können in einer Zeile erledigt werden (automatische Initialisierung von Java):

```
double[] x = new double[K];
```

Arrays

Drei Schritte, um in Java ein Feld (*array*) oder Array anzulegen:

- Deklaration mit Angabe von Namen und Typ
- Speicher reservieren (Array anlegen)
- Elemente des Arrays mit Werten initialisieren

```
double[] x;                  // Variable als Array deklarieren
x = new double[K];           // und erzeugen (Speicher reservieren)
for (int k = 0; k < K; k++)
  x[k] = 0.0;                // initialisieren
```

Alle drei Schritte können in einer Zeile erledigt werden (automatische Initialisierung von Java):

```
double[] x = new double[K];
```

Deklaration mit Initialisierung durch eine Werteliste.

```
int[] fibo = { 0, 1, 1, 2, 3, 5, 8 };
```

Mehrdimensionale Arrays

- In Java gibt es keine ‘echten’ mehrdimensionalen Array.
- Man bildet Arrays von Arrays (von Arrays ...)
- Dies hat viele wichtige Konsequenzen, z.B. beim Kopieren und Vergleich, die später besprochen werden.

Mehrdimensionale Arrays

- In Java gibt es keine 'echten' mehrdimensionalen Array.
- Man bildet Arrays von Arrays (von Arrays ...)
- Dies hat viele wichtige Konsequenzen, z.B. beim Kopieren und Vergleich, die später besprochen werden.

```
// Deklaration 2-dim Array:
int[][] x;
// Erzeugen eines 2-dim Arrays
x = new int[4][3];
for (int i = 0; i < 4; i++)
  for (int j = 0; j < 3; j++)
    x[i][j] = i - j;
```

Mehrdimensionale Arrays

- In Java gibt es keine 'echten' mehrdimensionalen Array.
- Man bildet Arrays von Arrays (von Arrays ...)
- Dies hat viele wichtige Konsequenzen, z.B. beim Kopieren und Vergleich, die später besprochen werden.

```
// Deklaration 2-dim Array:
int[][] x;
// Erzeugen eines 2-dim Arrays
x = new int[4][3];
for (int i = 0; i < 4; i++)
  for (int j = 0; j < 3; j++)
    x[i][j] = i - j;
```

Die Arrays müssen nicht rechteckig sein:

```
int[][] x;
x = new int[5][];
for (int i = 0; i < x.length; i++) {
  x[i] = new int[i+1];
  for (int j = 0; j < x[i].length; j++)
    x[i][j] = i - j;
```

Mehrdimensionale Arrays

- In Java gibt es keine 'echten' mehrdimensionalen Array.
- Man bildet Arrays von Arrays (von Arrays ...)
- Dies hat viele wichtige Konsequenzen, z.B. beim Kopieren und Vergleich, die später besprochen werden.

```
// Deklaration 2-dim Array:
int[][] x;
// Erzeugen eines 2-dim Arrays
x = new int[4][3];
for (int i = 0; i < 4; i++)
  for (int j = 0; j < 3; j++)
    x[i][j] = i - j;
```

Die Arrays müssen nicht rechteckig sein:

```
int[][] x;
x = new int[5][];
for (int i = 0; i < x.length; i++) {
  x[i] = new int[i+1];
  for (int j = 0; j < x[i].length; j++)
    x[i][j] = i - j;
```

Zwei Möglichkeiten der Deklaration mit einer Werteliste:

```
int[][] square = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int[][] triangle = new int[][] {{1}, {2, 3}, {4, 5, 6}};
```

Iterieren über Arrays

Um alle Elemente eines Arrays anzusprechen, kann man über die Indizes iterieren:

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (int k = 0; k < folge.length; k++)
  sum += folge[k];
```

Iterieren über Arrays

Um alle Elemente eines Arrays anzusprechen, kann man über die Indizes iterieren:

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (int k = 0; k < folge.length; k++)
  sum += folge[k];
```

Oder man iteriert direkt über das Array: Diese Mechanimus wird in der nächsten Vorlesung besprochen (`Iterable`).

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (double zahl : folge)
    sum += zahl;
```

Iterieren über Arrays

Um alle Elemente eines Arrays anzusprechen, kann man über die Indizes iterieren:

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (int k = 0; k < folge.length; k++)
  sum += folge[k];
```

Oder man iteriert direkt über das Array: Diese Mechanimus wird in der nächsten Vorlesung besprochen (`Iterable`).

```
double[] folge = {12, -4, 5.6, 17};
double sum = 0.0;
for (double zahl : folge)
    sum += zahl;
```

Direkte Iteration über mehrdimensionale Arrays:

```
int[][] triangle = {{1}, {2, 3}, {4, 5, 6}};
int sum = 0;
for (int[] row : triangle)
    for (int element : row)
        sum += element;
```

Objektorientierte Programmierung (OOP)

Die Programmiersprache Java umfasst folgende Aspekte:

- imperative Prinzipen (Variablen, Operatoren, Fallunterscheidung, Schleifen, einfache statische Methoden, wie in C)
- allgemeine Objektorientierung (Objekte, Klassen, Vererbung, Schnittstellen)
- spezielle Erweiterungen von Java (Ausnahmen, Generics, Closures)

Das Konzept von Klassen und Objekten

Realität

reales Objekt

- hat Eigenschaften (Attribute)
- man kann etwas damit machen (Operationen)

Das Konzept von Klassen und Objekten

Realität	**reales Objekt** ▸ hat Eigenschaften (Attribute) ▸ man kann etwas damit machen (Operationen)
⇓ **Modellierung / Abstraktion** (Programmierung)	
Code	**Klasse** (fasst Objekte eines Typs zusammen) ▸ Definiert, welche Eigenschaften das Objekt besitzen kann (Attribute) ▸ Implementation der Operationen: Methoden

Das Konzept von Klassen und Objekten

Realität

reales Objekt

- hat Eigenschaften (Attribute)
- man kann etwas damit machen (Operationen)

⇓ **Modellierung / Abstraktion** (Programmierung)

Code

Klasse (fasst Objekte eines Typs zusammen)

- Definiert, welche Eigenschaften das Objekt besitzen kann (Attribute)
- Implementation der Operationen: Methoden

⇓ **Instanziierung** (beim Programmablauf)

Speicher

Objekt (Exemplar, Instanz der Klasse)

- Besitzt konkrete Eigenschaften (Attribute haben Werte)
- Anwendung einer Operation: Aufruf der Instanzmethode für das konkrete Objekt.

Klassen in Java

- Eine **Klasse** (*class*) definiert zusammengesetzte Datenstrukturen (wie `struct` in C) und die darauf zulässigen Operationen (objektbezogene **Methoden**).
- Jede Klasse besitzt mindestens einen **Konstruktor**, eine Methode mit dem Klassennamen, bei der kein Ausgabeargument angegeben wird, auch nicht `void`.
- Der Konstruktor wird aufgerufen, wenn man mit **`new`** ein Exemplar oder Instanz (*instance*) der Klasse erzeugt. Dies ist dann ein **Objekt**.

Klassen in Java

- Eine **Klasse** (*class*) definiert zusammengesetzte Datenstrukturen (wie `struct` in C) und die darauf zulässigen Operationen (objektbezogene **Methoden**).
- Jede Klasse besitzt mindestens einen **Konstruktor**, eine Methode mit dem Klassennamen, bei der kein Ausgabeargument angegeben wird, auch nicht `void`.
- Der Konstruktor wird aufgerufen, wenn man mit `new` ein Exemplar oder Instanz (*instance*) der Klasse erzeugt. Dies ist dann ein **Objekt**.
- Es kann mehrere Konstruktoren geben, wenn sie unterschiedliche Signaturen (Sequenz der Argumenttypen) haben.
- Dabei rufen häufig komplexere Konstruktoren die einfacheren Konstruktoren auf: **Verkettung von Konstruktoren**.
- Das Schlüsselwort `this` ist eine Referenz auf das Objekt, für das es aufgerufen wird. Mit `this(...)` kann ein anderer Konstruktor ausgerufen werden. Das muss aber immer die erste Codezeile im Konstruktor sein.

Klassen in Java

- Eine **Klasse** (*class*) definiert zusammengesetzte Datenstrukturen (wie `struct` in C) und die darauf zulässigen Operationen (objektbezogene **Methoden**).
- Jede Klasse besitzt mindestens einen **Konstruktor**, eine Methode mit dem Klassennamen, bei der kein Ausgabeargument angegeben wird, auch nicht `void`.
- Der Konstruktor wird aufgerufen, wenn man mit `new` ein Exemplar oder Instanz (*instance*) der Klasse erzeugt. Dies ist dann ein **Objekt**.
- Es kann mehrere Konstruktoren geben, wenn sie unterschiedliche Signaturen (Sequenz der Argumenttypen) haben.
- Dabei rufen häufig komplexere Konstruktoren die einfacheren Konstruktoren auf: **Verkettung von Konstruktoren**.
- Das Schlüsselwort `this` ist eine Referenz auf das Objekt, für das es aufgerufen wird. Mit `this(...)` kann ein anderer Konstruktor ausgerufen werden. Das muss aber immer die erste Codezeile im Konstruktor sein.
- Die Daten sind nach außen gekapselt: Der Zugriff ist nur über die festgelegten Methoden möglich, siehe Abstrakte Datentypen in der folgenden Vorlesung.

Kategorien von Variablen in Klassen

In Klassen gibt es drei Kategorien von Variablen:

- **Klassenvariablen** sind Datenfelder, die als **`static`** definiert werden. Sie existieren nur einziges mal pro Klasse und alle Objekte der Klasse können auf sie zugreifen.
- **Instanzvariablen** existieren unabhängig voneinander für jede Instanz der Klasse. Sie repräsentieren Eigenschaften bzw. den momentanen Zustand des konkreten Objektes.
- Lokale Variablen werden *innerhalb* von Methoden definiert und existieren nur für die Dauer des Methodenaufrufs.

Kategorien von Methoden in Klassen

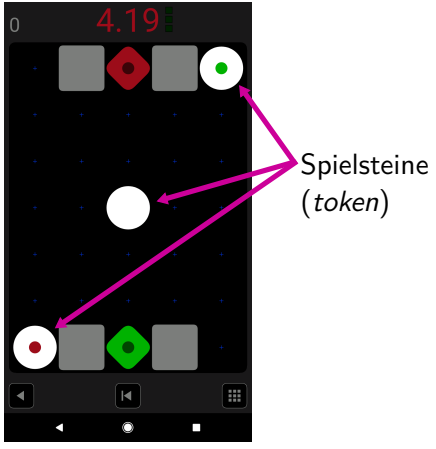
In Klassen gibt es zwei Kategorien von Methoden:

- **Klassenmethoden** sind Methoden, die als **`static`** definiert werden. Sie beziehen sich auf kein konkretes Objekt der Klasse (können allerdings als ein Argument ein Objekt der eigenen Klasse nehmen). Daher können sie nur auf Klassenvariablen und nicht auf Instanzvariablen zugreifen.
  Klassenmethoden ruft man über den Klassennamen und den Punkt Operator auf, z.B. `Math.log(17)`.

- **Instanzmethoden** existieren unabhängig voneinander für jede Instanz der Klasse. Sie beziehen sich immer auf eine konkrete Instanz bzw. Objekt der Klasse und können auch auf private Eigenschaften dieses Objektes zugreifen.
  Instanzmethoden ruft man über den Variablenamen des Objektes und den Punkt Operator auf, z.B. `str.length()` für eine Variable `str` der Klasse `String`.

Beispiel Klasse ‘Token’ (Spielsteine)

```
1  class Token {
2    static int counter;    // Klassenvariable
3    int xPos, yPos;        // Instanzvariablen
4
5    Token() {              // 1. Konstruktor
6      counter++;
7    }
8    Token(int x, int y) { // 2. Konstruktor
9      this();              // Verkettung
10     xPos = x;
11     yPos = y;
12   }
13                          // Methode
14   void moveTo(int x, int y) {
15     xPos = x;
16     yPos = y;
17   }
18 }
```



Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe unten.

Beispiel Klasse 'Token' (Spielsteine)

```java
class Token {
  static int counter;   // Klassenvariable
  int xPos, yPos;       // Instanzvariablen

  Token() {             // 1. Konstruktor
    counter++;
  }
  Token(int x, int y) { // 2. Konstruktor
    this();             // Verkettung
    xPos = x;
    yPos = y;
  }
                        // Methode
  void moveTo(int x, int y) {
    xPos = x;
    yPos = y;
  }
}
```

Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe unten.

```java
// Array deklarieren
Token[] spielstein = new Token[3];
// Konstruktor mit new aufrufen,
// um Objekte zu erstellen
spielstein[0] = new Token(0, 0);
spielstein[1] = new Token(2, 3);
spielstein[2] = new Token(4, 6);
// Methode aufrufen, um
// Objekt zu veraendern
spielstein[1].moveTo(0, 3);
// Zugriff auf Klassenvariable
System.out.println(Token.counter);
```

Beispiel Klasse 'Token' (Spielsteine)

```java
class Token {
  static int counter;   // Klassenvariable
  int xPos, yPos;       // Instanzvariablen

  Token() {             // 1. Konstruktor
    counter++;
  }
  Token(int x, int y) { // 2. Konstruktor
    this();             // Verkettung
    xPos = x;
    yPos = y;
  }
                        // Methode
  void moveTo(int x, int y) {
    xPos = x;
    yPos = y;
  }
}
```

```java
// Array deklarieren
Token[] spielstein = new Token[3];
// Konstruktor mit new aufrufen,
// um Objekte zu erstellen
spielstein[0] = new Token(0, 0);
spielstein[1] = new Token(2, 3);
spielstein[2] = new Token(4, 6);
// Methode aufrufen, um
// Objekt zu veraendern
spielstein[1].moveTo(0, 3);
// Zugriff auf Klassenvariable
System.out.println(Token.counter);
```

Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe unten.

Beispiel Klasse 'Token' (Spielsteine)

```java
class Token {
  static int counter;   // Klassenvariable
  int xPos, yPos;       // Instanzvariablen

  Token() {             // 1. Konstruktor
    counter++;
  }
  Token(int x, int y) { // 2. Konstruktor
    this();             // Verkettung
    xPos = x;
    yPos = y;
  }
                        // Methode
  void moveTo(int x, int y) {
    xPos = x;
    yPos = y;
  }
}
```

Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe unten.

```java
// Array deklarieren
Token[] spielstein = new Token[3];
// Konstruktor mit new aufrufen,
// um Objekte zu erstellen
spielstein[0] = new Token(0, 0);
spielstein[1] = new Token(2, 3);
spielstein[2] = new Token(4, 6);
// Methode aufrufen, um
// Objekt zu veraendern
spielstein[1].moveTo(0, 3);
// Zugriff auf Klassenvariable
System.out.println(Token.counter);
```

Beispiel Klasse 'Token' (Spielsteine)

```java
class Token {
  static int counter;   // Klassenvariable
  int xPos, yPos;       // Instanzvariablen

  Token() {             // 1. Konstruktor
    counter++;
  }
  Token(int x, int y) { // 2. Konstruktor
    this();             // Verkettung
    xPos = x;
    yPos = y;
  }
                        // Methode
  void moveTo(int x, int y) {
    xPos = x;
    yPos = y;
  }
}
```

Hier fehlen noch die für Java typischen Zugriffsmodifizierer, siehe unten.

```java
// Array deklarieren
Token[] spielstein = new Token[3];
// Konstruktor mit new aufrufen,
// um Objekte zu erstellen
spielstein[0] = new Token(0, 0);
spielstein[1] = new Token(2, 3);
spielstein[2] = new Token(4, 6);
// Methode aufrufen, um
// Objekt zu veraendern
spielstein[1].moveTo(0, 3);
// Zugriff auf Klassenvariable
System.out.println(Token.counter);
```

Die Klassenvariable gehört zur ganzen Klasse. Daher wird sie nicht mit einem Objekt, sondern dem Klassennamen aufgerufen.

Sichtbarkeitstypen von Variablen

- Klassen sind in einer Hierarchie angeordnet: Oberklassen, Unterklassen, Vererbung (nächste Vorlesung mehr dazu)
- Alle Klassen, die in einem gemeinsamen Verzeichnis liegen, bilden ein Paket (*package*). Dies sollten sinnvolle zusammengehörige Klassen sein.

Sichtbarkeitstypen von Variablen

- Klassen sind in einer Hierarchie angeordnet: Oberklassen, Unterklassen, Vererbung (nächste Vorlesung mehr dazu)
- Alle Klassen, die in einem gemeinsamen Verzeichnis liegen, bilden ein Paket (*package*). Dies sollten sinnvolle zusammengehörige Klassen sein.
- Die Sichtbarkeits- und Zugriffsrechte von Variablen in dieser Hierarchie werden bei der Deklaration durch die Zugriffsmodifizierer **`public`**, **`protected`** und **`private`** festgelegt. Ohne Modifizierer gilt das Zugriffsrecht '*package*'.

Wer darf?	private	package	protected	public
Die Klasse selbst, innere Klassen	ja	ja	ja	ja
andere Klassen im selben Paket	nein	ja	ja	ja
Unterklassen in anderem Paket	nein	nein	ja	ja
Sonstige Klassen	nein	nein	nein	ja

Sichtbarkeitstypen von Variablen

- Klassen sind in einer Hierarchie angeordnet: Oberklassen, Unterklassen, Vererbung (nächste Vorlesung mehr dazu)
- Alle Klassen, die in einem gemeinsamen Verzeichnis liegen, bilden ein Paket (*package*). Dies sollten sinnvolle zusammengehörige Klassen sein.
- Die Sichtbarkeits- und Zugriffsrechte von Variablen in dieser Hierarchie werden bei der Deklaration durch die Zugriffsmodifizierer **`public`**, **`protected`** und **`private`** festgelegt. Ohne Modifizierer gilt das Zugriffsrecht '*package*'.

Wer darf?	private	package	protected	public
Die Klasse selbst, innere Klassen	ja	ja	ja	ja
andere Klassen im selben Paket	nein	ja	ja	ja
Unterklassen in anderem Paket	nein	nein	ja	ja
Sonstige Klassen	nein	nein	nein	ja

- Zusätzlich kann der Modifizierer **`final`** verwendet werden, um Konstanten zu definieren.

Update der Beispiel Klasse

```
1  public class Token {
2    private static final Shapetype shape = CIRCLE;  // Konstante
3    private static int counter;
4    private int xPos, yPos;
5
6    public Token() {
7      counter++;
8    }
9    public Token(int x, int y) {
10     this();
11     xPos = x;
12     yPos = y;
13   }
14
15   public void moveTo(int x, int y) {
16     xPos= x;
17     yPos= y;
18   }
19 }
```

Bemerkung zur Sichtbarkeit von Instanzvariablen

- Instanzvariablen sollten nicht **`public`** definiert werden:
  - Kontrollierter Zugriff
  - Flexibilität: Implementation der Datenstruktur kann geändert werden, ohne dass der Client-Code geändert werden muss.

Bemerkung zur Sichtbarkeit von Instanzvariablen

- Instanzvariablen sollten nicht **`public`** definiert werden:
  - Kontrollierter Zugriff
  - Flexibilität: Implementation der Datenstruktur kann geändert werden, ohne dass der Client-Code geändert werden muss.
- Aber wie kann dann auf `xPos`, `yPos` eines Spielsteins zugegriffen werden?

Bemerkung zur Sichtbarkeit von Instanzvariablen

- Instanzvariablen sollten nicht **`public`** definiert werden:
  - Kontrollierter Zugriff
  - Flexibilität: Implementation der Datenstruktur kann geändert werden, ohne dass der Client-Code geändert werden muss.
- Aber wie kann dann auf `xPos`, `yPos` eines Spielsteins zugegriffen werden?
- Über *getter* und *setter* Methoden!

```
1  private int xPos;
2  // ...
3  public int getXPos() {
4    return xPos;
5  }
6  public void setXPos(int x) {
7    xPos= x;
8  }
```

- Auf primitive Datentypen wird diese Konvention nicht immer angewendet. Hier muss der Nutzen gegen den zusätzlichen Aufwand abgewogen werden.

Objekte

- Objekte sind Instanzen von Klassen. Sie werden per **`new`** durch den Konstruktor erzeugt.
- Objekte sind durch folgende Aspekte charakterisiert:
  - **Identität:** Speicherbereich des Objektes
  - **Zustand:** Wert des Datentyps (Instanzvariablen)
  - **Verhalten:** Definiert durch die Objektmethoden

Objekte

- Objekte sind Instanzen von Klassen. Sie werden per **new** durch den Konstruktor erzeugt.
- Objekte sind durch folgende Aspekte charakterisiert:
  - **Identität:** Speicherbereich des Objektes
  - **Zustand:** Wert des Datentyps (Instanzvariablen)
  - **Verhalten:** Definiert durch die Objektmethoden
- Der Zugriff auf Objekte geschieht über Referenzen. Daher werden die nicht-primitiven Datentypen auch Referenztypen genannt.
- Bei einem Aufruf von new wird Speicher reserviert, Werte initialisiert durch den Konstruktor und eine Referenz auf das Objekt zurückgegeben.

Objekte

- Objekte sind Instanzen von Klassen. Sie werden per **new** durch den Konstruktor erzeugt.
- Objekte sind durch folgende Aspekte charakterisiert:
  - **Identität:** Speicherbereich des Objektes
  - **Zustand:** Wert des Datentyps (Instanzvariablen)
  - **Verhalten:** Definiert durch die Objektmethoden
- Der Zugriff auf Objekte geschieht über Referenzen. Daher werden die nicht-primitiven Datentypen auch Referenztypen genannt.
- Bei einem Aufruf von new wird Speicher reserviert, Werte initialisiert durch den Konstruktor und eine Referenz auf das Objekt zurückgegeben.
- Wenn new mehrfach aufgerufen wird, werden mehrere Objekte derselben Klasse erzeugt, jedes mit einer eigenen Identität (siehe Beispiel Klasse auf S. 17).

Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

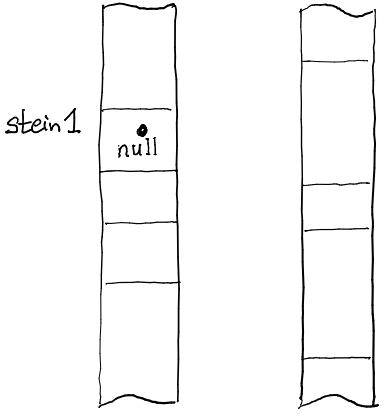
```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

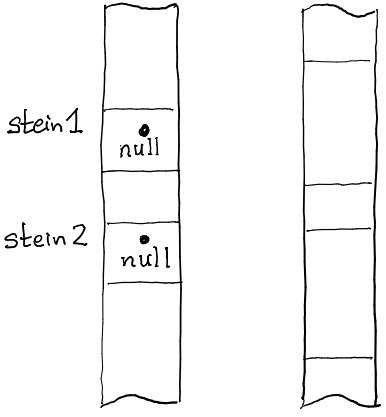


Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

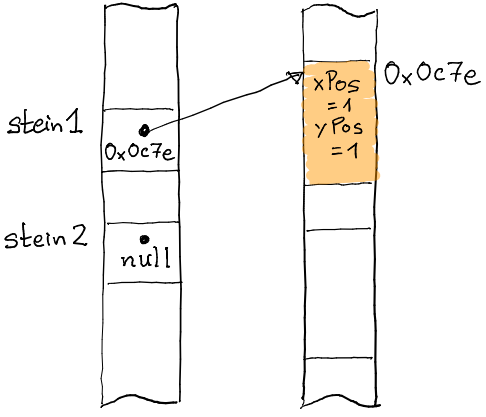


Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

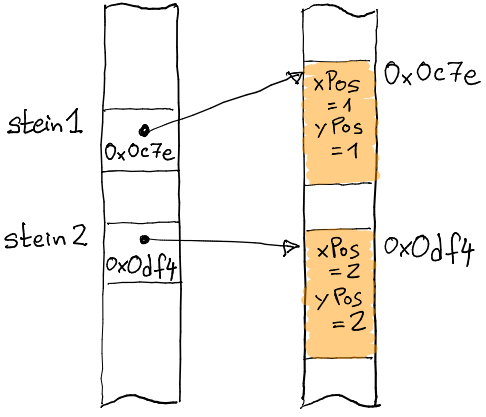


Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

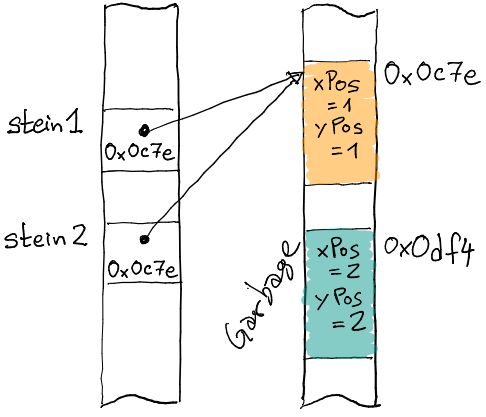


Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```

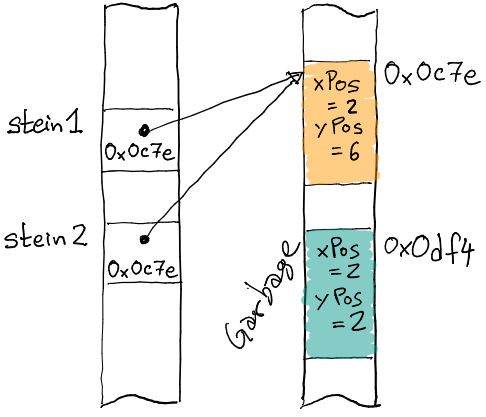


Zuweisungen von Referenzentypen

- Eine Zuweisung mit einem Referenztyp erzeugt keine Kopie des Objektes, sondern nur eine neue Referenz auf das bestehende Objekt.
- Dies ist ein wesentlicher Unterschied von primitiven Datentypen und Referenztypen.

```
Token stein1;
Token stein2;
stein1= new Token(1, 1);
stein2= new Token(2, 2);
stein2= stein1;
stein2.moveTo(2,6);
System.out.println("Stein1: " + stein1);
System.out.println("Stein2: " + stein2);
```

```
> javac Token.java
> java Token
Stein1: (2, 6)
Stein2: (2, 6)
```



Wie funktionierte die Ausgabe von Token?

- Jede Klasse ist eine Unterklasse von **`Object`**.
- Klassen erben die Methoden ihrer Oberklasse (mehr darüber folgt).
- Die Methoden können *überschrieben* werden. Dann ist die überschriebene Methode der Oberklasse immer noch über "**`super.`**" zugänglich.

Wie funktionierte die Ausgabe von Token?

- Jede Klasse ist eine Unterklasse von **`Object`**.
- Klassen erben die Methoden ihrer Oberklasse (mehr darüber folgt).
- Die Methoden können *überschrieben* werden. Dann ist die überschriebene Methode der Oberklasse immer noch über "**`super.`**" zugänglich.
- Die `Object` Klasse implementiert die Methode `toString()` dadurch, dass Klassenname und Speicherplatz ausgegeben werden.
- Die `toString` Methoden werden automatisch beim *casting* zu `String` verwendet.

Wie funktionierte die Ausgabe von Token?

- Jede Klasse ist eine Unterklasse von **`Object`**.
- Klassen erben die Methoden ihrer Oberklasse (mehr darüber folgt).
- Die Methoden können *überschrieben* werden. Dann ist die überschriebene Methode der Oberklasse immer noch über "**`super.`**" zugänglich.
- Die `Object` Klasse implementiert die Methode `toString()` dadurch, dass Klassenname und Speicherplatz ausgegeben werden.
- Die `toString` Methoden werden automatisch beim *casting* zu `String` verwendet.

```
public String toString() {     // Implementation von toString in Token Klasse
  String s = "(" + xPos + ", " + yPos + ")  " + super.toString();
  return s;
}
```

Wie funktionierte die Ausgabe von Token?

- Jede Klasse ist eine Unterklasse von **`Object`**.
- Klassen erben die Methoden ihrer Oberklasse (mehr darüber folgt).
- Die Methoden können *überschrieben* werden. Dann ist die überschriebene Methode der Oberklasse immer noch über "**`super.`**" zugänglich.
- Die `Object` Klasse implementiert die Methode `toString()` dadurch, dass Klassenname und Speicherplatz ausgegeben werden.
- Die `toString` Methoden werden automatisch beim *casting* zu `String` verwendet.

```
public String toString() {     // Implementation von toString in Token Klasse
  String s = "(" + xPos + ", " + yPos + ")  " + super.toString();
  return s;
}
```

```
> javac Token.java
> java Token
Stein1: (2, 6) Token@12bb4df8
Stein2: (2, 6) Token@12bb4df8
```

Garbage Collection

- Durch Zuweisungen von Referenztypen kann ein Objekt im Speicher “verloren” gehen, wenn keine Referenz auf das Objekt mehr existiert (siehe Seite 22).
- Solche Objekte werden als **Garbage** (Müll) bezeichnet, da auf sie nicht mehr zugegriffen werden kann.
- Bei Java läuft im Hintergrund eine **Speicherbereinigung** (*Garbage Collection*), die den Speicherplatz automatisch wieder freigibt.

Rückschau

Nach dieser Vorlesung sollten Sie

- die Geschichte von Java kennen, insbesondere

  das Konzept der JVM mit Java Byte Code
- wissen, wie man kleine Java Programm schreibt, kompiliert und startet
- mit den Hauptunterschieden von C und Java vertraut sein

Rückschau

Nach dieser Vorlesung sollten Sie

- die Geschichte von Java kennen, insbesondere
  das Konzept der JVM mit Java Byte Code
- wissen, wie man kleine Java Programm schreibt, kompiliert und startet
- mit den Hauptunterschieden von C und Java vertraut sein
- die primitiven Datentypen von Java kennen
  und Arrays benutzen können
- das Konzept der objektorientierten Programmierung verstanden haben
- und Unterschied von Klassen- und Instanzvariablen kennen
- passende Sichtbarkeitstypen für Variablen und Methoden wählen können
- sich über die Eigenart von Referenztypen bei Zuweisungen bewusst sein.

Anmeldung in MOSES für die Tutorien

- **Nicht Vergessen:** Bis **heute um 18 Uhr** in MOSES für die Tutorien anmelden.

  Falls dies nicht möglich ist, Email mit Angabe von Matr.Nr., Studiengang/Uni und blockierten Zeitslots an ‹algodat@neuro.tu-berlin.de› schicken.

Anmeldung in MOSES für die Tutorien

- **Nicht Vergessen:** Bis **heute um 18 Uhr** in MOSES für die Tutorien anmelden.

  Falls dies nicht möglich ist, Email mit Angabe von Matr.Nr., Studiengang/Uni und blockierten Zeitslots an ‹algodat@neuro.tu-berlin.de› schicken.

- Anmeldung auf ISIS
- Termin für eingeteiltes Tutorium prüfen, sobald auf ISIS verfügbar

Anmeldung in MOSES für die Tutorien

- **Nicht Vergessen:** Bis **heute um 18 Uhr** in MOSES für die Tutorien anmelden.

  Falls dies nicht möglich ist, Email mit Angabe von Matr.Nr., Studiengang/Uni und blockierten Zeitslots an ‹algodat@neuro.tu-berlin.de› schicken.

- Anmeldung auf ISIS
- Termin für eingeteiltes Tutorium prüfen, sobald auf ISIS verfügbar
- Prüfungsanmeldung möglichst direkt über QISPOS oder siehe ISIS
  An/Abmeldung bis 25.05.2018 möglich

Literatur

- Sedgewick R & Wayne K, **Introduction to Programming in Java: An Interdisciplinary Approach.** 2. Auflage, Addison-Wesley Professional, 2017. Onlinefassung: `https://introcs.cs.princeton.edu/java`
- Ullenboom C, **Java ist auch eine Insel.** 13. Auflage, Rheinwerk Computing, 2018. Onlinefassung: `http://openbook.rheinwerk-verlag.de/javainsel`
- Vornberger O, **Algorithmen. Skript zur Vorlesung im WS 2016/2017.** PDF und HTML unter `http://www-lehre.inf.uos.de/~ainf`

Spezielle Aspekte:

- `https://introcs.cs.princeton.edu/java`, hier insbesondere die Seite `.../faq/c2java.html`
- `https://wiki.byte-welt.net/wiki/Warum_Instanzvariablen_private_deklarieren`
- `https://de.wikibooks.org/wiki/Java_Standard:_Klassen`
- `https://javapapers.com/core-java/java-history`

Danksagung

Außer der angegeben Literatur basieren die Folien dieses Kurses direkt und indirekt auf den Vorlesungen meiner Vorgängerinnen und Vorgänger Oliver Brock, Anja Feldmann und Marc Alexa. Vielen Dank für die Weitergabe des wertvollen Materials und die Erlaubnis, es zu benutzen!

Die Folien wurden mit LaTeX erstellt unter Verwendung vieler Pakete, u.a. beamer, listings, lstbackground, pgffor und colortbl sowie eine Vielzahl von Tipps auf `tex.stackexchange.com` und anderen Internetseiten.

Bildreferenzen

- Seite 8, Google+ Meldung zu YouTube *number of views overflow*: Screenshot von `https://plus.google.com/+YouTube/posts/BUXfdWqu86Q`
- Alle anderen Abbildungen wurden für diese Vorlesung erstellt.

Index