



**Hochschule für Technik  
und Wirtschaft Berlin**

*University of Applied Sciences*

Vorkurs Programmieren

HTW Berlin  
SoSe 2016

# Organisatorisches

- Vorlesung + Übung
  - Mario Neises
  - Mario.neises@student.htw-berlin.de
  - <https://github.com/mn-io> (hello-java)
- Tutorium
  - Laura Laugwitz

# Organisatorisches

- 09:45 – 11:15 Uhr: Vorlesung + Übung
- 11:30 – 13:00 Uhr: Vorlesung + Übung
- 13:45 – 15-15 Uhr: Tutorium
- 15:30 – 17:00: Uhr Selbstlernzeit

# Einführung: Was ist Informatik?

- (Angewandte) Wissenschaft
- Kennen / Auswahl geeigneter Verfahren und Mittel,  
z.B. bei Programmierung, Systemadministration, ...
- Lebenslanges Lernen
- Lernen aus Fehlern

Hinweis:

Gewöhnung an Anfang dauert. Das ist normal

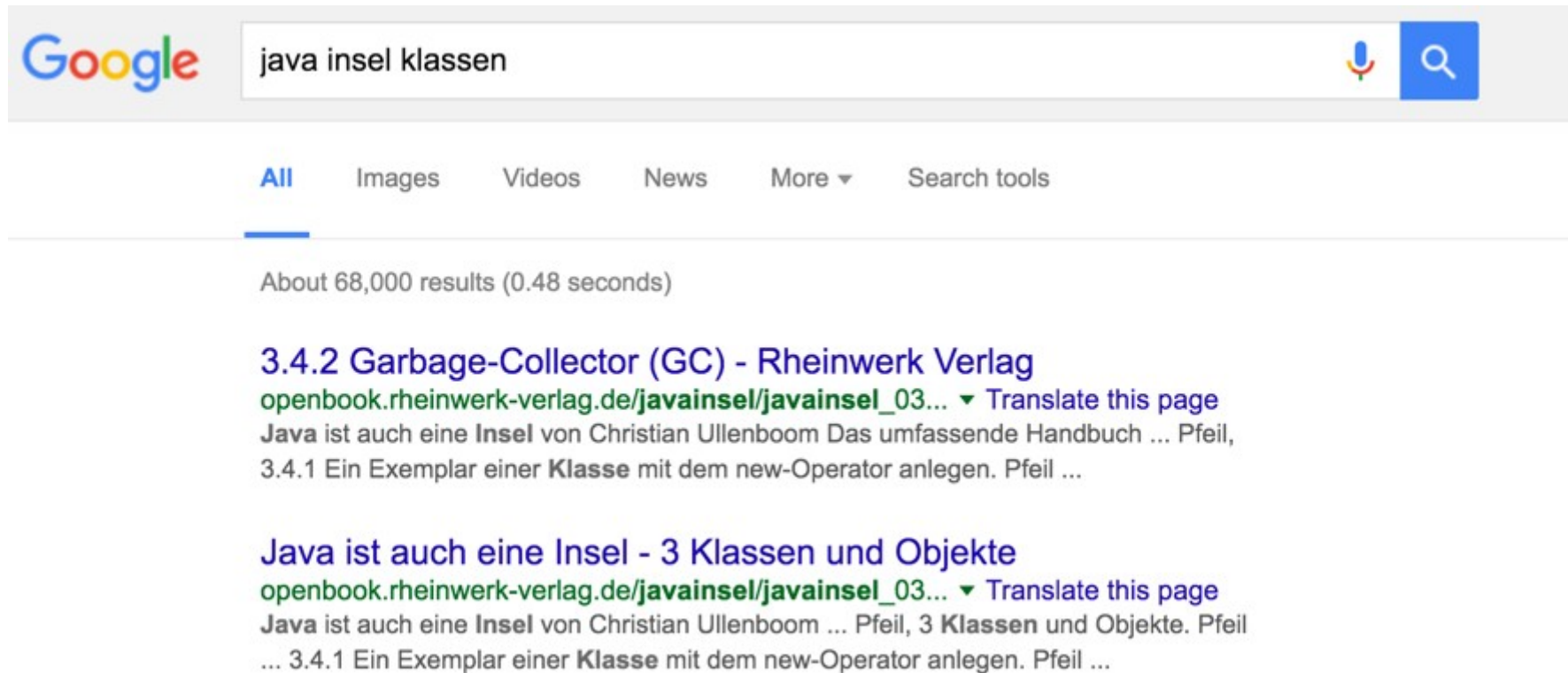
# Einführung: Programmierung

- Ein Handwerk der Informatik
- Testen und spielen: Übung macht den Meister
  - Werkzeuge
  - Programmiersprachen / Frameworks / Patterns (Muster)
  - Try and Error
  - Divide and Conquer

# Einführung: Aufgaben und Probleme

- In der Informatik werden folgende Problemstellungen bearbeitet:
  - Ist ein Problem lösbar?
  - Mit welchem Aufwand ist ein Problem lösbar?
  - Z.B. suchen und finden in Listen, auf Landkarten, ...

# Einführung: Hilfe zur Selbsthilfe



The screenshot shows a Google search interface. The search bar contains the text "java insel klassen". To the right of the search bar are a microphone icon and a blue search button with a magnifying glass. Below the search bar, there are tabs for "All", "Images", "Videos", "News", "More", and "Search tools". The "All" tab is selected and underlined. Below the tabs, it says "About 68,000 results (0.48 seconds)". The first search result is titled "3.4.2 Garbage-Collector (GC) - Rheinwerk Verlag" in blue. Below the title is a green link "openbook.rheinwerk-verlag.de/javainssel/javainssel\_03..." followed by a small downward arrow and the text "Translate this page". The snippet below the link reads: "Java ist auch eine Insel von Christian Ullenboom Das umfassende Handbuch ... Pfeil, 3.4.1 Ein Exemplar einer Klasse mit dem new-Operator anlegen. Pfeil ...". The second search result is titled "Java ist auch eine Insel - 3 Klassen und Objekte" in blue. Below the title is a green link "openbook.rheinwerk-verlag.de/javainssel/javainssel\_03..." followed by a small downward arrow and the text "Translate this page". The snippet below the link reads: "Java ist auch eine Insel von Christian Ullenboom ... Pfeil, 3 Klassen und Objekte. Pfeil ... 3.4.1 Ein Exemplar einer Klasse mit dem new-Operator anlegen. Pfeil ...".

Google

java insel klassen

All Images Videos News More Search tools

About 68,000 results (0.48 seconds)

**3.4.2 Garbage-Collector (GC) - Rheinwerk Verlag**  
[openbook.rheinwerk-verlag.de/javainssel/javainssel\\_03...](https://openbook.rheinwerk-verlag.de/javainssel/javainssel_03...) ▼ Translate this page  
Java ist auch eine Insel von Christian Ullenboom Das umfassende Handbuch ... Pfeil,  
3.4.1 Ein Exemplar einer Klasse mit dem new-Operator anlegen. Pfeil ...

**Java ist auch eine Insel - 3 Klassen und Objekte**  
[openbook.rheinwerk-verlag.de/javainssel/javainssel\\_03...](https://openbook.rheinwerk-verlag.de/javainssel/javainssel_03...) ▼ Translate this page  
Java ist auch eine Insel von Christian Ullenboom ... Pfeil, 3 Klassen und Objekte. Pfeil  
... 3.4.1 Ein Exemplar einer Klasse mit dem new-Operator anlegen. Pfeil ...

# Einführung: Hilfe zur Selbsthilfe

- <http://openbook.rheinwerk-verlag.de/javainsel/>  
(via google: java Insel <?>)
- <https://docs.oracle.com/javase/7/docs/api/>  
(via google: java api <?>)
- <https://stackoverflow.com/>  
(via google: meistens von alleine)



# Einführung: Hilfe zur Selbsthilfe

- Englisch als Hauptsprache
  - mehr Suchergebnisse und Lösungen
  - Fachbegriffe und Fachliteratur
- Unbedingt zu beachten
  - API (Application Programming Interface)  
lesen lernen
  - Konventionen beachten

# Einführung: Vorgehen bei Code-Beispielen

- 1:1 korrekt abschreiben
  - keine Kreativität
  - keine eigenen Ideen
- alle was **rot** ist sind Fehler
  - **sofort** beheben, denn weiter schreiben macht keinen Sinn!
  - oder Code auskommentieren mit //
- wenn das Programm **läuft**:
  - spielen
  - Kreativität einschalten

# Einführung: Vom Code zum Prozess

1) Programmcode schreiben

2) Compilieren / compile:

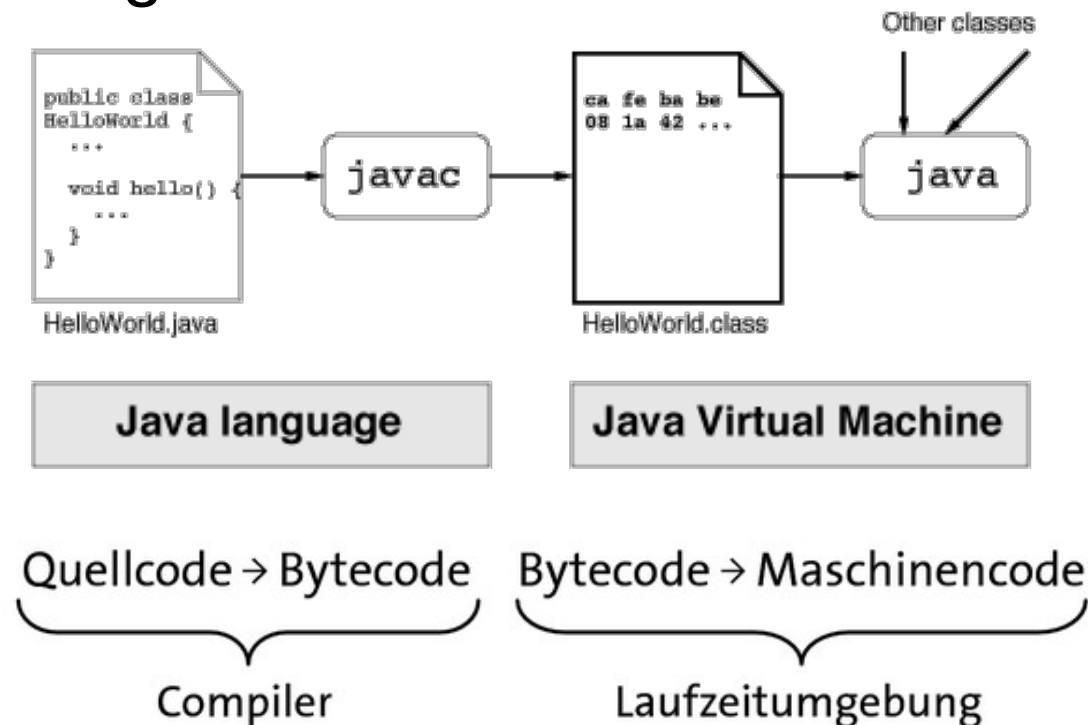
- in Maschinensprache übersetzen

3) Ausführen / execute / run:

- Lädt den Programmcode in den Speicher und führt ihn aus
- Programm (allgemein) wird zum Prozess (konkret)
- Wenn ein Programm läuft: Laufzeit / runtime

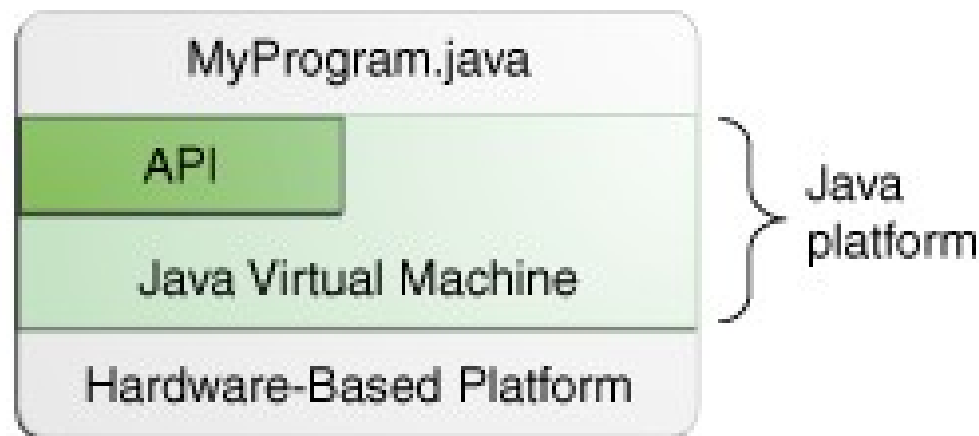
# Einführung: Vom Code zum Prozess

- Compiler: `javac`
  - Macht Programmcode ausführbar
- Programmausführung: `java`
  - Startet Programm als Prozess



# Einführung: Woraus besteht "Java"?

- Java: Programmiersprache, in Version 7 oder 8
- JVM: Java Virtual Machine,  
Hardware-Abstraktion
- JDK, SDK: Software Development Kit
  - Java SE: für uns interessant
  - Java EE: Erweiterung, für Firmenkunden



# Einführung: Zusammenfassung

- Hilfe zur Selbsthilfe
- Handwerk Programmierung mit Werkzeugen
- Abgrenzung: Java, JVM, SDK
- Von Programmcode-Kompilierung zur Programm-Ausführung als Prozess
- Divide and Conquer

# Hello World: Grundaufbau

# Hello World: Grundaufbau

- Konvention: Hello.java – public class Hello
  - Großbuchstabe am Anfang
- Packages: ~ Ordner, Verzeichnisse
  - Trenner: Punkt

```
package folienExamples;
```

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```



# Hello World: Grundaufbau

- Grundgerüst mit Blöcken { }
  - Class-Block
  - Main-Block **innerhalb** des Class-Block
  - Einrückung pro Block zur Lesbarkeit

```
package folienExamples;
```

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

# Hello World: Grundaufbau

- Vorsicht bei Schlüsselwörtern:  
Nicht für eigene Zwecke verwenden,  
Abwandlungen aber OK
- Semikolon am Zeilenende, außer bei Blöcken

```
package folienExamples;
```

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

# Einführung: Grundaufbau

- Grundgerüst bei Java notwendig
  - definiert wie JVM unser Programm starten kann
  - Konvention: `main( )`
- Arbeitsweise innerhalb der unserer Funktionen:
  - jede Zeile (bzw. bis Semikolon) ~ ein Befehl
  - Befehle gepackt in Blöcke `{ }`
  - Befehle verarbeiten Variablen und/oder Funktionen

# Hello World: Grundaufbau (erweitert)

```
package folienExamples;
```

```
public class HelloExtended {
```

```
    public static void main(String[] args) {  
        String hi = "Hello World";  
        System.out.println(hi);  
        int i = 1 + 2;  
        myFunction(i);  
    }
```

```
    public static void myFunction(int i) {  
        System.out.println("i ist: " + i);  
    }
```

```
}
```

# Übung: Werkzeuge

- reiner Text-Editor notwendig  
(kein Microsoft Word oder ähnliches)
- besser: IDE (Integrated Development Environment)
  - IntelliJ Idea: <https://www.jetbrains.com/idea/>

# Übung: Vorgehen

- 1:1 korrekt abschreiben
  - keine Kreativität
  - keine eigenen Ideen
- alle was **rot** ist sind Fehler
  - **sofort** beheben, denn weiter schreiben macht keinen Sinn!
  - oder Code auskommentieren mit //
- wenn das Programm **läuft**:
  - spielen
  - Kreativität einschalten

# Hello World: Zusammenfassung

- Computer sind nicht intelligent
  - daher unbedingt Konventionen beachten
  - keine Kreativität bis das Programm läuft / wir wissen was wir machen
  - Hauptfehler: Klassenname nicht gleich Dateiname
- Arbeitsweise
  - Arbeit in verschiedenen Blöcken: Einrückung
  - Abarbeitung der Befehle von oben nach unten

# Übung 1

- Schreiben Sie ein HelloWorld-Programm, welches den Text "Hello World" ausgibt
  - Nutzen Sie Reformat-Code-Funktion der IDE
  - Benennen Sie ihr Programm um in Hello
- Compilieren Sie das Programm via IDE und Terminal
- Führen Sie das Programm aus via IDE und Terminal
  - Zusatz: Rufen Sie das Programm mit Parametern auf

Terminal-Befehle: `pwd`, `ls`, `cd`, `cat`, `java`, `javac`



# Variablen

- *Ähnlich* wie in der Mathematik
- Variablen mit Namen geben Referenz auf Daten
  - Bsp.: `int i = 5;`
- Deklaration: **`int i = 5;`**
  - Angabe des Datentyps und Namen
  - nur **1x** pro Block möglich
  - notwendig
  - verliert Gültigkeit am Ende des Blockes

# Variablen

- Deklaration: **int i = 5;**
  - Angabe des Datentyps und Namen
  - nur **1x** pro Block möglich
  - notwendig
  - verliert Gültigkeit am Ende des Blockes
- Initialisierung: **int i = 5;**
  - Wertzuweisung von **rechts nach links**  
mathematisch wie  $x := 1$
  - kann weggelassen werden, da sonst Standardwert  
(meist 0, null oder false)

# Variablen

- Initialisierung: `int i = 5;`
  - Wertzuweisung von **rechts nach links**  
mathematisch wie  $x := 1$
  - kann weggelassen werden, da sonst Standardwert (meist 0, null oder false)
- Wertzuweisung:
  - vom gleichen Typ
  - Verwendung der eigenen Variable möglich:  
z.B: `int i = 5;`  
`i = i + 5;`

# Variablen: Primitive Datentypen

- Ganze Zahlen
  - `int` ~ Integer (lat. numerus integer)
  - $2^{32}$  Werte, ~ -2.147.483.648 bis 2.147.483.647
  - Verwendung: siehe vorher
  - `long`:  $2^{64}$  Werte, ~ -/+  $9 \cdot 10^{19}$

# Variablen: Primitive Datentypen

- Reelle Zahlen: mit Dezimalpunkt
  - `float` ~ floating point number, Fließkommazahl
  - `double`
- Achtung: Ungenaue Computerarithmetik
  - Problem:  
z.B.: Brüche wie  $1/3$  bilden unendliche lange Zahl,  
aber kein unendlicher Speicher
- Exkurs:  
[https://de.wikipedia.org/wiki/Liste\\_von\\_Programmfehlerbeispielen](https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen)

# Variablen: Primitive Datentypen

- Buchstaben
  - `char` ~ Character, Zeichen
  - Verwendung: `char c = 'a';`
  - nur 1 Zeichen
- Wahrheitswerte
  - `boolean` ~ boolesche Variable nach George Boole
  - Verwendung: `boolean isSleeping = true;`
  - mögliche Werte: `true`, `false`

# Variablen: Datentypen

- Texte
  - String ~ Zeichenkette/-folge
  - Verwendung: `String hi = "hello";`
  - kein primitiver Datentyp, aber dazu später mehr...

# Variablen: Operationen mit Zahlen

- vgl.: Rechenoperationen

- Zuweisung: = `int i = 5;`
- Vergleich: ==, >=, <= `5 == i; i == 5; (boolean)`
- Addition: + `i + 5; (int)`  
Subtraktion mit - , Multiplikation mit \*
- Division: / `i / 5; (int) Achtung!`
- Modulo: % `i % 5; (int)`

- char-Operationen wie bei Zahlen



# Variablen: Operationen mit boolean

- vgl.: Rechenoperationen

- Zuweisung: = `boolean b = false;`
- Und: &, && `b & true; (boolean)`
- Oder: |, || `b || false; (boolean)`
- Negation: ! `!b; (boolean)`

# Variablen: Operationen mit String

- String arbeitet etwas anders:
  - Zuweisung:           =               String hi = "hello";
  - Konkatenation:   +               "hello" + " world"; (String)

# Variablen

- Unveränderliche Variablen:
  - Konstanten wie `Math.PI`
  - Schlüsselwort: `final`
  - `final double PI = 3.1415;`
- Kurzschreibweisen:

```
int i = 5;
```

```
i+= 2; // i = i + 2;
```

```
i++; // i = i +1;
```

# Variablen: Zusammenfassung

- Repräsentieren unsere Daten
- Zugriff über Namen
- haben einen Datentyp
- können überschrieben werden
- können verändert werden, z.B. mit Operationen
  - Auswertung von rechts nach links
  - Überschreiben der eigenen Variable wie in `i = i + 1;`
  - Kurzschreibweisen wie `i++`, `i+=1;`

# Übung 2

- Schreiben Sie ein Programm `OperationPlayground`, welches
  - eine Zahl auf gerade/ungerade testet
  - das Ergebnis in einer Variable speichert
  - diese Information in einen lesbaren String schreibt und ausgibt
  - Zusatz: wie testen wir Zahlen auf Teilbarkeit durch 3?

Hinweise:

- `psvm + [Tab]` wird zu `public static void main(String...`
- `sout + [Tab]` wird zu `System.out.println();`

# Kontrollstrukturen

- Bisher: Programmfluss linear
  - Befehl nach Befehl, Zeile für Zeile
  - Ausführung je nach Ergebnis: z.B.: isEven

# Kontrollstrukturen: Anweisungen

- Jetzt: Programmfluss beeinflussbar
  - springt je nach Bedingung

```
boolean condition = 5 % 2 == 0;
if(condition) {
    System.out.println("Gerade");
} else {
    System.out.println("Nicht gerade");
}
```

# Kontrollstrukturen: Anweisungen

- Erweiterung: if-elseif-else
  - Was wird ausgegeben?

```
boolean condition = 6 % 2 == 0;
if(condition) {
    System.out.println("Gerade");
} else if (6 % 3 == 0) {
    System.out.println("Durch 3 teilbar");
} else {
    System.out.println("Nicht gerade");
}
```



# Kontrollstrukturen: Anweisungen

- Erweiterung: switch-case
  - (fast) wie if-elseif-else
  - arbeitet jedoch nicht mit boolean

```
switch (int-variable) {  
    case 0:  
        . . .  
        break;  
    case 3:  
        . . .  
        break;  
    default:  
        . . .  
}
```

# Kontrollstrukturen: Anweisungen

```
int i = 3;
switch (i) {
    case 0:
        System.out.println("==0");
        break;
    case 3:
        System.out.println("==3");
        break;
    case 5:
        System.out.println("==5");
        break;
    default:
        System.out.println("...");
}
```

# Kontrollstrukturen: Anweisungen

- Anweisungen: `if`, `switch`
- Fallunterscheidung
- Weitere Kontrolle mit:
  - `return`
  - `break` (nicht bei `if`)

# Kontrollstrukturen: Schleifen

- Programmfluss soll sich bedingt wiederholen
- Schleifen:
  - haben eine Bedingung ob sie weiter ausgeführt werden soll
  - diese Bedingung kann immer wahr sein (sollte aber nicht)
  - i.d.R. eine Zählvariable, welche inkrementiert wird (i++)

# Kontrollstrukturen: Schleifen

- for-Schleife
  - Wann wird `i++` ausgeführt?
  - Wie lange können wir auf `i` zugreifen?

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

# Kontrollstrukturen: Schleifen

- while-Schleife(n)
  - Beenden sich die Schleifen?

```
int i = 0;  
while(i < 10) {  
    System.out.println(i);  
}
```

```
do {  
    System.out.println(i);  
} while(i < 10);
```

# Kontrollstrukturen: Schleifen

- while-Schleife(n)
  - Sind die Schleifen äquivalent?

```
int i = 0;
while(i < 10) {
    System.out.println(i);
    i++;
}
```

```
do {
    System.out.println(i);
    i++;
} while(i < 10);
```

# Kontrollstrukturen: Schleifen

- while-Schleife(n)
  - Sind die Schleifen äquivalent?

```
int i = 0;
while(i < 10) {
    System.out.println(i);
    i++;
}

i = 0;
do {
    System.out.println(i);
    i++;
} while(i < 10);
```



# Konstrollstrukturen: Schleifen

- Alle Schleifen können ineinander überführt werden, sodass sie äquivalent sind.
- for-Schleife:
  - hat eigene Zählvariable
  - Inkrementierung nach Block-Ausführung
- while/do-while:
  - Inkrementierung nicht vergessen
  - Inkrementierung am Anfang / Ende / ... des Blockes

# Kontrollstrukturen: Zusammenfassung

- Arbeiten (meistens) mit Bedingungen (boolean)
- Abgrenzung:
  - Anweisungen: Ausführung ja/nein pro Block
  - Schleifen: Ausführung n-mal pro Block
- for-Schleife
  - `i++` am Ende des Blockes
- Weitere Kontrolle mit
  - `return`
  - `continue`
  - `break`

# Übung 3

- Schreiben Sie ein Programm `LoopPlayground`, welches:
  - die Zahlen 0 bis einschließlich 10 zählt
  - ausgibt, ob die aktuelle Zahl durch 2 und 3 teilbar ist
- Schreiben Sie weitere Programme
  - `ForLoopPlayground`,
  - `WhileLoopPlayground`,
  - `DoWhilePlayground`,welche die jeweilige Schleifenart nutzen

# Kontrollstrukturen: Schleifen

- Inkrementierung nicht zwingend +1
- Code-Block wird n-mal ausgeführt
  - Block muss keinen Bezug zur Zählvariable haben
- Ein Block kann weitere Blöcke enthalten

# Funktionen

- Vgl.: Mathematik  $f(x) = y$
- EVA:
  - Eingabe (von  $x$ , Parameter)
  - Verarbeitung (in  $f$ )
  - Ausgabe (von  $y$ )
- Hinweis: Funktionen haben Klammern ()
- Mehrere Parameter möglich
- Rückgabe mit `return`
- Datentypen für Parameter und Rückgabewert
- Rückgabewert kann in einer Variable gespeichert werden

# Funktionen: Deklaration und Aufruf

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        System.out.println(1+2);  
    }  
}
```

# Funktionen: Deklaration und Aufruf

- neuer Block auf gleicher Ebene wie andere Funktion main()
- ohne Parameter, ohne Rückgabewert

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        add();  
    }  
  
    public static void add() {  
        System.out.println(1+2);  
    }  
}
```

# Funktionen: Deklaration und Aufruf

- mit Parameter, ohne Rückgabewert

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        add(1+2);  
    }  
  
    public static void add(int result) {  
        System.out.println(result);  
    }  
}
```



# Funktionen: Deklaration und Aufruf

- mit mehreren Parameter, ohne Rückgabewert

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        int first = 1;  
        int second = 2;  
        add(first, second);  
    }  
  
    public static void add(int a, int b) {  
        System.out.println(a + b);  
    }  
}
```

# Funktionen: Deklaration und Aufruf

- Übergabe von Variablen direkt oder über Referenz
- Namen werden neu vergeben

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        int first = 1;  
        int second = 2;  
        add(first, second);  
    }  
  
    public static void add(int a, int b) {  
        System.out.println(a + b);  
    }  
}
```

# Funktionen: Deklaration und Aufruf

- mit mehreren Parametern, ohne Rückgabewert

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        int first = 1;  
        int second = 2;  
        add(first, second);  
    }  
  
    public static void add(int a, int b) {  
        System.out.println(a + b);  
        // return;  
    }  
}
```

# Funktionen: Deklaration und Aufruf

- mit mehreren Parametern, mit Rückgabewert

```
public class FunctionPlayground {  
    public static void main(String[] args) {  
        int first = 1;  
        int second = 2;  
        int result = add(first, second);  
        System.out.println(result);  
    }  
  
    public static int add(int a, int b) {  
        // int localResult = a + b;  
        return a + b;  
    }  
}
```

# Funktionen: Zusammenfassung

- Deklaration und Aufruf, beachte Klammern( )
- mit / ohne Parameter
- mit / ohne Rückgabewert: void oder Datentyp
  - sofern Rückgabewert vorhanden:  
kann in Variable gespeichert werden
- Weitere Möglichkeiten:
  - Aufruf von eigener Funktion: rekursiv

# Übung 4

- Schreiben Sie ein Programm Calculator, welches + - \* / und % abdeckt.
  - Jede Rechenoperation soll in einer eigenen Funktion realisiert werden, welche einen geeigneten Datentyp zurückgeben.
  - Das Ergebnis soll erst am Ende ausgegeben werden.
  - Schreiben Sie ein switch-case, welches einen ein char nach der Operation abfragt.
  - Zusatz: Verhindern Sie eine Division durch 0.

# Arrays: Motivation

- Gruppieren von Variablen gleichen Typs

```
public class ArrayPlayground {  
    public static void main(String[] args) {  
        int x = 0;  
        int y = 3;  
        int z = 1;  
        printCoordinates(x, y, z);  
    }  
  
    private static void printCoordinates(int x, int y, int z) {  
        System.out.println("Coords: " + x + "," + y + "," + z);  
    }  
}
```

# Arrays: Verwendung

- Gruppieren von Variablen gleichen Typs
- Lesen / Schreiben via Index [ ]
- Start: 0

```
public class ArrayPlayground {  
    public static void main(String[] args) {  
        int[] coordinate = new int[]{0, 3, 1};  
        // int[] point = new int[3];  
        // point[0] = 3; ...  
        printCoordinates(coordinate);  
    }  
  
    private static void printCoordinates(int[] p) {  
        System.out.println("Coords:" + p[0] + "," + p[1] + "," + p[2]);  
    }  
}
```



# Arrays: Deklaration

- Helfer: Arrays.toString(myArray)
- 2 Arten der Initialisierung
  - Anzahl Elemente fest
  - new-Schlüsselwort

```
public class ArrayPlayground {  
    public static void main(String[] args) {  
        int[] coordinate = new int[]{0, 3, 1};  
        // int[] point = new int[3];  
        // point[0] = 3; ...  
        printCoordinates(coordinate);  
    }  
  
    private static void printCoordinates(int[] p) {  
        System.out.println("Coords:" + p[0] + "," + p[1] + "," + p[2]);  
    }  
}
```

# Übung 5

- Schreiben Sie ein Programm `ArrayForPlayground`, welches das Ergebnis aus `ForLoopPlayground` in einem boolean-Array speichert.
- Schreiben Sie ein Programm `ArraySumPlayground`, welches in einem int-Array alle Zahlen addiert.
- Schreiben Sie ein Programm `ArrayFindPlayground`, welches in einem int-Array die größte Zahl findet .

# Arrays: Zusammenfassung

- Datentyp um Datentypen zu verwalten: Meta-Datentyp
- Zugriff über Index [ ]
- Feste Größe
- Arbeitet gut mit for-Schleife zusammen

# Primitive Datentypen im Speicher

- Variable verhält sich wie direkter Wert:
  - Call-by-value
  - Passen i.d.R. in eine Speicherzelle (heute 64bit)
- Vollständige Kopie bei
  - Neuer Zuweisung
  - Übergabe als Funktionsparameter
- Kopie verändert Original nicht

# Übung 6

- Schreiben Sie ein Programm SwapPlayground, welches
  - den Wert von 2 int-Variablen vertauscht.
  - den Wert von 2 int-Variablen in einem Array vertauscht.
  - Lagern Sie beide Vorgehen in eine Funktion aus.
  - Was beobachten Sie?

# Arrays im Speicher

- Meta-Datentyp verwaltete andere Datentypen
  - Container für mehrere Variablen
  - Alle Variablen vom selben (primitive) Datentyp
- Ist eine Referenz auf interne Daten
  - Call-by-reference
  - schreiben / lesen von internen Daten (mit Index):  
wie bei Call-by-value (wenn primitiver Datentyp)
  - schreiben / lesen auf Array-Variable (ohne Index):  
verändert nur Array-Referenz
  - wenn letzte Referenz überschrieben: Daten im Container gelöscht

# Demo

- Schreiben Sie ein Programm `ReferenzPlayground`, welches die Referenz auf einen Array in unterschiedlichen Variablen speichert.
- Verändern Sie die Array-Einträge unabhängig voneinander.
- Was fällt auf?

# Klassen und Objekte

- Bsp.:
  - Klasse String
  - `String hi = "Hello";`
  - Vgl: `String hi = new String("Hello");`
- Klassen
  - Art Datentyp, neben primitiven Datentypen
  - Beachte: Großschreibung bei Klassennamen
- Objekte
  - Schlüsselwort `new` erzeugt Objekt von Klasse
  - Variablen speichern Objekte
  - Datentyp entspricht der Klasse



# Klassen und Objekte

- Bsp.:
  - `String hi = new String("Hello");`
  - `hi.isEmpty()`
- Objekte
  - können mehrere Variablen beinhalten (vgl. Arrays)
  - können Funktionen besitzen
    - innerhalb Funktionen: Selbstreferenz mit `this`
  - Zugriff über Punkt (s.o.)

# Klassen und Objekte

- Bsp.:
  - `String.valueOf(int)`
- Klassen
  - Bauplan / Vorlage für Objekte
  - legen gemeinsame *statische* Eigenschaften fest:  
Schlüsselwort: static
  - Zugriff über Punkt (s.o.)
  - können auch mehrere Variablen beinhalten
  - können Funktionen besitzen

# Übung 7

- Schreiben Sie ein Programm ClassPlayground, welches
  - Objekte von einer Klasse House erzeugt
  - House hat
    - eine Farbe als String-Variable
    - eine Hausnummer als int-Variable
    - eine toString()-Methode, welche alle Eigenschaften ausgibt
- Was bewirken hinzufügen / auslassen von `static` / `private` / `public` bei Funktionen / Variablen?
- Wieso kennen static-Funktionen kein `this`?

# OOP

- Abgrenzung Funktionsaufruf:
  - Bisher: isEmpty(myString)
  - Jetzt: myString.isEmpty()
- Vorteil:
  - Funktionen (~ Methoden) und Variablen (~ Attribute) sind gruppiert / gekapselt: private
- Was passiert wenn Objekt-Referenz verloren geht, z.B. bei Funktionsende oder Überschreiben?
- Inwiefern grenzt sich der Zugriff zu Klassenfunktionen/variablen ab?

Danke für die Aufmerksamkeit.

Viel Spaß beim Studieren.

Viel Erfolg beim selbstständigen Lernen.