Vorkurs Programmieren

HTW Berlin
SoSe 2016

1.) Wiederholung

1.) Einführung

Programmierung

- Handwerk, kennen der Werkzeuge
- Testen (und spielen)
- Lebenslanges Lernen (aus Fehlern)
- Abwägen: Rechenzeit, Rechenleistung, Speicherplatz

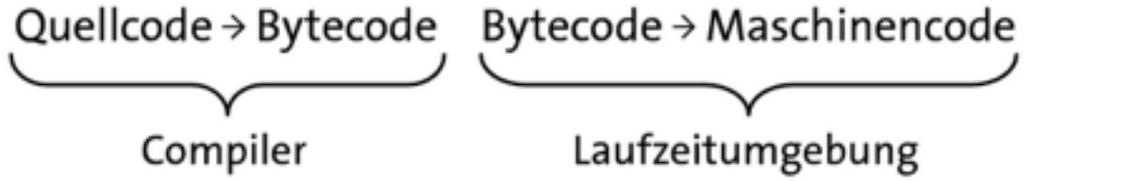
Hilfe zur Selbsthilfe

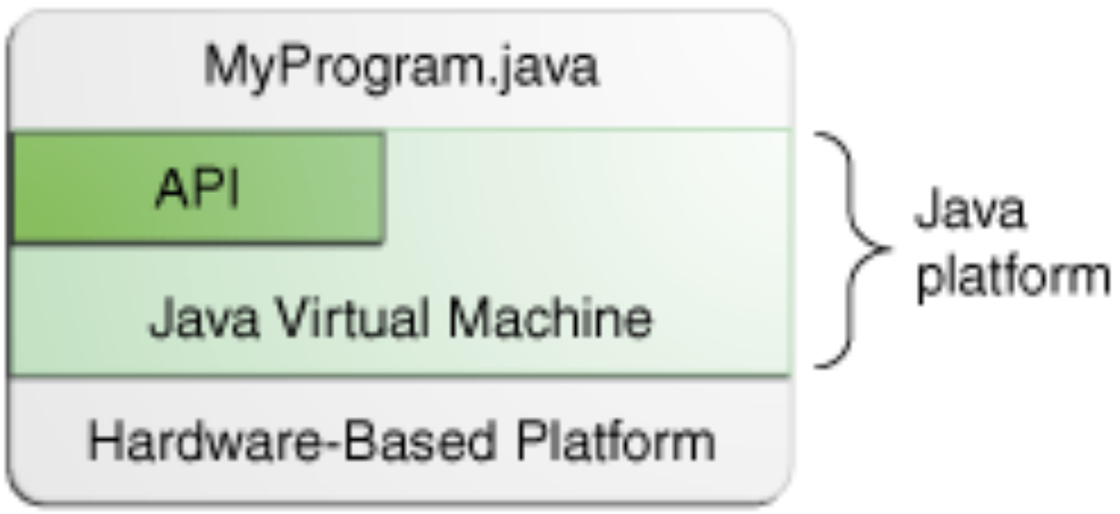
- Google / Java ist auch eine Insel / Java API

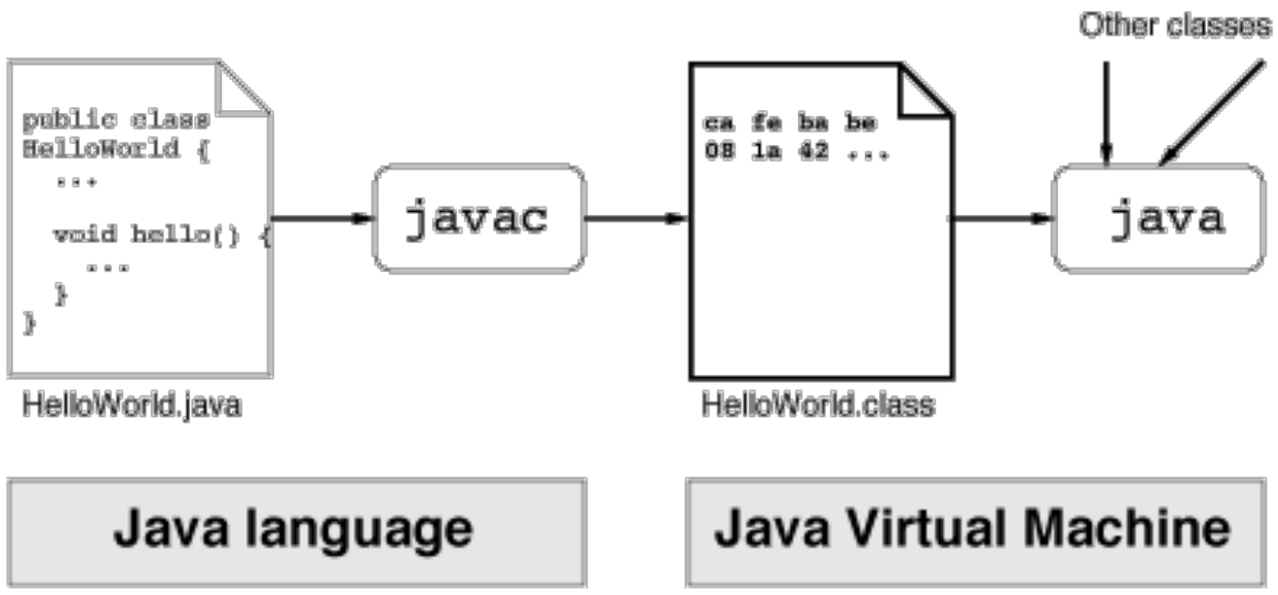
- Konventionen und Paradigma

Vom Code zum Prozess

- Compiler: Übersetzt Programmcode in Maschinensprache
- Laufzeit / Runtime: Programmausführung (Prozess)







Java: Zusammenfassung

- Paradigma: OOP (gut für Modulariserung)
- Garbage Collection
- Plattformunabhängigkeit
- Nicht hardwarenah

1.) Wiederholung

1.) Grundlagen

Hello World

```
public class Hello {

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

Basis-Datentypen

- Zahlen:
- Typen: `int, float, float, double`
- Operatoren: `+ - * / %` mit Kurzschreibweisen: `+=`, `++`
- `final` Konstanten: `Math.PI` (begrenzte Genauigkeit)

Basis-Datentypen

- Buchstaben:
  - Typen: `String, char`
  - Operatoren: `+` (Konkatenation)
- Wahrheitswerte:
  - Typ: `Boolean` mit `TRUE` und `FALSE`

Kontrollstrukturen

- Anweisungen
  - if-else
  - if-elseif-else
  - ? :
  - switch-case
- Schleifen
  - for
  - while
  - do while
- return, continue, break

Funktionen

```
public static void main(String[] args) {
    int hoursToSleep = 4;
    System.out.println(printSleep(hoursToSleep));
}
```

```
private static String printSleep(int hoursToSleep) {
    do {
        hoursToSleep--;
        return hoursToSleep + " more hours to sleep";
    } while(hoursToSleep > 0);
}
```

Deklaration eigener Funktionen mit Rückgabewert

Wie arbeitet ein Programm?

- Folge von Befehlen bzw. von Funktionen / Methoden / Prozeduren (Imperative / Prozeduale Programmierung)
- Verarbeitung von Variablen und Konstanten
- Springen in Schleifen oder je nach Fallunterscheidung
- Code wiederverwenden mittels Funktionen

Inhalt Vorlesung 2

1. ~~Wiederholung~~
2. Mehr Datentypen
    1. Arrays
    2. Referenztypen
3. Klassen
    1. Objekte
    2. Vererbung

2.) Datentypen

1.) Arrays

Bsp.: Viele Variablen mit selben zweck

- Gruppieren sinnvoll
- ggfs. als ein Argument

```
public class Main {
    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("Coordinates: "+x+","+y+","+z);
    }
}
```

Bsp.: Viele Variablen mit selben zweck

- Besser: Arrays
- Gruppiert identische Datentypen
- Lesen / Schreiben mit Index (Start: 0)
- Feste Anzahl Elemente / Länge

```
public class Main {
    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("Coordinates:"+p[0]+","+p[1]+","+p[2]);
    }
}
```

2.) Datentypen

2.) Referenztypen

Bsp.: dynamische Arrays

- Arrays: sind schnell und einfach
- Nicht dynamisch in der Länge

```
public class Main {
    public static void main(String[] args) {
        int[] coordinate = new int[] {0, 3};
        coordinate[2] = 1;
        printCoordinates(coordinate);
    }

    private static void printCoordinates(int[] p) {
        System.out.print("Coordinates:");
        for (int i = 0; i < p.length; i++) {
            System.out.println(p[i] + ", ");
        }
    }
}
```

Bsp.: dynamische Arrays

- ArrayLists sind keine primitiven Datentypen, sondern Objekte

```
public class Main {
    public static void main(String[] args) {
        ArrayList coordinate = new ArrayList();
        coordinate.add(0);
        coordinate.add(3);
        coordinate.add(1);
        printCoordinates(coordinate);
    }

    private static void printCoordinates(ArrayList p) {
        System.out.print("Coordinates:");
        for (int i = 0; i < p.size(); i++) {
            System.out.println(p.get(i) + ", ");
        }
    }
}
```

Bsp.: Finanzwesen

- Berechnungen mit float / double: fehleranfällig
- Besser: BigDecimal

```
public class Main {

    public static void main(String[] args) {
        BigDecimal result = BigDecimal.TEN.add(BigDecimal.ONE);
        System.out.println(result.toString());
    }
}
```

Bsp.: Finanzwesen

- Sind Objekte / Klassen

- Keine arithmetischen Operationen, sondern Funktionsaufrufe

```
public class Main {

    public static void main(String[] args) {
        BigDecimal result = BigDecimal.TEN.add(BigDecimal.ONE);
        System.out.println(result.toString());
    }
}
```

Vergleich

- primitive Datentypen
  - speichern Wert direkt vor Ort (call-by-value)
  - i.d.R. nur so groß wie ein Speicherbereich (heute: 64 bit)

`boolean`	`false`	`true, false`
`char`	'\u0000'	16-Bit-Unicode-Zeichen (0x0000 ... 0xFFFF)
`byte`	0	-2^7 bis 2^7 - 1 (-128 ... 127)
`short`	0	-2^15 bis 2^15 - 1 (-32.768 ... 32.767)
`int`	0	-2^31 bis 2^31 - 1 (-2.147.483.648 ... 2.147.483.647)
`long`	0L	-2^63 bis 2^63 - 1 (-9.223.372.036.854.775.808 ...
`float`	0F	1,40239846E-45f ... 3,40282347E+38f
`double`	0D	4,94065645841246544E-324 ... 1,79769313486231570E+308

- Referenztypen:
  - speichern Referenz auf eigentlichen Werte (call-by-reference)
  - siehe: Array und ArrayList

Vergleich

- primitive Datentypen
  - Kleinschreibung
  - unveränderlich (immutable), konstant, call-by-value
- Referenztypen:
  - Konvention: Großschreibung, CamelCase
  - Klassen und Objekte mit Referenz, call-by-reference
  - un-/ veränderlich

Überblick

- (einige) wichtige Referenztypen
  - BigDecimal / BigInteger
  - String
  - ArrayList / HashMap
- werden mit **`new`** erzeugt
- keine Referenz: **`null`**
- wenn keine Referenz mehr im System: Garbage Collection

String

- String: unveränderlich, Kurzschreibweise: “”
  - int length()
  - char chartAt(int position)
  - boolean isEmpty()
  - String toLowerCase()
  - boolean equals(String otherString)
  - …

Datenstruktur: HashMap

- HashMap: Key-Value-Speicher, Datenstruktur
  - int size()
  - void put(Object key, Object value)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - Object remove(Object key)
  - …

Datenstruktur: ArrayList

- ArrayList: dynamischer Array, Datenstruktur
  - int size()
  - void add(Object element)
  - boolean contains(Object element)
  - Object get(int position)
  - Object remove(int position)
  - …

Problem: Generische Typen

```
public class Main {
    public static void main(String[] args) {
        int[] pointA = new int[3];
        pointA[0] =  3;

        ArrayList pointAL = new ArrayList();
        pointAL.add("3");
        int x = (int) pointAL.get(0);
    }
}
```

- Problem: Arrays wissen, welchen Datentyp sie erlauben, ArrayLists (noch) nicht.

Lösung: Generische Typen

```
public class Main {
    public static void main(String[] args) {
        int[] pointA = new int[3];
        pointA[0] =  3;

       ArrayList<Integer> pointAL = new ArrayList<Integer>();
       pointAL.add("3");
       int x = pointAL.get(0);

    }
}
```

- Lösung: Übergabe des Datentyps (Generics)
- Problem: Was ist `Integer`?

Wrapper-Klassen

- `int`, `boolean`, etc.. sind primitive Datentypen
- OOP-Äquivalent: Wrapper-Klassen
  - notwendig, wenn zwingend Klasse (z.B. verwandt mit `Object`) gebraucht wird
  - unveränderlich, aber call-by-reference
  - kann `null` sein
- Bsp.: (vgl. Konvention!)
  - `new Integer(1)`
  - `Boolean.TRUE`

Datentypen: Zusammenfassung

- Werte vom gleichen Datentyp zusammenfassen zu Arrays
- Datenstrukturen und andere Referenztypen nutzen wie `String`, `BigDecimal` und `ArrayList`
- primitive Datentypen wie `int` nutzen mittels Wrapper-Klassen wie `Integer` bei generischen Datentypen

3.) Klassen

2.) OOP

Objekte und Klassen

- Erlaubt Wiederverwendung, nicht nur von Funktionen

- Idee: Abbildung von reellen Objekten

- vorher: `add(obj1, obj2)`
  OOP: `obj1.add(obj2)`

Objekte und Klassen

- Klassen: Vorlage für neue Objekte
  - z.B.: `ArrayList`, `String`
- Objekt: Ausprägung einer Klasse mit konkreten Werten
  - interne Werte bilden Zustand
  - z.B.: `ArrayList<String> obj = new ArrayList<String>();`

Klassen und Objekte: new

```java
// Vehicle.java
public class Vehicle {
    String color;

    public String toString() {
        return "Color: " + color;
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        vehicle.color = "green";
        System.out.println(vehicle.toString());
    }
}
```

Weitere Aspekte

- Zugriffsregelung für Felder und Methoden:
- `public`, `private`, `protected`, (`package`)
- Felder meistens mit Zugriffsmethoden (getters/setters)

Klassen und Objekte: Objektvariablen (Felder)

```java
// Vehicle.java
public class Vehicle {
    String color;

    public String toString() {
        return "Color: " + color;
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        vehicle.color = "green";
        System.out.println(vehicle~~.toString()~~);
    }
}
```

Klassen und Objekte: Konstruktor

```java
// Vehicle.java
public class Vehicle {
    String color;

    public Vehicle(String color) {
        this.color = color;
    }

    public String toString() {
        return "Color: " + color;
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle("green");
        vehicle.color = "green";
        System.out.println(vehicle);
    }
}
```

Klassen und Objekte: statische Member

```
// Vehicle.java
public class Vehicle {
    private static int totalNumber = 0;
    String color;

    public Vehicle(String color) {
        this.color = color;
        totalNumber++;
    }

    public String toString() {
        return "Color: " + color + ", in total: " + totalNumber;
    }

    public static int getTotalNumber() {
        return totalNumber;
    }
}
```

Weitere Aspekte

- zwingende Konvention: Klassenname.java
- Gruppieren in Packages (Ordner), Packages bilden Ordnerstruktur ab
- Bsp.: `package de.htw;` entspricht Ordner `de/htw/`
- Import von anderen Klassen und Packages

3.) Klassen

2.) Vererbung

Vererbung

- Wiederverwenden von Klassen ist nur ein Vorteil:
- Vererbung: anpassen an andere Bedürfnisse
- Relation: Elternklasse vererbt an Kindklasse
- Alles erbt von **`Object`** (außer primitive Typen)

Vererbung: Eltern- und Kinderklasse

```java
// Vehicle.java
public class Vehicle {
    String color;

    public String toString() {
        return "Color: " + color;
    }
}
```

```java
// Car.java
public class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String color, int numberOfDoors) {
        super(color);
        this.numberOfDoors = numberOfDoors;
    }
    public boolean isFamilyFriendly() {
       return numberOfDoors > 3;
    }
}
```

Vererbung: Eltern- und Kinderklasse

```
// Main.java
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("red", 3);
        System.out.println(myCar.color);
    }
}
```

Klassen: Zusammenfassung

- Klassen (abstrakt, allgemein) und
  Objekte (Instanz einer Klasse, konkret)

- Bekannt aus: Java API (Bsp.: ArrayLists)
  Referenz auf Inhalt

- Organisation in Packages

- Modifier wie `public`, `private`, `protected` und `getters()`/`setters()`

- Vererbung mit Eltern- und Kinderklasse,
  Methoden überschreiben