Hochschule Fulda
University of Applied Sciences

Programmierung 2

JVM, Projekte, Packages

Mai 2025

alexander.gepperth@cs.hs-fulda.de

Java-Grundlagen

Hochsprachen

- Hochsprachen beschreiben Algorithmen unabhängig vom Prozessor/Computer
- Nicht immer komplett möglich (Datentypen, Big/Little-Endianness, Dateisysteme, …)
- Hochsprachen: C, C++, C#, JavaScript, Java, Python, Lisp, Prolog, …
  - lesbar, portabel
  - ggf. langsam

Maschinensprache

- Alle modernen Computer haben eine eigene “Programmier-sprache”, die so genannte **Maschinensprache**
- Maschinensprache wird vom Prozessor verstanden
- Maschinensprache fkt. nur mit best. Prozessor
  - schnell
  - wenig lesbar
  - kaum portabel

DEMO: Maschinencode eines compilierten C-Programms mit `objdump` ausgeben!

Was ist ein Compiler?

- Compiler übersetzen Hochsprachen-Code in Maschinensprache f. einen best. Computer/Prozessor
  - schneller Code
  - Code ist lesbar und portabel

```
#include "stdio.h"

int main(int argc, char** argv) {
  for (int i = 0; i < 10; i++) {
    printf("Hello world (%d)\n", i) ;

    if (i == 9) {
      printf("Done!\n") ;
    }
  }
}
```

```
0000000000001169 <main>:
    1169:               f3 0f 1e fa             endbr64
    116d:               55                      push   %rbp
    116e:               48 89 e5                mov    %rsp,%rbp
    1171:               48 83 ec 20             sub    $0x20,%rsp
    1175:               89 7d ec                mov    %edi,-0x14(%rbp)
    1178:               48 89 75 e0             mov    %rsi,-0x20(%rbp)
    117c:               c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    1183:               eb 32                   jmp    11b7 <main+0x4e>
    1185:               8b 45 fc                mov    -0x4(%rbp),%eax
    1188:               89 c6                   mov    %eax,%esi
    118a:               48 8d 05 73 0e 00 00    lea    0xe73(%rip),%rax
    # 2004 <_IO_stdin_used+0x4>
    1191:               48 89 c7                mov    %rax,%rdi
    1194:               b8 00 00 00 00          mov    $0x0,%eax
    1199:               e8 d2 fe ff ff          call   1070 <printf@plt>
    119e:               83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
    11a2:               75 0f                   jne    11b3 <main+0x4a>
    11a4:               48 8d 05 6b 0e 00 00    lea    0xe6b(%rip),%rax
    # 2016 <_IO_stdin_used+0x16>
    11ab:               48 89 c7                mov    %rax,%rdi
    11ae:               e8 ad fe ff ff          call   1060 <puts@plt>
    11b3:               83 45 fc 01             addl   $0x1,-0x4(%rbp)
    11b7:               83 7d fc 09             cmpl   $0x9,-0x4(%rbp)
    11bb:               7e c8                   jle    1185 <main+0x1c>
    11bd:               b8 00 00 00 00          mov    $0x0,%eax
    11c2:               c9                      leave
    11c3:               c3                      ret
```

Java: Kompilierzeit --vs-- Laufzeit

- Java ist eine Compilersprache
  - **Kompilierzeit**: Java-Compiler `javac` prüft das Programm syntaktisch und semantisch, übersetzt es in Maschinencode (führt nichts aus!!)
  - **Laufzeit**: Java Virtual Machine `java` lädt Maschinencode aus Dateien, führt Programm aus

Demo: Compilieren und Ausführen eines Java-Programms

- **DEMO:** `JavaClass.java` wird in ein `.class`-File kompiliert und ausgeführt (Kommandozeile)
- **DEMO**: selber Vorgang in Eclipse
- **DEMO**: wie sieht erzeugter Java-Maschinencode aus?

Die Java Virtual Machine (JVM)

- Java-Programme werden stets für denselben Computer/Prozessor compiliert: die JVM
- **Java Virtual Machine (JVM)**: Modell eines (simulierten) Computers inklusive Prozessor
- Compilierte Java-Programme laufen auf jedem Rechner, der eine JVM implementiert!

Die Java Virtual Machine (JVM)

- Grund für Benutzung der JVM:
  - Compilierung von Hochsprachen-Programmen auf verschiedenen Rechnern oft problematisch
  - Einfacher, eine JVM zu implementieren als einen Compiler
  - Garantierte Protabilität falls JVM korrekt implementiert ist (kann geprüft werden)

CUT: Q&A

- Vorlesungs-Quiz 1 machen!

Von der Klasse zum Projekt

Von der Klasse zum Projekt

- Manche Probleme erfordern das Zusammenspiel vieler Klassen
- Java definiert Regeln für solche **Projekte**
  - Regeln für `.java` Files
  - Regeln für Packages

Regeln für .java-Files

- Es darf nur eine `public`-Klasse (& beliebig viele nicht-`public`-Klassen) pro `.java`-File geben
- jede Klasse wird in ein `.class`-File compiliert
- `.class`-Files werden per `import` eingebunden
- Kein `import` nötig falls im selben Ordner

```
import java.util.Scanner ;
```

Regeln für .java-Files

- **DEMO:** `JavaClass.java` wird in ein `.class`-File kompiliert und ausgeführt
- **DEMO:** `Test.java` importiert und benutzt `JavaClass`

Regeln für Packages

- Projekte können in **Packages** gegliedert werden
- **Package**: Sammlung thematisch verwandter Klassen
- Packages können Unterpackages enthalten!
- Bekannte Packages: `java.lang`, `java.util`, `java.io`, `java.awt` und `javax.swing`

Packages

- **Demo**: String von der Konsole lesen mit `LargeJavaClass.java`!
- Was macht die Anweisung "`import java.util.Scanner ;`" ?
- Macht die Klasse `Scanner` aus dem **Package** `java.util` verfügbar!

Packages nutzen

voll qualifizierter Klassenname

```
import java.util.Scanner ;
```

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält

Packages nutzen

```
import java.util Scanner ;
```

Package

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält

Packages nutzen

```
import java.util.Scanner ;
```

einfacher Klassenname

- Jede Klasse hat einen **voll qualifizierten Namen**, der das Package und alle Unterpackages enthält

Der import-Mechanismus

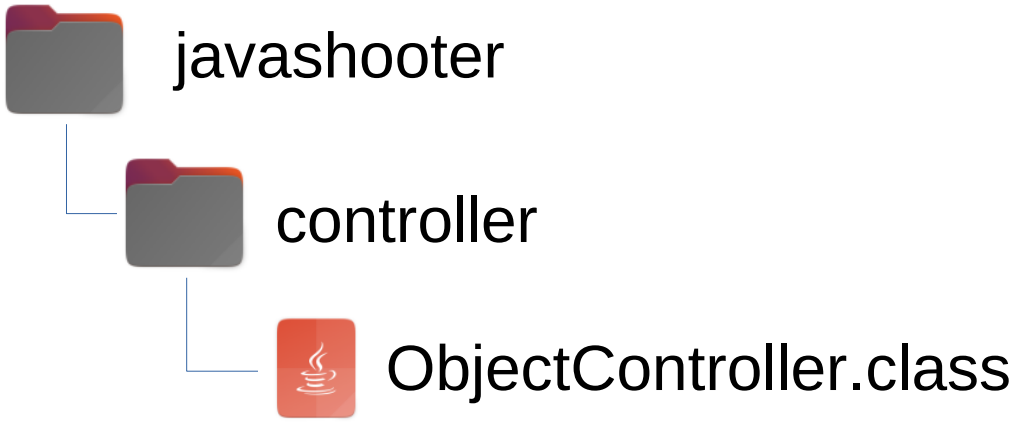
```
import javashooter.controller.ObjectController;
```

- Wozu dient der voll qualifizierte Klassenname?
  - `import` muss Position der `.class`-Datei im Dateisystem kennen, um die Klasse zu nutzen
  - Packages oder Unterpackages: Verzeichnisse im Dateisystem
  - voll qualifizierter Name: bestimmt die Position der `.class`-Datei im Dateisystem

Der import-Mechanismus

```
import javashooter.controller.ObjectController;
```

Packages/Unterpackages definieren Verzeichnisstrukturen!!



CLASSPATH

- CLASSPATH ist ein String, die der Java-Compiler bzw die JVM kennen muss
- Typische Form:

/home/alex:/home/alex/prog2:/home/test

CLASSPATH

- CLASSPATH ist ein String, die der Java-Compiler bzw die JVM kennen muss
- Typische Form:

/home/alex:/home/alex/prog2:/home/test

Der import-Mechanismus und CLASSPATH

```
import javashooter.controller.ObjectController;
```

- Bei diesem `import`-Befehl wird folgendes gemacht:
  - Auswertung von **CLASSPATH**: Liste mit Suchpfaden
  - `import` versucht, in jedem `<suchpfad>` die Datei `<suchpfad>/javashooter/controller/ObjectController.class` zu finden
  - Es wird geprüft, ob das `.class`-File auch eingebunden werden **darf** (public/package-public)

Der import-Mechanismus und CLASSPATH

```
import javashooter.controller.ObjectController;
```

- Vorsicht: CLASSPATH existiert in zwei Ausführungen:
  - zur Kompilierzeit
  - zur Laufzeit
- Muss daher für jedes Projekt 2x festgelegt werden!

Zugriffsrechte für Klassen: public, package-public

- 2 Möglichkeiten für Top-Level Klassen
  - public: jede andere Klasse kann sie importieren

```
public class TestClass { }
```

  - package-public: nur Klassen aus dem selben Package können sie importieren

```
class TestClass { }
```

  - private/protected: später

Eigene Packages erstellen

- Schlüsselwort `package` als erstes Kommando einer `.java`-Datei + voll qualifizierter Package-Name

```
package spaceinvadersProject.controller;
```

- alle Klassen in Datei gehören zu diesem Package
- Compiler weiss wohin er die `.class`-Datei schreiben soll (**WOHIN in diesem Fall??)**

Eclipse-Demo

- Erstellen von Packages, Unterpackages, Klassen kann man in Eclipse automatisieren!

CUT: Q&A

- Vorlesungs-Quiz 2 machen!