Lernzettel

Java-Einführung: Syntax, Typen, Generics, Speicherverhalten und Unit-Tests

Universität: Technische Universität Berlin
Kurs/Modul: Algorithmen und Datenstrukturen

Erstellungsdatum: September 6, 2025

Lernzettel: Java-Einführung: Syntax, Typen, Generics, Speicherverhalten und Unit-Tests

(1) Allgemeine Struktur und Syntax. Java-Programme bestehen aus Klassen, Paketen und Methoden. Die Einstiegsklasse muss eine `public static void main(String[] args)`-Methode enthalten, damit das Programm gestartet wird.

```
bluepackage beispiel;

bluepublic blueclass Hello {
  bluepublic bluestatic bluevoid main(String[] args) {
    System.out.println("Hallo");
  }
}
```

(2) Syntax-Grundbegriffe.

- Groß-/Kleinschreibung ist wichtig (Java ist case-sensitive).
- Klassen- und Methoden-Namenskonventionen: KlassenCamelCase, Methoden/Variablen camelCase.
- Kommentare:
  - Einzeilige Kommentare mit `//`
  - Mehrzeilige Kommentare mit `/* ... */`
- Datentypen werden beim Deklarieren angegeben.

(3) Typen in Java.

- Primitivtypen: `int`, `long`, `short`, `byte`, `char`, `boolean`, `float`, `double`.
- Referenztypen: Klassen, Interfaces, Arrays.
- Wrapper-Typen: `Integer`, `Long`, `Double`, `Boolean`, ...
- Typinferenz: `var` (seit Java 10) erlaubt lokale Typinferenz, z. B. `var list = new ArrayList<String`
- Typumwandlungen (Casting) beachten: explizit `(Type) value`.

(4) Generics. Generics ermöglichen Typensicherheit bei Sammlungen und APIs.

```
blueimport java.util.List;
blueimport java.util.ArrayList;

List<String> names = bluenew ArrayList<>();
names.add("Alice");
String first = names.get(0);
```

(5) Speicherverhalten und Laufzeit-Umgebung.

- Primitive Typen speichern Werte direkt (stack-ähnlich, abhängig vom Kontext).
- Objektinstanzen werden im Heap alloziert; Variablen/Zugriffe auf Objekte halten Referenzen (Zeiger) auf diese Objekte.
- Garbage Collection: automatisches Freigeben ungenutzter Objekte. Typischerweise generational (Young/Old Gen).
- Autoboxing/Unboxing: automatische Umwandlung zwischen Primitivetypen und Wrapper-Typen (z. B. `int` vs. `Integer`).
- Speicher-Overhead durch Objekte (Kopfzeilen, Felder) versus primitive Werte; performanceintensiv durch Boxings.

(6) Unit-Tests (JUnit 5). JUnit dient der Spezifikation und Verifikation von Korrektheit.

```
blueimport org.junit.jupiter.api.Test;
blueimport bluestatic org.junit.jupiter.api.Assertions.*;

blueclass CalculatorTest {

  @Test
  bluevoid add_simple() {
    assertEquals(5, Calculator.add(2, 3));
  }
}
```

(7) Hinweise zu praktischer Anwendung.

- Typensicherheit durch Generics verhindert ClassCastException zur Laufzeit.
- Generics nutzen Typ-Erasure zur Kompatibilität; Laufzeit-Informationen über Generic-Typen gehen verloren.
- Speichermanagement: möglichst wenige unnötige Objekt-Neuerstellungen; verwenden von `StringBuilder` statt wiederholtem String-Konkatenieren.
- Tests sollten zentrale Pfade der Implementierung abdecken (Edge-Cases, Ausnahmefälle, Performance-Profile eher abstrakt).