Lernskript: Große Wiederholung (Kern-APIs & Java 8+ Features)

Dieses Dokument fasst die wichtigsten Konzepte der Meilensteine 2 und 3 zusammen.

Block 1: Kern-APIs (Collections & I/O)

Thema 1.1: Das Java Collections Framework (JCF)

- **Kernidee:** Ein Baukasten aus Interfaces und Klassen zur Verwaltung von Objektgruppen.
- **Die 3 Haupt-Interfaces:**
  - `List`**:** Eine **geordnete** Liste, die **Duplikate erlaubt**.
    - **Use Case:** Einkaufsliste, chronologische Events, Arbeitsschritte.
    - **Implementierungen:** `ArrayList` (schneller Lesezugriff), `LinkedList` (schnelles Einfügen/Löschen in der Mitte).
  - `Set`**:** Eine Menge, die **keine Duplikate** erlaubt.
    - **Use Case:** Eindeutige Benutzer-IDs, besuchte Orte, Tags.
    - **Implementierungen:** `HashSet` (schnell, ungeordnet), `LinkedHashSet` (Einfügereihenfolge), `TreeSet` (sortiert).
  - `Map`**:** Eine Sammlung von **Schlüssel-Wert-Paaren**. Jeder Schlüssel ist einzigartig.
    - **Use Case:** Telefonbuch (Name -> Nummer), Konfiguration (Eigenschaft -> Wert).
    - **Implementierung:** `HashMap` (schnell, ungeordnet).
- **Profi-Tipps (JavaMasta's Vertiefung):**
  - **Modern & Unveränderlich:** Nutze `List.of()`, `Set.of()`, `Map.of()` (seit Java 9) für feste, sichere Collections.
  - **Wichtig:** Für eigene Objekte in `HashSet`/`HashMap` **IMMER** `equals()` und `hashCode()` korrekt überschreiben! `Records` tun dies automatisch.

Thema 1.2: Einfache Dateiein- und -ausgabe (File I/O)

- **Kernidee:** Daten aus Dateien lesen und in sie schreiben.
- **Der moderne Standardweg:**
  - `try-with-resources`**:** Löst das alte Problem des manuellen Schließens von Ressourcen. Alles, was `AutoCloseable` implementiert, wird automatisch geschlossen. Ideal für das zeilenweise Lesen großer Dateien.

```
try (BufferedReader reader = new BufferedReader(new
FileReader("log.txt"))) {
    // ... zeilenweise lesen ...
}
```

  - `Files`**-API (seit Java 11):** Bietet extrem bequeme Methoden für einfache Fälle.

```
// Kleine Datei komplett lesen
String config = Files.readString(Path.of("config.ini"));
// String in eine Datei schreiben (überschreibt)
Files.writeString(Path.of("status.txt"), "OK");
```

- **Profi-Tipps (JavaMasta's Vertiefung):**
  - **Streaming vs. In-Memory:** `BufferedReader` *streamt* Daten (speichereffizient). `Files.readString()` liest *in den Speicher* (nur für kleine Dateien).
  - **Anhängen:** `Files.writeString(path, text, StandardOpenOption.APPEND);`

Block 2: Java 8+ Features (Funktionale Programmierung)

Thema 2.1: Lambdas & Funktionale Interfaces

- **Kernidee:** Kurze, anonyme Funktionen, um Verhalten direkt an Methoden zu übergeben. Ersetzen umständliche anonyme innere Klassen.
- **Voraussetzung:** Ein **Funktionales Interface** (ein Interface mit genau einer abstrakten Methode).
- **Beispiel:** `(s1, s2) -> Integer.compare(s1.length(), s2.length())`
- **Profi-Tipps (JavaMasta's Vertiefung):**
  - **Methodenreferenz `::`:** Noch kürzere Syntax, wenn das Lambda nur eine existierende Methode aufruft (z.B. `System.out::println`).

Thema 2.2: Die Java Stream API

- **Kernidee:** Ein "Fließband" für Daten aus einer Quelle (`List`, etc.), um sie über eine "Pipeline" von Operationen zu verarbeiten.
- **Aufbau einer Pipeline:**
  1. **Quelle:** `.stream()`
  2. **Zwischenoperationen (lazy):** `filter()`, `map()`, `sorted()`, `distinct()`. Geben einen neuen Stream zurück.
  3. **Endoperation:** `collect()`, `count()`, `forEach()`. Startet die Verarbeitung und erzeugt ein Ergebnis.
- **Profi-Tipps (JavaMasta's Vertiefung):**
  - **`flatMap`:** "Bügelt" verschachtelte Streams (z.B. `Stream<List<String>>`) zu einem einzigen, flachen Stream glatt.
  - **Vorsicht bei `parallelStream()`:** Nur für Experten bei rechenintensiven Aufgaben. Standard ist immer `.stream()`.

Thema 2.3: Der `Optional<T>`-Typ

- **Kernidee:** Ein "Container"-Objekt, das einen Wert entweder enthält oder leer ist. Löst das Problem der `NullPointerException`.
- **Anwendung:** Primär als **Rückgabetyp** von Methoden, die möglicherweise kein Ergebnis finden.
- **Der falsche Weg:** `.get()` benutzen.
- **Der richtige Weg (funktionale Methoden):**
  - `ifPresent(Consumer)`

    - `orElse(defaultValue)`
    - `map(Function)`
- **Profi-Tipps (JavaMasta's Vertiefung):**
    - Eine Methode, die eine **Collection** sucht, sollte eine **leere Collection** zurückgeben, kein `Optional<List>`.
    - `orElseGet(Supplier)` ist performanter als `orElse()` bei aufwändiger Erzeugung des Standardwerts.