🟫 Zusammenfassung: Exceptions, Fehlerbehandlung & Ein-/Ausgabe

Wenn Objekte miteinander interagieren, können Fehler auftreten: falsche `Parameter`, fehlende Ressourcen, unerwartete Zustände oder externe Probleme (z. B. Dateien fehlen). Um solche Situationen kontrolliert zu behandeln, stellt Java den `Exception-Mechanismus` bereit.

◇ 1. Exceptions – Fehler kontrolliert behandeln

Eine `Exception` ist ein Objekt, das Informationen über einen Fehler enthält. Sie wird „geworfen" `throw`, wenn etwas schiefläuft.

Typische Fehlerquellen:

- Klient versteht die Fähigkeiten einer `Methode` falsch
- Klient übergibt **ungültige** `Parameter`
- Dienstleister kann eine Aufgabe wegen äusserer Umstände nicht erfüllen (z. B. Datei fehlt)

Ohne Exceptions:

→ Programm bricht ab oder produziert falsche Resultate.

Mit Exceptions:

→ Fehler werden sauber gemeldet und behandelt.

◇ 2. Exception-Typen: geprüft & ungeprüft

✔ Ungeprüfte Exceptions

- Unterklassen von `RuntimeException`
- Compiler verlangt keine Behandlung
- Typische Beispiele:
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `IllegalArgumentException`
- Geeignet für Programmierfehler.

✔ Geprüfte Exceptions

- Müssen mit `try-catch` oder `throws` behandelt werden
- Werden vom Compiler überprüft

- Typische Beispiele:
  - `IOException`
  - `FileNotFoundException`
- Geeignet für Situationen, die ausserhalb der Kontrolle des Programms liegen.

3. Exception-Handler (try-catch)

Ein Codeblock, der möglichen Fehler abfängt und darauf reagiert.

Beispiel:

```
try {
    FileReader fr = new FileReader("daten.txt");
} catch (IOException e) {
    System.out.println("Datei konnte nicht geladen werden!");
}
```

4. Zusicherungen (Assertions)

Dienen dazu, Annahmen im Code zu überprüfen.

Werden hauptsächlich beim Entwickeln aktiviert.

Helfen, Programmierfehler früh sichtbar zu machen.

Beispiel:

```
assert wert >= 0 : "wert darf nicht negativ sein";
```

5. Ein-/Ausgabe `I/O` in Java

`I/O` ist fehleranfällig, weil:

→ Programme in unterschiedlichen Umgebungen laufen

Ressourcen `Dateien`, `Netzwerke`, `Geräte` fehlen können

→ Deshalb treten dort häufig `Exceptions` auf.

Java bietet:

- `Reader` / `Writer` für Text
- `InputStream` / `OutputStream` für Binärdaten
- `java.nio` als moderne Alternative für schnelleres & effizienteres `I/O`

Beispiel:

```
try (BufferedReader br = new BufferedReader(new FileReader("text.txt"))) {
    String zeile = br.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
```

◇ 6. Serialisierung

Serialisierung bedeutet:

→ Ein ganzes `Objekt` (inkl. anderer verknüpfter `Objekte`) lässt sich als Datenstrom speichern und wieder laden.

Voraussetzung:

```
class Kunde implements Serializable { ... }
```

▧ Praktisches Beispiel: Eigene Exception & Handler

```
class UngueltigerWertException extends Exception {
    public UngueltigerWertException(String msg) {
        super(msg);
    }
}

class Konto {
    private double saldo;

    public void abheben(double betrag) throws UngueltigerWertException {
        if (betrag < 0) {
            throw new UngueltigerWertException("Betrag darf nicht negativ sein!");
        }
        if (betrag > saldo) {
            throw new UngueltigerWertException("Nicht genug Geld!");
        }
        saldo -= betrag;
    }
}
```

🧠 Merksätze

`Exceptions` ermöglichen kontrollierte Fehlerbehandlung statt Abstürzen.

Ungeprüft = Programmierfehler;

geprüft = externe Probleme `I/O`, `Netzwerk` usw.

`try-catch` fängt `Exceptions` ab und verhindert Abstürze.

`Assertions` helfen beim Entwickeln, interne Inkonsistenzen zu finden.

`I/O` ist fehleranfällig → `Exceptions` sind dort die Norm.

Serialisierung speichert ganze `Objektstrukturen`.