Section 5: Java Grundlagen & Methoden (Fundamentals & Methods)

5.1 Das erste Java-Programm – Grundstruktur

Die Anatomie einer Java-Datei

Jede Java-Datei hat eine feste Grundstruktur. Eine Klasse, eine `main`-Methode, und darin der ausführbare Code:

```
publicclass FirstClass{
publicstaticvoid main(String[] args){
        System.out.print("Hello, Max");
}
}
```

Wichtige Regeln:

- **Dateiname = Klassenname**: Die Datei `FirstClass.java` muss die Klasse `FirstClass` enthalten
- `public`: Die Klasse ist von überall zugänglich
- `static`: Die Methode gehört zur Klasse selbst, nicht zu einem Objekt (mehr dazu bei OOP)
- `void`: Die Methode gibt keinen Wert zurück
- `String[] args`: Kommandozeilenargumente (ein Array von Strings)

`System.out.print()` vs `System.out.println()`

Methode	Beschreibung	Beispiel
`System.out.print()`	Gibt Text aus, **ohne** Zeilenumbruch	`print("A"); print("B");` → `AB`
`System.out.println()`	Gibt Text aus **mit** Zeilenumbruch	`println("A"); println("B");` → `A\nB`
`System.out.printf()`	Formatierte Ausgabe (wie in C)	`printf("Wert: %d", 42);` → `Wert: 42`

String-Verkettung (Concatenation)

Strings können mit dem `+`-Operator verkettet werden:

```
System.out.println("This is "+
"another "+
"still more.");
// Ausgabe: This is another still more.
```

Auch Zahlen können mit Strings verkettet werden – sie werden automatisch in Strings umgewandelt:

```
int myVariable=50;
System.out.println("myVariable = "+ myVariable);
// Ausgabe: myVariable = 50
```

5.2 Variablen und primitive Datentypen

Was ist eine Variable?

Eine Variable ist ein benannter Speicherplatz im Speicher, der einen Wert enthält. In Java muss jede Variable mit einem **Datentyp deklariert** werden.

```
int topScore=80;// Ganzzahl
double x=20.00d;// Fließkommazahl
boolean isAlien=false;// Wahrheitswert
String makeOfCar="Volkswagen";// Zeichenkette (KEIN primitiver Typ!)
```

Die 8 primitiven Datentypen in Java

Typ	Größe	Wertebereich	Beispiel
`byte`	1 Byte	-128 bis 127	`byte b = 100;`
`short`	2 Bytes	-32.768 bis 32.767	`short s = 30000;`
`int`	4 Bytes	-2.147.483.648 bis 2.147.483.647	`int i = 100000;`
`long`	8 Bytes	-9.223.372.036.854.775.808 bis ...	`long l = 100L;`
`float`	4 Bytes	~7 Dezimalstellen	`float f = 3.14f;`
`double`	8 Bytes	~15 Dezimalstellen	`double d = 3.14d;`
`char`	2 Bytes	Ein Unicode-Zeichen	`char c = 'A';`
`boolean`	1 Bit*	`true` oder `false`	`boolean b = true;`

> *Hinweis: `boolean` ist formal 1 Bit, wird aber in der JVM oft als 1 Byte gespeichert.

Wichtige Unterschiede:

- **`int` vs `long`**: `long` für Zahlen die nicht in `int` passen. Suffix `L` verwenden: `long big = 9999999999L;`
- **`float` vs `double`**: `double` ist der Standard für Fließkommazahlen. `float` braucht Suffix `f`: `float f = 3.14f;`
- **`String` ist KEIN primitiver Typ** – es ist eine Klasse (Referenztyp). Deshalb wird `String` großgeschrieben.

Typumwandlung (Type Casting)

```
// Implizite Konvertierung (Widening) - automatisch, kein Datenverlust
int myInt=100;
long myLong= myInt;// int → long automatisch
double myDouble= myLong;// long → double automatisch

// Explizite Konvertierung (Narrowing) - manuell, möglicher Datenverlust
double price=9.99;
long rounded=(long)(price*1000);// double → long, Nachkommastellen werden abge
schnitten!
```

Inkrement und Dekrement

```
int myVariable=50;
myVariable++;// myVariable ist jetzt 51 (Inkrement um 1)
myVariable--;// myVariable ist jetzt 50 (Dekrement um 1)
```

5.3 Operatoren

Arithmetische Operatoren

Operator	Beschreibung	Beispiel	Ergebnis
`+`	Addition	`5 + 3`	`8`
`-`	Subtraktion	`5 - 3`	`2`
`*`	Multiplikation	`5 * 3`	`15`
`/`	Division	`5 / 3`	`1` (Ganzzahldivision!)
`%`	Modulo (Rest)	`5 % 3`	`2`

Achtung bei Ganzzahldivision:

```
int a=5/3;// Ergebnis: 1 (nicht 1.666...)
double b=5.0/3;// Ergebnis: 1.6666...
```

Der Modulo-Operator `%` ist besonders nützlich:

```
double sum=(20.00+80.00)*100;// = 10000.0
double theRemainder= sum%40.00d;// = 0.0 (kein Rest)
```

Typische Anwendungen von Modulo:

- Prüfen ob eine Zahl gerade ist: `number % 2 == 0`
- Prüfen ob eine Zahl durch X teilbar ist: `number % X == 0`
- Sekunden aus Gesamtsekunden extrahieren: `totalSeconds % 60`
- Minuten aus Gesamtminuten: `totalMinutes % 60`

Vergleichsoperatoren

Operator	Bedeutung	Beispiel
`==`	Gleich	`score == 100`
`!=`	Ungleich	`score != 0`
`>`	Größer als	`score > 50`
`<`	Kleiner als	`score < 100`
`>=`	Größer oder gleich	`score >= 80`
`<=`	Kleiner oder gleich	`score <= 95`

Logische Operatoren

Operator	Bedeutung	Beispiel
`&&`	UND (AND)	`a > 0 && b > 0` – beide müssen wahr sein
`||`	ODER (OR)	`a > 0 || b > 0` – mindestens eines muss wahr sein

Operator	Bedeutung	Beispiel
`!`	NICHT (NOT)	`!isAlien` – kehrt den Wert um

Beispiele aus dem Kurs:

```
int topScore=80;
int secondTopScore=95;

// UND: Beide Bedingungen müssen wahr sein
if((topScore> secondTopScore)&&(topScore<100)){
    System.out.println("Greater than second and less than 100");
}

// ODER: Mindestens eine Bedingung muss wahr sein
if((topScore>90)||(secondTopScore<=90)){
    System.out.println("Either or both of the conditions are true");
}

// NICHT: Umkehrung eines boolean-Wertes
boolean isCar=false;
if(!isCar){
    System.out.println("isCar ist false, also !isCar ist true");
}
```

`Math` Klasse – Nützliche mathematische Methoden

```
Math.round(3.7)// → 4 (Runden)
Math.round(3.14)// → 3
Math.PI// → 3.141592653589793
Math.PI*(r* r)// Kreisfläche berechnen
```

5.4 Statements, Whitespace und Einrückung

Was ist ein Statement?

Ein **Statement** ist eine vollständige Anweisung in Java, die mit einem Semikolon `;` endet:

```
int myVariable=50;// Das ist ein Statement
myVariable++;// Das ist ein Statement
System.out.println("Test");// Das ist ein Statement
```

Whitespace

Java ignoriert zusätzliche Leerzeichen und Zeilenumbrüche. Der Compiler sieht keinen Unterschied zwischen:

```
int x=5;
```

und:

```
int x=5;
```

Aber für die **Lesbarkeit** ist gute Formatierung essentiell.

Einrückung (Indentation)

Einrückung zeigt die logische Struktur des Codes:

```
if(myVariable==0){
    System.out.println("It's now zero");// 4 Leerzeichen eingerückt
}
```

Regel: Alles innerhalb von `{ }` wird um eine Ebene eingerückt (üblicherweise 4 Leerzeichen oder 1 Tab).

5.5 Ausdrücke und Schlüsselwörter (Expressions & Keywords)

Was ist ein Ausdruck (Expression)?

Ein **Ausdruck** ist alles, was einen Wert ergibt:

```
double kilometers=(100*1.609344);// "100 * 1.609344" ist ein Ausdruck
int highScore=50;// "50" ist ein Ausdruck
highScore=1000+ highScore;// "1000 + highScore" ist ein Ausdruck
```

Schlüsselwörter (Keywords)

Schlüsselwörter sind reservierte Wörter in Java, die eine besondere Bedeutung haben und **nicht als Variablennamen** verwendet werden dürfen:

`int`, `double`, `boolean`, `if`, `else`, `for`, `while`, `return`, `void`, `public`, `static`, `class`, `new`, `true`, `false`, `null`, ...

Es gibt insgesamt **ca. 50 reservierte Schlüsselwörter** in Java.

5.6 Die if-Anweisung und Code-Blöcke

Die `if` Anweisung

Die `if`-Anweisung ist die grundlegendste Kontrollstruktur. Sie führt Code nur aus, wenn eine Bedingung `true` ist.

```
int topScore=80;
if(topScore>=100){
    System.out.println("You got the high score!");
}
```

`if-else` und `if-else if-else`

```
int score=800;

if(score<5000&& score>1000){
    System.out.println("Score zwischen 1000 und 5000");
```

```
}elseif(score<1000){
    System.out.println("Score unter 1000");
}else{
    System.out.println("Score ist 5000 oder mehr");
}
```

Ablauf: Java prüft die Bedingungen von oben nach unten. Sobald eine Bedingung `true` ist, wird der zugehörige Block ausgeführt und alle weiteren `else if` / `else` werden übersprungen.

Code-Blöcke `{ }`

Ein **Code-Block** fasst mehrere Statements zusammen:

```
if(isAlien==false){
    System.out.println("It is not an alien!");// Statement 1
    System.out.println("And I'm scared of aliens");// Statement 2
}
```

Wichtig: Ohne geschweifte Klammern gilt nur das **nächste** Statement als zur `if`-Anweisung gehörig:

```
if(true)
    System.out.println("Nur dieses wird bedingt ausgeführt");
    System.out.println("Dieses wird IMMER ausgeführt!");// FEHLERQUELLE!
```

Best Practice: Immer geschweifte Klammern verwenden, auch bei nur einem Statement.

Verschachtelte if-Anweisungen

Man kann `if`-Anweisungen ineinander verschachteln. Ein gutes Beispiel ist die Schaltjahr-Berechnung:

```
publicstaticboolean isLeapYear(int year){
if(year<1|| year>9999){
returnfalse;// Ungültiger Bereich
}

if(year%4==0){// Durch 4 teilbar?
if(year%100==0){// Durch 100 teilbar?
if(year%400==0){// Durch 400 teilbar?
returntrue;// → Schaltjahr (z.B. 2000)
}else{
returnfalse;// → KEIN Schaltjahr (z.B. 1900)
}
}else{
returntrue;// → Schaltjahr (z.B. 2024)
}
}else{
returnfalse;// → KEIN Schaltjahr (z.B. 2025)
}
}
```

Schaltjahr-Regeln:

1. Durch 4 teilbar → mögliches Schaltjahr
2. Aber: Durch 100 teilbar → kein Schaltjahr
3. Aber: Durch 400 teilbar → doch ein Schaltjahr

Beispiele: 2024 ✓, 1900 ✗, 2000 ✓, 2025 ✗

5.7 Der Ternäre Operator (Conditional Operator)

Syntax

```
Ergebnis=(Bedingung)? WertWennTrue: WertWennFalse;
```

Der ternäre Operator ist eine **Kurzform** für einfache `if-else`-Anweisungen.

Beispiele aus dem Kurs:

```
// Statt:
boolean isDomestic;
if(makeOfCar=="Volkswagen"){
    isDomestic=false;
}else{
    isDomestic=true;
}

// Kann man schreiben:
boolean isDomestic= makeOfCar=="Volkswagen"?false:true;

// Weiteres Beispiel:
int ageOfClient=20;
String ageText= ageOfClient>=18?"Over 18":"Under 18";
System.out.println("Our client is "+ ageText);
// Ausgabe: Our client is Over 18

// Direkt in einer Zuweisung:
String s=(isDomestic)?"This car is domestic":"This car is not domestic";
```

Wann den ternären Operator verwenden?

- **Ja:** Für einfache Zuweisungen basierend auf einer Bedingung
- **Nein:** Für komplexe Logik oder mehrere Anweisungen → lieber `if-else` verwenden

```
// Gut: Einfache Zuweisung
int max= summer?45:35;

// Schlecht: Zu komplex für ternären Operator
// Lieber if-else verwenden wenn die Logik komplex wird
```

5.8 Methoden (Methods)

Was ist eine Methode?

Eine **Methode** ist ein benannter Block von Code, der eine bestimmte Aufgabe ausführt. Methoden helfen, Code zu **organisieren**, **wiederzuverwenden** und **lesbarer** zu machen.

Methoden-Deklaration

```
publicstatic Rückgabetyp methodenName(Parametertyp parameter1, Parametertyp pa
rameter2){
// Methodenkörper
return wert;// nur wenn Rückgabetyp nicht void ist
}
```

Methode OHNE Rückgabewert (`void`)

```
publicstaticvoid displayHighScorePosition(String playerName,int highScorePosit
ion){
    System.out.println(playerName+" managed to get into position "
+ highScorePosition+" on the high score list");
}
```

- `void` bedeutet: Die Methode gibt **nichts** zurück
- Sie führt nur eine Aktion aus (z.B. etwas auf der Konsole ausgeben)
- `return;` kann ohne Wert verwendet werden, um die Methode vorzeitig zu beenden

Methode MIT Rückgabewert

```
publicstaticint calculateHighScorePosition(int playerScore){
int position=4;// Standardposition
if(playerScore>=1000){
        position=1;
}elseif(playerScore>=500){
        position=2;
}elseif(playerScore>=100){
        position=3;
}
return position;// Gibt die berechnete Position zurück
}
```

- Der Rückgabetyp (`int`) muss zum Typ des `return` Wertes passen
- Der Rückgabewert kann in einer Variable gespeichert oder direkt verwendet werden:

```
// In Variable speichern:
int highScorePosition= calculateHighScorePosition(1500);

// Direkt verwenden:
System.out.println("Position: "+ calculateHighScorePosition(1500));
```

Verschiedene Rückgabetypen

```
// int zurückgeben
publicstaticint calculateScore(boolean gameOver,int score,int levelCompleted,i
nt bonus){
int finalScore= score;
if(gameOver){
        finalScore+=(levelCompleted* bonus);
        finalScore+=1000;
}
return finalScore;
}

// boolean zurückgeben
publicstaticboolean isLeapYear(int year){
// ... Logik ...
returntrue;// oder false
}

// double zurückgeben
publicstaticdouble area(double radius){
if(radius<0){
return-1;// Fehlercode
}
return Math.PI*(radius* radius);
}

// String zurückgeben
publicstatic String getDurationString(int minutes,int seconds){
return hours+"h "+ remainingMinutes+"m "+ seconds+"s";
}

// long zurückgeben
publicstaticlong toMilesPerHour(double kilometersPerHour){
if(kilometersPerHour<0){
return-1;
}
return Math.round(kilometersPerHour/1.609);
}
```

Vorzeitiges Beenden einer Methode mit `return`

`return` kann auch in `void`-Methoden verwendet werden, um die Methode vorzeitig zu beenden:

```
publicstaticvoid printMegaBytesAndKiloBytes(int kiloBytes){
if(kiloBytes<0){
        System.out.println("Invalid Value");
return;// Methode wird hier sofort beendet!
}

int megaBytes= kiloBytes/1024;
int remainder= kiloBytes%1024;
```

```
    System.out.println(kiloBytes+" KB = "+ megaBytes+" MB and "+ remainder+" K
B");
}
```

Parameter vs Argumente

- **Parameter**: Die Variablen in der Methodendeklaration (formale Parameter)
- **Argumente**: Die konkreten Werte beim Methodenaufruf (aktuelle Parameter)

```
// "playerScore" ist der PARAMETER
publicstaticint calculateHighScorePosition(int playerScore){ ...}

// "1500" ist das ARGUMENT
calculateHighScorePosition(1500);
```

Methoden aufrufen und kombinieren

Methoden können andere Methoden aufrufen:

```
publicstaticvoid main(String[] args){
int highScorePosition= calculateHighScorePosition(1500);// Methode 1 aufrufen
    displayHighScorePosition("Max", highScorePosition);// Methode 2 aufrufen
}
```

Eine Methode kann sich auch **selbst** als Hilfsmethode nutzen (Delegation):

```
publicstatic String getDurationString(int seconds){
if(seconds<0){
return"Invalid data";
}
// Delegiert an die überladene Version mit 2 Parametern:
return getDurationString(seconds/60, seconds%60);
}
```

5.9 Method Overloading (Methodenüberladung)

Was ist Methodenüberladung?

Methodenüberladung bedeutet, dass mehrere Methoden **den gleichen Namen** haben, sich aber in ihren **Parametern unterscheiden** (Anzahl, Typ oder Reihenfolge der Parameter).

Regeln für Überladung:

1. **Gleicher Methodenname**
2. **Unterschiedliche Parameterliste** (mindestens einer dieser Unterschiede):
    - Unterschiedliche Anzahl der Parameter
    - Unterschiedliche Typen der Parameter
    - Unterschiedliche Reihenfolge der Typen

3. **Der Rückgabetyp allein reicht NICHT aus** um Methoden zu unterscheiden!

Beispiel 1: calculateScore mit verschiedener Parameteranzahl

```
// Version 1: Mit Name und Score
publicstaticint calculateScore(String playerName,int score){
    System.out.println("Player "+ playerName+" scored "+ score+" points");
return score*1000;
}

// Version 2: Nur mit Score (ruft Version 1 auf)
publicstaticint calculateScore(int score){
return calculateScore("Anonymous", score);// Delegiert an Version 1!
}

// Version 3: Ohne Parameter
publicstaticint calculateScore(){
    System.out.println("No player name, no player score");
return0;
}
```

Aufruf:

```
calculateScore("Max",500);// Ruft Version 1 auf
calculateScore(10);// Ruft Version 2 auf
calculateScore();// Ruft Version 3 auf
```

Beispiel 2: Flächenberechnung mit verschiedenen Parametertypen

```
// Kreis: Ein Parameter (Radius)
publicstaticdouble area(double radius){
if(radius<0)return-1;
return Math.PI*(radius* radius);
}

// Rechteck: Zwei Parameter (Länge, Breite)
publicstaticdouble area(double x,double y){
if(x<0|| y<0)return-1;
return x* y;
}
```

Beispiel 3: Zentimeter-Konvertierung

```
// Version 1: Nur Inches
publicstaticdouble convertToCentimeters(int heightInches){
return heightInches*2.54;
}

// Version 2: Feet und Inches (ruft Version 1 auf)
```

```
publicstaticdouble convertToCentimeters(int heightFeet,int remainingInches){
int feetToInches= heightFeet*12;
int totalInches= feetToInches+ remainingInches;
double result= convertToCentimeters(totalInches);// Delegation an Version 1!
return result;
}
```

Was geht NICHT?

```
// FEHLER! Gleiche Parameter, nur anderer Rückgabetyp → Compile Error
publicstaticint calculateScore(){return0;}
publicstaticvoid calculateScore(){ System.out.println("...");}
// → "method calculateScore() is already defined"
```

Vorteile der Methodenüberladung:

1. **Einheitlicher Name**: Der Aufrufer muss sich nicht verschiedene Namen merken
2. **Flexibilität**: Verschiedene Varianten für verschiedene Eingaben
3. **Code-Wiederverwendung durch Delegation**: Eine überladene Methode kann eine andere aufrufen
4. **Lesbarkeit**: `area(5.0)` und `area(5.0, 4.0)` sind beide intuitiv verständlich

---

5.10 Eingabevalidierung (Input Validation)

Warum validieren?

Methoden sollten ungültige Eingaben erkennen und **sauber behandeln**, anstatt falsche Ergebnisse zu liefern oder abzustürzen.

Muster 1: Rückgabe eines Fehlerwerts

```
publicstaticlong toMilesPerHour(double kilometersPerHour){
if(kilometersPerHour<0){
return-1;// -1 als Fehlercode
}
return Math.round(kilometersPerHour/1.609);
}
```

Muster 2: Frühzeitiges Beenden mit `return`

```
publicstaticvoid printMegaBytesAndKiloBytes(int kiloBytes){
if(kiloBytes<0){
        System.out.println("Invalid Value");
return;// Beende sofort, kein weiterer Code wird ausgeführt
}
// ... normaler Code ...
}
```

Muster 3: Bereichsvalidierung

```
publicstaticboolean shouldWakeUp(boolean bark,int hourOfDay){
if(hourOfDay<0|| hourOfDay>23){
returnfalse;// Ungültige Stunde
}
return bark&&(hourOfDay<8|| hourOfDay>22);
}
```

Muster 4: Rückgabe eines Fehler-Strings

```
publicstatic String getDurationString(int seconds){
if(seconds<0){
return"Invalid data of seconds ("+ seconds+"), must be a positive integer valu
e";
}
return getDurationString(seconds/60, seconds%60);
}
```

5.11 Praktische Programmierkonzepte aus den Übungen

Konzept: Konstanten mit `static final`

Statt "magic numbers" im Code zu verwenden, definiere Konstanten:

```
privatestaticfinalint MIN_PER_HOUR=60;
privatestaticfinalint HOURS_PER_DAY=24;
privatestaticfinalint DAYS_PER_YEAR=365;
privatestaticfinalint MIN_PER_DAY= HOURS_PER_DAY* MIN_PER_HOUR;
privatestaticfinalint MIN_PER_YEAR= MIN_PER_HOUR* HOURS_PER_DAY* DAYS_PER_YEA
R;
```

Vorteile:

- Code ist lesbarer: `minutes / MIN_PER_YEAR` statt `minutes / 525600`
- Änderungen müssen nur an einer Stelle gemacht werden
- Konstanten werden in `UPPER_SNAKE_CASE` geschrieben

Konzept: Ganzzahldivision und Modulo für Zeitberechnungen

Ein sehr häufiges Muster in der Programmierung:

```
publicstaticvoid printYearsAndDays(long minutes){
long days= minutes/(60*24);// Gesamtminuten → Tage
long years= days/365;// Tage → Jahre
long remainderDays= days%365;// Resttage

// 525600 min = 1 y and 0 d
// 561600 min = 1 y and 25 d
}
```

Gleiches Muster für Stunden/Minuten/Sekunden:

```
int hours= minutes/60;
int remainingMinutes= minutes%60;
// Wenn minutes = 125: hours = 2, remainingMinutes = 5
```

Konzept: Dezimalvergleich mit Typ-Casting

Fließkommazahlen auf Gleichheit zu prüfen ist problematisch. Eine Lösung: Auf eine bestimmte Anzahl Dezimalstellen vergleichen:

```
publicstaticboolean areEqualByThreeDecimalPlaces(double number1,double number
2){
long number1Rounded=(long)(number1*1000);// Abschneiden (nicht runden!)
long number2Rounded=(long)(number2*1000);
return number1Rounded== number2Rounded;
}
// areEqualByThreeDecimalPlaces(3.175, 3.176) → false
// areEqualByThreeDecimalPlaces(3.0, 3.0) → true
```

Wichtig: `(long)` schneidet ab, rundet nicht! `(long)(3.999) = 3`

Konzept: Boolean-Ausdrücke direkt zurückgeben

Statt:

```
if(a+ b== c){
returntrue;
}
returnfalse;
```

Besser:

```
return a+ b== c;// Der Vergleich ergibt direkt true oder false
```

Weiteres Beispiel:

```
// Statt:
publicstaticboolean isTeen(int x){
if(x>=13&& x<=19){
returntrue;
}
returnfalse;
}

// Besser:
publicstaticboolean isTeen(int x){
return(x>=13&& x<=19);
}
```

Konzept: Komplexe boolesche Logik

```
publicstaticboolean shouldWakeUp(boolean bark,int hourOfDay){
// Der Hund bellt UND es ist vor 8 Uhr ODER nach 22 Uhr
return bark&&(hourOfDay<8|| hourOfDay>22);
}
```

Die Klammern sind hier wichtig! Ohne sie würde `&&` vor `||` ausgewertet:

- `bark && hourOfDay < 8 || hourOfDay > 22` → falsche Auswertung!
- `bark && (hourOfDay < 8 || hourOfDay > 22)` → korrekt!

Konzept: Mehrere Bedingungen für Klassifizierung

```
publicstaticvoid printEqual(int a,int b,int c){
if(a<0|| b<0|| c<0){
        System.out.println("Invalid Value");
}elseif(a== b&& b== c){
        System.out.println("All numbers are equal");
}elseif((a== b)||(a== c)||(b== c)){
        System.out.println("Neither all are equal or different");
}else{
        System.out.println("All numbers are different");
}
}
```

5.12 Zusammenfassung der wichtigsten Konzepte

Konzept	Beschreibung
Klasse & main	Jedes Java-Programm braucht eine Klasse mit `public static void main(String[] args)`
Datentypen	8 primitive Typen (`int`, `double`, `boolean`, etc.) + Referenztypen (`String`)
Operatoren	Arithmetisch (`+`, `-`, `*`, `/`, `%`), Vergleich (`==`, `!=`, `<`, `>`), Logisch (`&&`, `||`, `!`)
if-else	Bedingte Ausführung von Code-Blöcken
Ternärer Operator	Kurzform: `bedingung ? wertTrue : wertFalse`
Methoden	Wiederverwendbare Code-Blöcke mit Namen, Parametern und optionalem Rückgabewert
Methodenüberladung	Gleicher Name, verschiedene Parameter → verschiedene Versionen
Eingabevalidierung	Ungültige Eingaben früh prüfen und behandeln
Modulo `%`	Rest einer Division – essentiell für Zeitberechnungen und Teilbarkeitsprüfungen
Typ-Casting	Explizite `(typ)` oder implizite Typumwandlung
Konstanten	`static final` für unveränderliche Werte mit sprechenden Namen

Dieser Konspekt basiert auf dem Code und den Übungen der Section 5 des Java-Kurses.