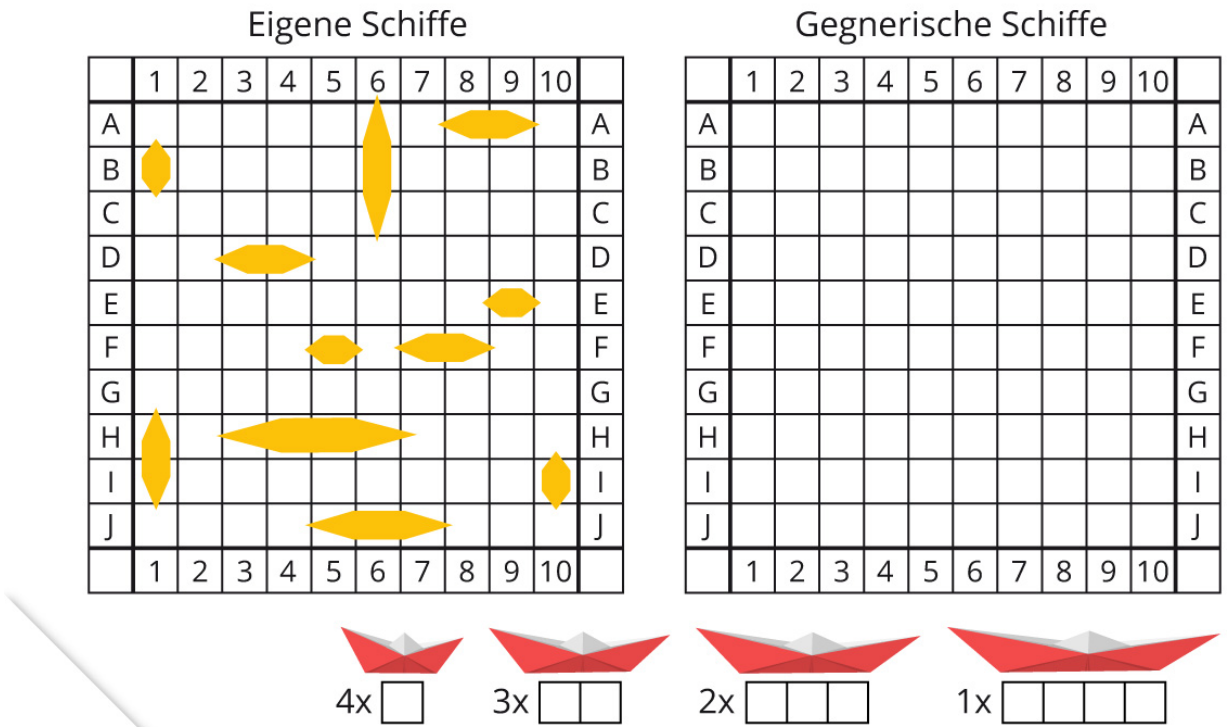
Schiffe versenken – Wiederholung Grundlagen

18. November 2024

Plan für die Woche

Montag	Dienstag	Mittwoch	Donnerstag	Freitag
• Syntax und Grundelemente • Kontrollstrukturen • Klassen und Objekte	• Arrays • ArrayList • Umgang mit großen Aufgabenstellungen	• Einführung in UML • Erste Grundlagen OOP	• Konsolen Ein- und Ausgabe	• Strategien Fehlererkennung und Debugging

Projekt: Schiffe versenken



https://www.talu.de/schiffe-versenken/

Plan für heute

- Syntax und Grundelemente
- Kontrollstrukturen
- Klassen und Objekte

Syntax und Grundelemente

Datentypen (data types)

- Java ist **statisch typisiert:**
  - Variablen, Ergebnisse von Ausrücken etc. haben einen Datentyp
  - Steht bei Kompilierung fest
- zwei Typen von Datentypen
  - Primitive Datentypen (primitive data types)
  - Referenzdatentypen (reference data types)

Primitive Datentypen

Was ist die Ausgabe?

```
int a = 5;
a = 7;
String a = "Hallo";
System.out.println(a);
```

a) Compile Fehler

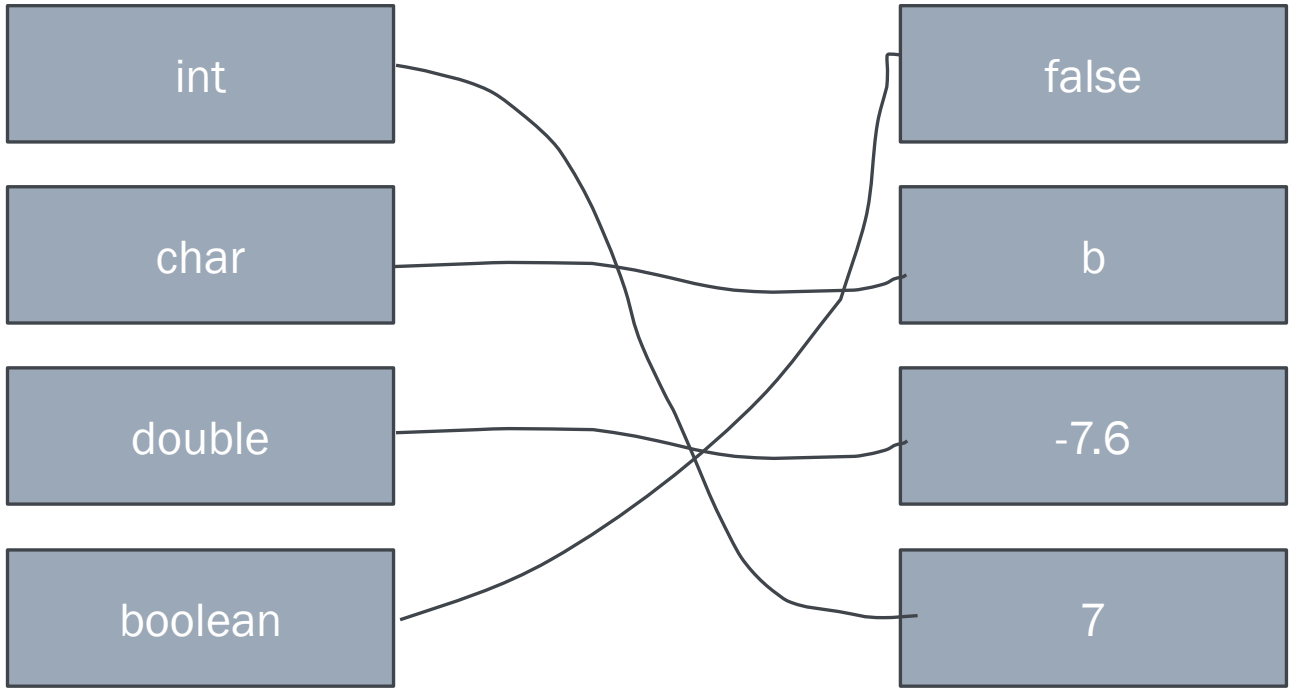
b) 7

c) 5

d) „Hallo“

Primitive Datentypen

Ordne dem primitiven Datentyp das passende Beispiel zu:



Primitive Datentypen

- Vier ganzzahlige Typen (integer type)
  - **byte**
  - **short**
  - **int**
  - **long**
- Zwei Gleitkommazahlen (floating point)
  - **float**
  - **double**
- Wahrheitswerte: **boolean**
- Zeichen: **char**

Primitive Datentypen

Primitive Datentypen

ganze Zahlen			
byte	1 Byte	ganze Zahlen	-2^7 bis 2^7-1 (-128 ... 127)
short	2 Byte	ganze Zahlen	-2^{15} bis $2^{15}-1$ (-32768 .. 32767)
int	4 Byte	ganze Zahlen	-2^{31} bis $2^{31}-1$ (ca. 2 Mrd.)
long	8 Byte	ganze Zahlen	-2^{63} bis $2^{63}-1$
Fliesskommazahlen (Dezimalzahlen, gebrochene Zahlen)			
float	4 Byte	Fliesskommazahlen	mit einfacher Genauigkeit
double	8 Byte	Fliesskommazahlen	mit doppelter Genauigkeit
Wahrheitswerte			
boolean	1 Byte	Wahrheitswerte	true oder false
Zeichen (‚character ‘)			
char	2 Byte	Zeichen	Unicode

1 Byte = 8 Bit

0	**1**	**1**	**0**	**1**	**1**	**1**	**0**

https://oinf.ch/kurs/programmieren/variablen/

Referenzdatentypen

- Objekte
- Zeichenketten (**String**)
- Arrays

Variablen (variables)

- Speicherung von Daten während der Laufzeit (Ausführung)
- Die Daten können jederzeit verändert werden
  - Durch Zuweisung anderer Daten
  - Durch Berechnung mit verändertem Ergebnis dieser Variable

- **Basis der Datenspeicherung und Datenverarbeitung**

Variablen (variables)

Pass by Value (Wertübergabe):

Eine **Kopie** des Wertes der Variable wird an die Methode übergeben. Änderungen an der Kopie **beeinflussen nicht** die Originalvariable.

-> Von Java genutzt

```java
public static void main (String args[]) {
    //Aufgabe 2: Was ist die Ausgabe?

    int num = 5;
    changeValue(num);
    System.out.println(num);

}

private static void changeValue(int x) {  1 usage
    x = 10;
}
```

Variablen (variables)

Pass by Reference (Referenzübergabe):

Eine Referenz (Speicheradresse) auf die Originalvariable wird an die Methode übergeben. Änderungen an der Referenz **beeinflussen die Originalvariable.**

```java
public static void main (String args[]) {

    StringBuilder text = new StringBuilder("Hello");
    modifyObject(text);
    System.out.println(text); // Ausgabe: Hello World

}

static void modifyObject(StringBuilder sb) {  1 usage
    sb.append(" World");
}
```

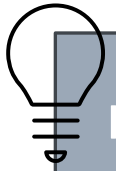
Variablen (variables)

Deklaration (declaration)

```
int x;  no usages
String str;  no usages
Object obj;  no usages
```

Deklaration und Initialisierung (initialisation)

```
int y = 10;  no usages
String hello = "Hello";  no usages
Object object = obj;  no usages
```



Info: Ein **Literal** in Java ist ein fester Wert, der direkt im Code geschrieben wird. Literale repräsentieren **konkrete Werte** für Variablen und Konstanten und werden beim Kompilieren in den Speicher geschrieben.

Variablen (variables)

```java
//Aufgabe 3: Was ist die Ausgabe?
class TestClass  {
    static int i;  no usages
    int y;  no usages

    public static void main (String args[]){
        int p;
        System.out.println(p);
    }
}
```

a) Compile Fehler

b) p

c) int

d) i

Variablen (variables)

final

Definition unveränderlicher Variablen.

```java
//Aufgabe 4: Was ist die Ausgabe?
class TestClassFinalVariables{
    final int x = 10;  2 usages
    final static int y = 20;  no usages

    public static void main (String args[]){
        x = 15;
        System.out.println(x);
    }

}
```

static

- Arbeiten auf Klassenebene statt auf Instanzebene
- Statische **Variablen**:
  - Gehört zur Klasse nicht zu Objekten
- Statische **Methoden**:
  - Können ohne ein Objekt zu erstellen über die Klasse abgerufen werden
  - Haben keinen Zugriff auf Instanzvariablen oder -methoden, außer über ein Objekt.
- Statische **Blöcke**:
  - Ein statischer Block wird einmal beim Laden der Klasse ausgeführt und dient zur Initialisierung statischer Variablen.

static

Was gehört zusammen?

```
class Counter {  no usages
    static int count = 0;
    Counter() {  no usages
        count++;
    }
}
```

Statische Variable

Statische Methode

```
class Utility {  1 usage
    static int add(int a, int b) {  1 us
        return a + b;
    }
}
int result = Utility.add( a: 5,  b: 3);
```

```
class Config {  no usages
    static String setting;  1 usage
    static {
        setting = "Standardwert";
    }
}
```

Statischer Block

Konstanten (constant)

- **Unveränderliche** Variable, deren Wert nach der Initialisierung nicht mehr geändert werden kann
- Häufig verwendet, um feste Werte wie PI, Maximalgrößen oder Konfigurationsdaten zu speichern
- Benamung i.d.R. in GROSSBUCHSTABEN und Unterstrich geschrieben
- mit **static**: Die Konstante gehört zur Klasse und wird nicht für jede Instanz separat gespeichert.
- mit **final**: Der Wert der Variablen kann nach der Zuweisung nicht mehr geändert werden.

Konstanten (constant)

```java
public class Circle {  1 usage
    public static final double PI = 3.14159;  1 usage

    public double calculateCircumference(double radius) {  1 usage
        return 2 * PI * radius;
    }
}
```

Operatoren (operators)

- Arithmetische Operatoren (arithmetic operators)
- Zuweisungsoperatoren (assignment operators)
- Vergleichsoperatoren (relational operators)
- Logische Operatoren (logical/conditional operators)
- Inkrement- und Dekrement-Operatoren (increment/decrement operators)
- Bitweise Operatoren (bitwise operators)

Operatoren (operators)

Arithmetische Operatoren (arithmetic operators)

Operator	Beschreibung	Kurzbeispiel
+	Addition	`int antwort = 40 + 2;`
-	Subtraktion	`int antwort = 48 - 6;`
*	Multiplikation	`int antwort = 2 * 21;`
/	Division	`int antwort = 84 / 2;`
%	Teilerrest, Modulo-Operation, errechnet den Rest einer Division	`int antwort = 99 % 57;`
+	positives Vorzeichen, in der Regel überflüssig	`int j = +3;`
-	negatives Vorzeichen	`int minusJ = -j;`

Operatoren (operators)

Zuweisungsoperatoren (assignment operators)

Operator	Beschreibung	Kurzbeispiel
=	einfache Zuweisung	`int var = 7;`
+=	Addiert einen Wert zu der angegebenen Variablen	`plusZwei += 2;`
-=	Subtrahiert einen Wert von der angegebenen Variablen	`minusZwei -= 2;`
/=	Dividiert die Variable durch den angegebenen Wert und weist ihn zu	`viertel /= 4;`
*=	Multipliziert die Variable mit dem angegebenen Wert und weist ihn zu	`vierfach *= 4;`
%=	Ermittelt den Modulo einer Variablen und weist ihn der Variablen zu	`restModulo11 %= 11;`
&=	"und"-Zuweisung	`maskiert &= bitmaske;`
|=	"oder"-Zuweisung	
^=	"exklusives oder"-Zuweisung	
^=	bitweise "exklusive oder"-Zuweisung	
>>=	Rechtsverschiebungzuweisung	
>>>=	Rechtsverschiebungzuweisung mit Auffüllung von Nullen	
<<=	Linksverschiebungzuweisung	`achtfach <<= 3;`

Operatoren (operators)

Vergleichsoperatoren (relational operators)

Das Ergebnis dieser Operationen ist aus der Menge `true`, `false`:

Operator	Beschreibung	Kurzbeispiel
==	gleich	`3 == 3`
!=	ungleich	`4 != 3`
>	größer als	`4 > 3`
<	kleiner als	`-4 < -3`
>=	größer als oder gleich	`3 >= 3`
<=	kleiner als oder gleich	`-4 <= 4`

Operatoren (operators)

Logische Operatoren (logical/conditional operators)

Operator	Beschreibung	Kurzbeispiel
!	Negation, invertiert den Ausdruck	`boolean lügnerSpricht = !wahrheit;`
&&	Und, `true`, genau dann wenn alle Argumente `true` sind	`boolean krümelmonster = istBlau && magKekse;`
||	or `true`, wenn *mindestens* ein Operand `true` ist	`boolean machePause = hungrig || durstig;`
^	Xor `true` wenn genau ein Operand `true` ist	`boolean zustandPhilosoph = denkt ^ ist;`

Operatoren (operators)

Inkrement- und Dekrement-Operatoren (increment/decrement operators)

Operator	Beschreibung	Kurzbeispiel
++	Postinkrement, Addiert 1 zu einer numerischen Variablen	`x++;`
++	Preinkrement, Addiert 1 zu einer numerischen Variablen	`++x;`
--	Postdekrement, Subtrahiert 1 von einer numerischen Variablen	`x--;`
--	Predekrement, Subtrahiert 1 von einer numerischen Variablen	`--x;`

Operatoren (operators)

Bitweise Operatoren (bitwise operators)

Operator	Beschreibung	Kurzbeispiel
~	(unäre) invertiert alle Bits seines Operanden	`0b10111011 = ~0b01000100`
&	bitweises "und", wenn beide Operanden 1 sind, wird ebenfalls eine 1 produziert, ansonsten eine 0	`0b10111011 = 0b10111111 & 0b11111011`
|	bitweises "oder", produziert eine 1, sobald einer seiner Operanden eine 1 ist	`0b10111011 = 0b10001000 | 0b00111011`
^	bitweises "exklusives oder", wenn beide Operanden den gleichen Wert haben, wird eine 0 produziert, ansonsten eine 1	`0b10111011 = 0b10001100 ^ 0b00110111`

Operator	Beschreibung	Kurzbeispiel
>>	Arithmetischer Rechtsshift: Rechtsverschiebung, alle Bits des Operanden werden um eine Stelle nach rechts verschoben, stand ganz links eine 1 wird mit einer 1 aufgefüllt, bei 0 wird mit 0 aufgefüllt	`0b11101110 = 0b10111011 >> 2`
>>>	Logischer Rechtsshift: Rechtsverschiebung mit Auffüllung von Nullen	`0b00101110 = 0b01011101 >>> 1`
<<	Linksverschiebung, entspricht bei positiven ganzen Zahlen einer Multiplikation mit 2, sofern keine "1" rausgeschoben wird.	`0b10111010 = 0b01011101 << 1`

Operatoren (operators)

Rangfolge	Typ	Operatoren
1	Postfix-Operatoren, Postinkrement, Postdekrement	`x++`, `x--`
2	Einstellige (unäre) Operatoren, Vorzeichen	`++x`, `--x`, `+x`, `-x`, `~b`, `!b`
3	Multiplikation, Teilerrest	`a*b`, `a/b`, `a % b`
4	Addition, Subtraktion	`a + b`, `a - b`
5	Bitverschiebung	`d << k`, `d >> k`, `d >>> k`
6	Vergleiche	`a < b`, `a > b`, `a <= b`, `a >= b`, `s instanceof S`
7	Gleich, Ungleich	`a == b`, `a != b`
8	UND (Bits)	`b & c`
9	Exor (Bits)	`b ^ c`
10	ODER (Bits)	`b | c`
11	Logisch UND	`B && C`
12	Logisch ODER	`B || C`
13	Bedingungsoperator	`a ? b : c`
14	Zuweisungen	`a = b`, `a += 3`, `a -= 3`, `a *= 3`, `a /= 3`, `a %= 3`, `b &= c`, `b ^= c`, `b |= c`, `d <<=k`, `d >>= k`, `d >>>= k`

Kontrollstrukturen

if-Statement

Die if-Anweisung überprüft eine Bedingung, und wenn diese true ist, wird der nachfolgende Block ausgeführt.

```
if(Bedingung) {
        // Code, der ausgeführt wird, wenn die Bedingung wahr ist }
    }
```

if-else-Statement

Mit if-else kann ein alternativer Codeblock ausführt werden, wenn die Bedingung false ist.

```
if (Bedingung) {
    // Code, wenn die Bedingung wahr ist
} else {
    // Code, wenn die Bedingung falsch ist
}
```

else-if-Statement

Manchmal gibt es mehrere Bedingungen, die überprüft werden sollen. Dafür wird eine else if-Anweisung verwendet.

```
if (Bedingung1) {
 // Code, wenn Bedingung1 wahr ist
 } else if (Bedingung2) {
// Code, wenn Bedingung1 falsch, aber Bedingung2 wahr ist
 } else {
// Code, wenn keine Bedingung wahr ist
 }
```

Bedingungsoperator

Ternärer Operator (ternary conditional operator) ?:

- Verkürzung des if-else-Statements

Bedingung ? Ausdruck1 : Ausdruck2;

-> Ausdruck1: Falls die Bedingung true

-> Ausdruck2: Falls die Bedingung false

Vergleich if-else

```
if (Bedingung) {
    // Code, wenn die Bedingung wahr ist
} else {
    // Code, wenn die Bedingung falsch ist
}
```

```
Bedingung ? Ausdruck1 : Ausdruck2;
```

Bedingungsoperator

```java
//Was ist die Ausgabe?
int num = 5;
String result = (num > 0) ? "Positiv" : "Negativ oder Null";
System.out.println(result);
```

```java
//Was ist die Ausgabe?
int a = 10, b = 20;
int max = (a > b) ? a : b;
System.out.println("Das Maximum ist: " + max);
```

while-Schleife (while-loop)

Die while-Schleife führt ihren Codeblock aus, solange die angegebene Bedingung true ist. Die Bedingung wird vor der Ausführung geprüft, weshalb es eine kopfgesteuerte Schleife ist.

```
while (Bedingung) {
    // Code, der wiederholt ausgeführt wird
}
```

do-while-Schleife (do-while loop)

Die do-while-Schleife garantiert, dass der Schleifeninhalt mindestens einmal ausgeführt wird, da die Bedingung erst nach der Ausführung geprüft wird. Sie ist eine fußgesteuerte Schleife.

```
do {
    // Code, der mindestens einmal ausgeführt wird
} while (Bedingung);
```

for-Schleife (for loop)

Die for-Schleife ist kompakter und eignet sich besonders für Schleifen mit bekannten Iterationen (z. B. Zählvorgänge).

```
for (Initialisierung; Bedingung; Aktualisierung) {
    // Code, der wiederholt ausgeführt wird
}
```

foreach-Schleifen (enhanced for loop)

Die foreach-Schleife wird verwendet, um über Elemente einer Sammlung wie Arrays oder Objekten, die eine Iterable-Schnittstelle implementieren, zu **iterieren**. Sie ist kompakter und lesbarer als eine klassische for-Schleife, insbesondere bei Iterationen über Listen oder Arrays.

```
for (Datentyp element : Sammlung) {
    // Code, der für jedes Element ausgeführt wird
}
```

Vergleich der Schleifen

Schleife	Wann verwendet?
for	Für bekannte Iterationen oder Zählerbasierte Schleifen.
foreach	Wenn über Sammlungen oder Arrays iteriert werden soll, ohne Indizes.
while	Wenn die Anzahl der Iterationen nicht bekannt ist, aber eine Bedingung gegeben ist.
do-while	Wenn der Code mindestens einmal ausgeführt werden muss.

Vergleich der Schleifen

Szenario	Schleifen-Typ
Eine Schleife soll 10-mal ausgeführt werden.	for
Die Schleife soll so lange laufen, bis der Benutzer "q" eingibt.	while
Eine Aktion soll mindestens einmal ausgeführt werden, bevor geprüft wird, ob die Bedingung zutrifft.	do-while

Unendliche Schleifen

```java
while (true) {
    System.out.println("Unendliche Schleife!");
}
```

```java
for (;;) {
    System.out.println("Unendliche Schleife!");
}
```

Aufgabe

Schreibe eine Methode, die einen Boolean übergeben bekommt und folgendes kann:

- **true**:
  - Berechnung der Summe aller natürlichen Zahlen bis 50
  - Ausgabe des Ergebnisses in der Konsole
- **false**:
  - Berechnung des Produkts aller natürlichen Zahlen bis 20 (also 20!)
  - Wenn das Ergebnis gerade durch 2 teilbar ist und größer als 50, dann soll eine Ausgabe des Ergebnisses erfolgen.
  - Falls das Ergebnis durch 3 teilbar ist, dann soll keine Ausgabe erfolgen.
  - Falls das Ergebnis durch 5 teilbar ist, dann soll eine andere Ausgabe erfolgen.

Klassen und Objekte

Klassen (class)

- Vorlage/“Bauplan“ für Objekte
- Enthält Methoden und Eigenschaften eines Objekts

```
class Klassenname {
    // Felder (Eigenschaften)
    Datentyp feldName;

    // Methoden (Verhalten)
    Rückgabetyp methodenName(Argumente) {
        // Methodenkörper
    }
}
```

Attribut (attribute)

Objekte (object)

- Instanz einer Klasse
- Enthält konkrete Werte
- Wird mit Hilfe des Konstruktors einer Klasse erzeugt
- Nur gültig in dem Anweisungsblock, in dem es deklariert wurde

Instanziierung von Objekten

Deklaration

Klassenname objektName;

Deklaration und Instanziierung (instantiate) von Objekten

```
Klassenname objektName = new Klassenname();
```

Konstruktor (constructor)

- Spezielle Methode, um Objekte einer Klasse zu initialisieren
- Gleicher Name, wie die Klasse
- Keine Rückgabewerte

```
class Klassenname {
    // Felder (Eigenschaften)
    Datentyp feld1;
    Datentyp feld2;

    // Konstruktor
    Klassenname(Datentyp parameter1, Datentyp parameter2) {
        this.feld1 = parameter1;
        this.feld2 = parameter2;
    }

    // Methoden (Verhalten)
    void zeigeDaten() {
        System.out.println("Feld1: " + feld1 + ", Feld2: " + feld2);
    }
}
```

Konstruktor (constructor)

- Standard-Konstruktor
- Parametrisierter Konstruktor
- Überladener Konstruktor

Konstruktor (constructor)

Standard-Konstruktor:

Wird verwendet, wenn **keine zusätzlichen Informationen** beim Erstellen eines Objekts benötigt werden.

```java
class Person {
    String name;
    int alter;

    // Standard-Konstruktor
    Person() {
        name = "Unbekannt";
        alter = 0;
    }

    void anzeigen() {
        System.out.println("Name: " + name + ", Alter: " + alter);
    }
}
```

Konstruktor (constructor)

Parametrisierter Konstruktor:

Wird verwendet, um Werte beim Erstellen des Objekts zu übergeben.

```
Person(String name, int alter) {
    this.name = name;
    this.alter = alter;
}
```

Konstruktor (constructor)

Überladener Konstruktor:

Mehrere Konstruktoren können in einer Klasse existieren, solange sie unterschiedliche Parameterlisten haben (**Methodenüberladung**).

```
class Person {
    String name;
    int alter;

    // Standard-Konstruktor
    Person() {
        name = "Unbekannt";
        alter = 0;
    }

    // Parametrisierter Konstruktor
    Person(String name, int alter) {
        this.name = name;
        this.alter = alter;
    }
}
```

Kommentare

Einzeilige Kommentare

Einzelzeilige Kommentare beginnen mit //. Alles, was nach // auf der gleichen Zeile folgt, wird vom Compiler ignoriert.

```
int x = 10;  // Dies ist ein Kommentar nach dem Code
```

Mehrzeilige Kommentare

Mehrzeilige Kommentare beginnen mit /* und enden mit */. Sie können über mehrere Zeilen gehen.

```
/*
    Dies ist ein mehrzeiliger Kommentar.
    Er kann über mehrere Zeilen gehen.
*/
int y = 20;  no usages
```

JavaDoc-Kommentare

Javadoc-Kommentare beginnen mit /** und enden mit */. Diese Kommentare sind speziell für die Dokumentation von Klassen, Methoden und Feldern gedacht. Sie können von Tools wie Javadoc verwendet werden, um automatisch Dokumentation zu generieren.

```java
/**
 * Diese Methode berechnet die Summe von zwei Zahlen.
 *
 * @param a Erste Zahl
 * @param b Zweite Zahl
 * @return Die Summe von a und b
 */
public int addiere(int a, int b) {  no usages
    return a + b;
}
```

Mehr Infos: http://www.scalingbits.com/java/javakurs1/javadoc

Programmier-Aufgabe

Schiffe versenken I

1. Erstelle eine Klasse *Ship* mit den Attributen length, hitCount und isSunk. Überlege, welcher Datentyp angemessen wäre.

2. Erstelle einen passenden parametrisierten Konstruktor.

3. Kommentiere deinen Code passend.

Literatur und Quellen

https://www14.in.tum.de/lehre/2016WS/info1/split/sec-Mehr-Java-handout.pdf

https://www.talu.de/schiffe-versenken/

https://users.informatik.uni-halle.de/~brass/oop13/j6_datat.pdf

https://oinf.ch/kurs/programmieren/variablen/

http://www7content.cs.fau.de/data/inf1nf/2018w/Inf1NF_16_Java-Datentypen.pdf

https://de.wikibooks.org/wiki/Java_Standard:_Operatoren