Section 6: Control Flow & Schleifen (Control Flow & Loops)

6.1 Die Switch-Anweisung

Was ist ein Switch?

Die `switch`-Anweisung ist eine Alternative zu langen `if-else if-else`-Ketten, wenn man **einen Wert gegen mehrere feste Werte** vergleichen will.

Traditioneller Switch (mit `case:` und `break`)

```
char charValue='E';
switch(charValue){
case'A':
        System.out.println("Letter 'A' is a NATO word: Able");
break;
case'B':
        System.out.println("Letter 'B' is a NATO word: Baker");
break;
case'C':
        System.out.println("Letter 'C' is a NATO word: Charlie");
break;
case'D':
        System.out.println("Letter 'D' is a NATO word: Dog");
break;
case'E':
        System.out.println("Letter 'E' is a NATO word: Easy");
break;
default:
        System.out.println("Letter not found");
}
```

Wichtige Regeln beim traditionellen Switch:

- Jeder `case` endet mit `break;` – sonst wird der nächste `case` **auch** ausgeführt ("fall-through")!
- `default` fängt alle Werte ab, die keinem `case` entsprechen
- Erlaubte Typen: `byte`, `short`, `int`, `char`, `String`, `enum` (NICHT `long`, `float`, `double`, `boolean`)

Fall-Through-Problem

```
// OHNE break: Fall-Through!
switch(value){
case1:
        System.out.println("Eins");// wird ausgeführt
case2:
        System.out.println("Zwei");// wird AUCH ausgeführt!
case3:
        System.out.println("Drei");// wird AUCH ausgeführt!
}
// Bei value = 1 werden ALLE drei Zeilen ausgegeben!
```

Enhanced Switch (Java 14+) – mit Pfeil-Syntax `->`

Der Enhanced Switch löst das Fall-Through-Problem und ist kompakter:

```
int switchValue=3;
switch(switchValue){
case1 -> System.out.println("Value was 1");
case2 -> System.out.println("Value was 2");
case3,4,5 ->{
        System.out.println("Was a 3, a 4 or a 5");
        System.out.println("Actually it was a "+ switchValue);
}
default -> System.out.println("Value was not 1, 2, 3, 4 or 5");
}
```

Vorteile des Enhanced Switch:

- Kein `break` nötig – kein Fall-Through möglich
- Mehrere Werte pro `case` mit Komma: `case 3, 4, 5 ->`
- Für mehrere Statements: geschweifte Klammern `{ }`

Switch als Ausdruck (Switch Expression)

Der Switch kann direkt einen **Wert zurückgeben** – das ist der modernste Ansatz:

```
publicstatic String getQuarter(String month){
returnswitch(month){
case"JANUARY","FEBRUARY","MARCH" ->"1st";
case"APRIL","MAY","JUNE" ->"2nd";
case"JULY","AUGUST","SEPTEMBER" ->"3rd";
```

```
case"OCTOBER","NOVEMBER","DECEMBER" ->"4th";
default ->{
            String badResponse= month+" is bad";
            yield badResponse;// "yield" statt "return" in Switch-
Blöcken!
}
};// Beachte das Semikolon am Ende!
}
```

`yield` vs `return`:

- `return` beendet die **gesamte Methode**
- `yield` gibt einen Wert aus einem **Switch-Expression-Block** zurück, ohne die Methode zu verlassen

Switch-Expression mit `yield`

```
String dayOfWeek=switch(day){
case0 ->{ yield"Sunday";}
case1 ->{ yield"Monday";}
case2 ->{ yield"Tuesday";}
case3 ->{ yield"Wednesday";}
case4 ->{ yield"Thursday";}
case5 ->{ yield"Friday";}
case6 ->{ yield"Saturday";}
default ->{ yield"Invalid day";}
};
```

Vergleich: Switch vs if-else-if

```
// if-else-if Version (mehr Code, weniger übersichtlich)
publicstaticvoid printWeekDay(int day){
    String dayOfWeek="Invalid Day";
if(day==0){
        dayOfWeek="Sunday";
}elseif(day==1){
        dayOfWeek="Monday";
}elseif(day==2){
        dayOfWeek="Tuesday";
}
// ... usw.
}
```

```
// Switch Version (kompakter, übersichtlicher)
publicstaticvoid printDayOfWeek(int day){
    String dayOfWeek=switch(day){
case0 ->{ yield"Sunday";}
case1 ->{ yield"Monday";}
// ... usw.
default ->{ yield"Invalid day";}
};
}
```

Wann Switch, wann if-else?

Situation	Empfehlung
Vergleich mit festen Werten	`switch`
Bereichsprüfungen (`> 10` , `< 100`)	`if-else`
Komplexe logische Bedingungen	`if-else`
Viele gleichartige Vergleiche	`switch`

Praxisbeispiel: Tage im Monat

```
publicstaticint getDaysInMonth(int month,int year){
if(year<1|| year>9999)return-1;

returnswitch(month){
case1,3,5,7,8,10,12 ->31;
case2 -> isLeapYear(year)?29:28;// Ternärer Operator im Switch!
case4,6,9,11 ->30;
default ->-1;
};
}
```

6.2 Die for-Schleife

Grundstruktur

```
for(Initialisierung; Bedingung; Aktualisierung){
// Schleifenkörper
}
```

Die drei Teile:

1. **Initialisierung**: Wird einmal am Anfang ausgeführt (`int i = 1`)

2. **Bedingung**: Wird **vor** jedem Durchlauf geprüft (`i <= 5`)
3. **Aktualisierung**: Wird **nach** jedem Durchlauf ausgeführt (`i++`)

Einfaches Beispiel

```
for(int counter=1; counter<=5; counter++){
    System.out.println("Counter: "+ counter);
}
// Ausgabe: Counter: 1, Counter: 2, ... Counter: 5
```

for-Schleife mit double

```
for(double rate=2.0; rate<=5.0; rate++){
double interestAmount= calculateInterest(10000.0, rate);
    System.out.println("10,000 at "+ rate+"% interest = "+ interes
tAmount);
}
```

for-Schleife mit Schrittweite (Inkrement)

Man ist nicht auf `i++` beschränkt:

```
// Schrittweite 0.25
for(double i=7.5; i<=10.0; i+=0.25){
double interestAmount= calculateInterest(100.0, i);
    System.out.println("$100 at "+ i+"% interest = $"+ interestAmo
unt);
}
```

`break` – Schleife vorzeitig beenden

`break` beendet die **gesamte** Schleife sofort:

```
for(double i=7.5; i<=10.0; i+=0.25){
double interestAmount= calculateInterest(100.0, i);
if(interestAmount>8.5){
break;// Schleife wird hier beendet
}
    System.out.println("$100 at "+ i+"% interest = $"+ interestAmo
unt);
}
```

`continue` – Aktuellen Durchlauf überspringen

`continue` überspringt den **Rest des aktuellen Durchlaufs** und springt zur nächsten Iteration:

```
int number=0;
while(number<50){
    number+=5;
if(number%25==0){
continue;// Überspringe 25 und 50
}
    System.out.print(number+"_");
}
// Ausgabe: 5_10_15_20_30_35_40_45_
// 25 und 50 fehlen!
```

Komplexe Schleifenbedingungen

Die Schleife kann mehrere Bedingungen kombinieren:

```
int count=0;
for(int i=10; count<3&& i<50; i++){
if(isPrime(i)){
        count++;
        System.out.println("The "+ count+". Prime number is --> "+
i);
}
}
```

Hier wird die Schleife beendet, wenn **entweder** 3 Primzahlen gefunden wurden **oder** `i` 50 erreicht.

Verschachtelte Schleifen (Nested Loops)

Schleifen können ineinander verschachtelt werden. Typisch für 2D-Ausgaben:

```
publicstaticvoid printSquareStar(int number){
for(int row=1; row<= number; row++){
for(int col=1; col<= number; col++){
if(row==1|| row== number||// Erste/letzte Zeile
                col==1|| col== number||// Erste/letzte Spalte
                row== col||// Hauptdiagonale
                col== number- row+1){// Nebendiagonale
                System.out.print("*");
```

```
}else{
                System.out.print(" ");
}
}
        System.out.println();// Neue Zeile nach jeder Reihe
}
}
```

Ausgabe für `printSquareStar(5)`:

```
*****
** **
* * *
** **
*****
```

Laufzeit bei verschachtelten Schleifen: Bei einer äußeren Schleife mit n Durchläufen und einer inneren mit m Durchläufen → n × m Gesamtdurchläufe.

6.3 Die while-Schleife

Grundstruktur

```
while(Bedingung){
// Schleifenkörper
// Bedingung muss sich irgendwann ändern!
}
```

Die `while`-Schleife prüft die Bedingung **vor** jedem Durchlauf. Wenn die Bedingung von Anfang an `false` ist, wird der Schleifenkörper **nie** ausgeführt.

Beispiel: Gerade/Ungerade Zahlen summieren

```
int range=4;
int sumEven=0, sumOdd=0;
int count=0;

while(range<=20){
    range++;
if(!isEvenNumber(range)){
        sumOdd+= range;
continue;// Überspringe den Rest für ungerade Zahlen
```

```
}
if(count!=5){
        sumEven+= range;
        count++;
}
}
```

Die do-while-Schleife

```
do{
// Schleifenkörper (wird mindestens einmal ausgeführt!)
}while(Bedingung);
```

Unterschied zu while: Der Schleifenkörper wird **mindestens einmal** ausgeführt, bevor die Bedingung geprüft wird.

```
int j=1;
boolean isReady=false;
do{
if(j>5){
break;
}
    System.out.println(j);
    j++;
    isReady=(j>0);
}while(isReady);
```

Endlos-Schleife mit `while(true)` und `break`

Ein häufiges Muster für Benutzereingaben:

```
while(true){// Endlos-Schleife
    System.out.println("Enter a number, or any character to exi
t:");
    String nextEntry= scanner.nextLine();
try{
double validNum= Double.parseDouble(nextEntry);
// ... verarbeite Zahl ...
}catch(NumberFormatException nfe){
break;// Beende Schleife bei ungültiger Eingabe
}
}
```

Vergleich: for vs while vs do-while

Eigenschaft	`for`	`while`	`do-while`
Bedingung geprüft	**vor** jedem Durchlauf	**vor** jedem Durchlauf	**nach** jedem Durchlauf
Mindestausführung	0 mal	0 mal	**1 mal**
Typischer Einsatz	Bekannte Anzahl Durchläufe	Unbekannte Anzahl	Mindestens 1 Durchlauf nötig
Zähler-Variable	Ja, eingebaut	Nein, extern	Nein, extern

6.4 Ziffern-Manipulation mit Schleifen

Das Grundmuster: Ziffern einzeln extrahieren

Dies ist eines der wichtigsten algorithmischen Muster in den Übungen:

```
while(number>0){
int lastDigit= number%10;// Letzte Ziffer extrahieren
// ... etwas mit lastDigit machen ...
    number/=10;// Letzte Ziffer entfernen
}
```

Wie es funktioniert am Beispiel `number = 1234`:

Schritt	`number`	`number % 10`	`number / 10`
1	1234	4	123
2	123	3	12
3	12	2	1
4	1	1	0 → Schleife endet

Anwendung 1: Quersumme (Digit Sum)

```
publicstaticint sumDigits(int number){
if(number<0)return-1;

int sum=0;
while(number>0){
        sum+= number%10;// Letzte Ziffer zur Summe addieren
        number/=10;// Letzte Ziffer entfernen
}
return sum;
```

```
}
// sumDigits(1234) → 1 + 2 + 3 + 4 = 10
```

Anwendung 2: Summe der geraden Ziffern

```
publicstaticint getEvenDigitSum(int number){
if(number<0)return-1;

int sum=0;
while(number>0){
int digit= number%10;
if(digit%2==0){// Nur gerade Ziffern
            sum+= digit;
}
        number/=10;
}
return sum;
}
// getEvenDigitSum(123456789) → 2 + 4 + 6 + 8 = 20
```

Anwendung 3: Zahl umkehren (Reverse)

```
publicstaticint reverse(int number){
int reverse=0;
while(number!=0){
        reverse= reverse*10+ number%10;// Ziffer anhängen
        number/=10;// Letzte Ziffer entfernen
}
return reverse;
}
// reverse(1234) → 4321
// reverse(100)  → 1 (führende Nullen gehen verloren!)
```

Schritt-für-Schritt für `reverse(1234)`:

Schritt	`number`	`number % 10`	`reverse * 10 + digit`	`reverse`
1	1234	4	0 × 10 + 4	4
2	123	3	4 × 10 + 3	43
3	12	2	43 × 10 + 2	432
4	1	1	432 × 10 + 1	4321

Anwendung 4: Palindrom-Check

Eine Zahl ist ein Palindrom, wenn sie vorwärts und rückwärts gleich ist (z.B. 121, 1221, 707):

```
publicstaticboolean isPalindrome(int number){
int reverse=0;
int original= number;
while(original!=0){
        reverse= reverse*10+ original%10;
        original/=10;
}
return reverse== number;
}
// isPalindrome(121) → true
// isPalindrome(-1221) → true (weil -1221 reversed auch -1221 ist)
```

Anwendung 5: Erste und letzte Ziffer

```
publicstaticint sumFirstAndLastDigit(int number){
if(number<0)return-1;

int lastDigit= number%10;// Letzte Ziffer: einfach % 10
while(number>=10){
        number/=10;// Teile solange durch 10...
}
int firstDigit= number;// ...bis nur die erste Ziffer übrig ist

return firstDigit+ lastDigit;
}
// sumFirstAndLastDigit(252) → 2 + 2 = 4
// sumFirstAndLastDigit(257) → 2 + 7 = 9
```

Anwendung 6: Anzahl der Ziffern zählen

```
publicstaticint getDigitCount(int number){
if(number<0)return-1;
if(number==0)return1;// Sonderfall: 0 hat 1 Ziffer

int count=0;
while(number!=0){
        count++;
```

```
        number/=10;
}
return count;
}
// getDigitCount(123) → 3
// getDigitCount(0) → 1
```

Alternative mit for-Schleife:

```
publicstaticint getDigitCount(int number){
if(number<0)return-1;
if(number==0)return1;

int count=0;
for(int i=1; i<= number; i*=10){
        count++;
}
return count;
}
```

6.5 Primzahlen und Teiler

Primzahl-Check

Eine Primzahl ist nur durch 1 und sich selbst teilbar (2, 3, 5, 7, 11, 13, ...):

```
publicstaticboolean isPrime(int wholeNumber){
if(wholeNumber<=2){
return(wholeNumber==2);// 2 ist die kleinste Primzahl
}

for(int divisor=2; divisor<= wholeNumber/2; divisor++){
if(wholeNumber% divisor==0){
returnfalse;// Teilbar → keine Primzahl
}
}
returntrue;
}
```

Optimierung: Man muss nur bis `√n` (Wurzel von n) prüfen, nicht bis `n/2`:

```
for(int divisor=2; divisor* divisor<= wholeNumber; divisor++){
...}
```

Alle Teiler einer Zahl finden

```
publicstaticvoid printFactors(int n){
if(n<1){
        System.out.println("Invalid Value");
return;
}
for(int i=1; i<= n; i++){
if(n% i==0){
            System.out.println(i);
}
}
}
// printFactors(6) → 1, 2, 3, 6
// printFactors(32) → 1, 2, 4, 8, 16, 32
```

Größter gemeinsamer Teiler (GGT / GCD) – Euklidischer Algorithmus

```
publicstaticint getGreatestCommonDivisor(int first,int second){
if(first<10|| second<10)return-1;

while(second!=0){
int remainder= first% second;
        first= second;
        second= remainder;
}
return first;
}
```

Ablauf für `GCD(25, 15)`:

Schritt	`first`	`second`	`remainder`
1	25	15	10
2	15	10	5
3	10	5	0
4	5	**0**	→ Ende, GCD = 5

Perfekte Zahl

Eine perfekte Zahl ist gleich der Summe ihrer echten Teiler (z.B. 6 = 1 + 2 + 3):

```
publicstaticboolean isPerfectNumber(int num){
if(num<1)returnfalse;

int sum=0;
for(int i=1; i< num; i++){
if(num% i==0){
            sum+= i;// Nur echte Teiler (< num)
}
}
return sum== num;
}
// isPerfectNumber(6) → true (1+2+3=6)
// isPerfectNumber(28) → true (1+2+4+7+14=28)
```

Größter Primfaktor

```
publicstaticint getLargestPrime(int number){
if(number<2)return-1;

int largestPrime=-1;
for(int i=2; i<= number; i++){
while(number% i==0){
            largestPrime= i;// Aktueller Faktor ist Primfaktor
            number/= i;// Zahl reduzieren
}
}
return largestPrime;
}
// getLargestPrime(21) → 7 (21 = 3 × 7)
// getLargestPrime(45) → 5 (45 = 3 × 3 × 5)
```

Warum funktioniert das? Weil wir bei 2 anfangen und alle Vielfachen eines Faktors entfernen, bevor wir zum nächsten gehen. Zusammengesetzte Zahlen wurden bereits durch ihre Primfaktoren entfernt, also sind alle gefundenen Teiler automatisch Primzahlen.

6.6 Benutzereingabe (User Input)

`System.console()` – Für Konsolen-Anwendungen

```
publicstatic String getInputFromConsole(int currentYear){
    String name= System.console().readLine("Enter your name: ");
    String dateOfBirth= System.console().readLine("Enter your birt
h year: ");
int age= currentYear- Integer.parseInt(dateOfBirth);
return"So you're "+ age+" years old!";
}
```

Problem: `System.console()` gibt `null` zurück, wenn das Programm nicht in einer echten Konsole läuft (z.B. in IntelliJ).

`Scanner` – Die universelle Alternative

```
import java.util.Scanner;

Scanner scanner=new Scanner(System.in);

// String einlesen
String name= scanner.nextLine();

// Zahl einlesen (als String → dann parsen)
String input= scanner.nextLine();
double number= Double.parseDouble(input);

// Oder direkt (aber Vorsicht mit nextLine danach!)
int n= scanner.nextInt();
```

Fehlerbehandlung bei Benutzereingabe

Benutzer können ungültige Werte eingeben. Die Kombination aus `try-catch` und `do-while` ist das Standardmuster:

```
Scanner scanner=new Scanner(System.in);
boolean validInput=false;
int age=0;

do{
    System.out.println("Enter a year of birth:");
try{
        age= checkDate(currentYear, scanner.nextLine());
        validInput= age<0?false:true;
}catch(NumberFormatException badUserData){
```

```
        System.out.println("Characters not allowed! Try again");
}
}while(!validInput);
```

Mehrere Zahlen einlesen und summieren

```
Scanner scanner=new Scanner(System.in);
int counter=1;
double sum=0;

do{
    System.out.println("Enter number #"+ counter+": ");
    String nextNumber= scanner.nextLine();
try{
double number= Double.parseDouble(nextNumber);
        counter++;
        sum+= number;
}catch(NumberFormatException nfe){
        System.out.println("Invalid number");
// counter wird NICHT erhöht → Benutzer muss erneut eingeben
}
}while(counter<=5);

System.out.println("The sum of the 5 numbers = "+ sum);
```

Min/Max-Challenge: Minimum und Maximum finden

```
Scanner scanner=new Scanner(System.in);
double max=0, min=0;
int loopCount=0;

while(true){
    System.out.println("Enter a number, or any character to exi
t:");
    String nextEntry= scanner.nextLine();
try{
double validNum= Double.parseDouble(nextEntry);
if(loopCount==0|| validNum< min){
            min= validNum;// Neues Minimum gefunden
}
if(loopCount==0|| validNum> max){
            max= validNum;// Neues Maximum gefunden
```

```
}
        loopCount++;
}catch(NumberFormatException nfe){
break;// Nicht-Zahl → Schleife beenden
}
}

if(loopCount>0){
    System.out.println("min = "+ min+", max = "+ max);
}else{
    System.out.println("No valid data entered");
}
```

Wichtiges Muster: `loopCount == 0` wird verwendet, um den allerersten Wert sowohl als Min als auch als Max zu setzen, da es noch keinen Vergleichswert gibt.

Wichtige Parsing-Methoden

Methode	Beschreibung	Beispiel
`Integer.parseInt(s)`	String → int	`Integer.parseInt("42")` → 42
`Double.parseDouble(s)`	String → double	`Double.parseDouble("3.14")` → 3.14
`Long.parseLong(s)`	String → long	`Long.parseLong("999999999")`
`scanner.nextLine()`	Liest eine ganze Zeile als String	
`scanner.nextInt()`	Liest direkt einen int	
`scanner.hasNextInt()`	Prüft ob nächste Eingabe ein int ist	

`scanner.hasNextInt()` – Elegante Eingabeprüfung

```
Scanner scanner=new Scanner(System.in);
int sum=0;
int count=0;

while(scanner.hasNextInt()){// Solange gültige Zahlen eingegeben w
erden
    sum+= scanner.nextInt();
    count++;
}
```

```
long average=(count==0)?0L: Math.round((double) sum/ count);
System.out.println("SUM = "+ sum+" AVG = "+ average);
```

6.7 Typ-Konvertierung und nützliche Methoden

`Integer.parseInt()` und `Double.parseDouble()`

```
String input="42";
int number= Integer.parseInt(input);// String → int

String input2="3.14";
double pi= Double.parseDouble(input2);// String → double
```

Wirft `NumberFormatException` wenn der String keine gültige Zahl ist:

```
Integer.parseInt("abc");// → NumberFormatException!
Integer.parseInt("3.14");// → NumberFormatException! (ist kein in
t!)
```

`Math.ceil()` – Aufrunden

```
Math.ceil(4.1)// → 5.0
Math.ceil(4.9)// → 5.0
Math.ceil(5.0)// → 5.0

// Typischer Einsatz: "Wie viele Eimer Farbe braucht man?"
double area=7.14;
double areaPerBucket=2.5;
int bucketsNeeded=(int) Math.ceil(area/ areaPerBucket);
// 7.14 / 2.5 = 2.856 → aufgerundet = 3 Eimer
```

`Math.round()` – Kaufmännisches Runden

```
Math.round(4.4)// → 4
Math.round(4.5)// → 5
Math.round(4.6)// → 5

// Durchschnitt berechnen und runden:
long average= Math.round((double) sum/ count);
```

Unterstrich in Zahlenliteralen (_)

Java erlaubt Unterstriche in Zahlen für bessere Lesbarkeit:

```
int million=1_000_000;// statt 1000000
int year=9_999;// statt 9999
long big=1_234_567_890L;// statt 1234567890L
```

6.8 Fortgeschrittene Übungskonzepte

Gemeinsame Ziffern zweier Zahlen finden

Zwei verschachtelte while-Schleifen vergleichen jede Ziffer von Zahl A mit jeder Ziffer von Zahl B:

```
publicstaticboolean hasSharedDigit(int a,int b){
if(a<=10|| a>=100|| b<=10|| b>=100)returnfalse;

int nextA= a;
while(nextA>0){
int aCheck= nextA%10;
int nextB= b;
while(nextB>0){
int bCheck= nextB%10;
if(aCheck== bCheck)returntrue;
            nextB/=10;
}
        nextA/=10;
}
returnfalse;
}
```

Elegantere Alternative für zweistellige Zahlen:

```
publicstaticboolean hasSharedDigit(int first,int second){
int firstLeft= first/10;// Zehner-Ziffer
int firstRight= first%10;// Einer-Ziffer
int secondLeft= second/10;
int secondRight= second%10;

return firstLeft== secondLeft|| firstLeft== secondRight
```

```
|| firstRight== secondLeft|| firstRight== secondRight;
}
```

Zahl in Wörter umwandeln (NumberToWords)

Dieses Problem kombiniert mehrere Techniken:

```
publicstaticvoid numberToWords(int number){
if(number<0){ System.out.println("Invalid Value");return;}
if(number==0){ System.out.println("Zero");return;}

int reverse= reverse(number);
int leadingZeroes= getDigitCount(number)- getDigitCount(reverse);

while(reverse!=0){
int lastDigit= reverse%10;
switch(lastDigit){
case0 -> System.out.println("Zero");
case1 -> System.out.println("One");
case2 -> System.out.println("Two");
// ... usw.
}
        reverse/=10;
}

// Führende Nullen der umgekehrten Zahl waren trailing Nullen der
originalen Zahl
for(int i=0; i< leadingZeroes; i++){
        System.out.println("Zero");
}
}
// numberToWords(1000) → "One Zero Zero Zero"
```

Problem der verlorenen Nullen: Wenn man `1000` umkehrt, bekommt man `1` (nicht `0001`). Deshalb vergleicht man die Ziffernanzahl des Originals mit der des Umgekehrten und gibt die Differenz als zusätzliche "Zero" aus.

FlourPacker – Logisches Denken mit Division

```
publicstaticboolean canPack(int bigCount,int smallCount,int goal){
if(bigCount<0|| smallCount<0|| goal<0)returnfalse;

int maxBigBags= goal/5;// Maximale Anzahl großer Beutel die passen
```

```
int bigBagsToUse= Math.min(bigCount, maxBigBags);// Tatsächlich nu
tzbare
int remainder= goal-(bigBagsToUse*5);// Rest nach großen Beuteln

return smallCount>= remainder;// Genug kleine Beutel für den Rest?
}
```

PaintJob – Methodenüberladung in der Praxis

```
// Version 1: Nur Fläche und Eimer-Kapazität
publicstaticint getBucketCount(double area,double areaPerBucket){
if(area<=0|| areaPerBucket<=0)return-1;
return(int) Math.ceil(area/ areaPerBucket);
}

// Version 2: Breite, Höhe, Eimer-Kapazität
publicstaticint getBucketCount(double width,double height,double a
reaPerBucket){
if(width<=0|| height<=0|| areaPerBucket<=0)return-1;
return(int) Math.ceil((width* height)/ areaPerBucket);
}

// Version 3: Breite, Höhe, Eimer-Kapazität, bereits vorhandene Ei
mer
publicstaticint getBucketCount(double width,double height,double a
reaPerBucket,int extraBuckets){
if(width<=0|| height<=0|| areaPerBucket<=0|| extraBuckets<0)return
-1;
int needed=(int) Math.ceil((width* height)/ areaPerBucket);
return needed- extraBuckets;
}
```

6.9 Zusammenfassung der wichtigsten Konzepte

Konzept	Beschreibung
Traditioneller Switch	`case:` + `break;` – Vorsicht vor Fall-Through
Enhanced Switch	`case X ->` – kein break nötig, moderner
Switch Expression	`switch` als Ausdruck mit `yield` für Rückgabewerte
for-Schleife	Bekannte Anzahl Iterationen, Zähler eingebaut
while-Schleife	Unbekannte Anzahl, Bedingung vor dem Durchlauf

Konzept	Beschreibung
do-while	Mindestens 1 Durchlauf, Bedingung nach dem Durchlauf
break	Beendet die gesamte Schleife sofort
continue	Überspringt den Rest des aktuellen Durchlaufs
Ziffern-Extraktion	`% 10` für letzte Ziffer, `/= 10` zum Entfernen
Zahl umkehren	`reverse = reverse * 10 + number % 10`
Scanner	Benutzereingabe mit `scanner.nextLine()`
try-catch	Fehlerbehandlung bei ungültiger Eingabe
Verschachtelte Schleifen	Für 2D-Muster und Ziffernvergleiche
Euklidischer Algorithmus	GGT berechnen mit Modulo-Schleife
Math.ceil()	Aufrunden (z.B. für "mindestens X Eimer")

Dieser Konspekt basiert auf dem Code und den Übungen der Section 6 des Java-Kurses.