Java Zusammenfassung

Standardprogramm

- Einstieg/Initialisierung in der main() Funktion
- Hier(Basic/Start) die Variablen deklarieren, Methoden aufrufen
- „Sichtbarkeit“/Gültigkeit innerhalb von { } -> Methoden/Klassen/Codeabschnitte

Variablen:

- speichern Daten in Datentypen
- klein geschrieben
- Gültigkeit innerhalb von geschweiften Klammern (Ausnahme: globale Variablen)
- Basisdatentypen: `int x = 3;`
- komplexe Datentypen: `MyClass var = new MyClass(); int[] array1 = new int[3];`

Basisdatentypen

ganzzahlig: Byte, Short, Integer, Long
Fließkomma: Float, Double
Zeichen: Character
Wahrheitswerte: Boolean

Casting

- wandelt (sinnvoll) Basisdatentypen ineinander um, zB double zu int
- Bsp: (neuer Datentyp) WertZumUmwandeln; `int integerWert = (int) 1.9387;`

Operatoren

arithmetisch: + - * / % ++ -- += -= *= /=
Vergleich: == != < <= > >=
logisch: ! && || ^^

Strings

- speichert Zeichenfolgen (Array aus Chars)
- lassen sich über Klassenmethoden bearbeiten/vergleichen

Bsp:

```
String string1 = new String; String string2 = „mein String“;
```

Arrays

- speichern beliebige Datentypen in Feldern
- mehrdimensional möglich
- Initialisierung über new oder Initialisierungsliste
- iterieren mit for- /for-each Schleife
- Felder werden mit 0 angefangen zu zählen
- Zugriff auf Element mit array1[i]

Bsp:

Datentyp [] Bezeichner = new Datentyp[Größe(Integer)]

```
int[] array1 = new int[5]; int[] array2 = {1,2,3};
```

Methoden

- allgemeingültig mit Variablen um gleichen Code für mehrere Werte zu nutzen
- Geltungsbereich innerhalb der geschweiften Klammern (auch für Variablen/PARAMETER)
- Name wird klein geschrieben
- Überladen: gleicher Name, Parameter MÜSSEN unterschiedlich sein, Rückgabetyp KANN
- Überschreiben: gleicher Name + Signatur(Parameterliste) + Rückgabetyp

Bsp:

Rückgabetyp Name (Parameterliste){ return(außer bei void); }

```
int addition(int variable1, int variable2)
{
    return variable1+variable2;
}
MyClass methodeX(MyClass variable1, int[] array1)
{
    // do some stuff
    return variable1;
}
```

Kontrollstrukturen

- Prüfen von Anweisungen

if-else Verzweigung

→ einseitig/mehrseitig

if(Bedingung){ }
else { }

Bsp:

```
if(x < 10)
{
    x += 3;
}
else
{
    x--;
}
```

n-seitige Verzweigung: Switch-case

→vordefinieren von speziellen Anwendungsfällen

- switch(Abfrage){
  case X: // Code; break;
  case Y: // Code; break;
  default: //Code; break; }

Bsp:

```
switch(intVariable)
{
    case 1: Sytem.out.println(„Menue 1 wird geöffnet“);
          break;
    case 2: System.out.println(„Menue 2 wird geöffnet“);
          break;
    default: Sytem.out.println(„Falsche Eingabe“);
          break;
}
```

Schleifen

- Mehrfaches durchlaufen von Codesegmenten -> Wiederholungen
- besitzen Abbruchbedingungen, sonst Endlosschleife

Kopfgesteuert: while-do

→Bedingung steht am Anfang der Schleife und wird vor der Ausführung geprüft

Bsp:

while(Bedingung) { //Code;}

```
while(intVariable < 10)
{
    System.out.println(intVariable);
    intVariable++;
}
```

Fußgesteuert: do-while

→Bedingung steht am Ende des Anweisungsblocks, wird mind. 1x ausgeführt

Bsp:

do{ //Code} while(Bedingung);

```
do
{
    System.out.println(intVariable);
    intVariable++;
} while(intVariable < 10);
```

- Zählschleife: for-Schleife

→zählt Variable in definierten Schritten hoch/runter, steuert somit die Ausführung

- Aufbau: Zählvariable(ganze Zahlen), Bedingung für Ausführung, Verändern der Zählvariable
- Kann auch einzelne Abschnitte von komplexen Datentypen(zB Arrays, Strings) ausgeben

Bsp:

for(ganzzahlige Zählvariable; Bedingung; Verändern der Zählvariable){ //Code; }

```
for(int i = 0; i < 10; i++)
{
    System.out.println(i);
}
```

For-each Schleife:

- iterieren von Arrays/Listen
- kann nur gesamtes Array durchlaufen (keine Bereiche)
- nur lesend, nicht schreibend
  → Elemente können nicht verändert werden, sondern nur aufgerufen

Bsp:

for(Datentyp Bezeichner : Objekt) {}

```
for(int e : intArray)
{
    System.out.println(e);  //gibt alle Elemente des Arrays aus ( e = Element)
}
```

Rekursion

- Aufruf einer Methode durch sich selbst
- Ergebnis kann direkt durch letzten Schritt oder durch Teilschritte(Rekursionsaufrufe) berechnet/ausgegeben werden
- Vorteile: Code teilweise leichter zu lesen, teils leichter zu implementieren
- Nachteile: Code teils schwerer verständlich, teils langsamer

→ Wahl ob iterativ oder rekursiv ist vom speziellen Fall abhängig

Aufbau:

1. Abbruchbedingung: gibt Ergebnis oder Teilergebnis zurück, ist der letzte Schritt
2. Rekursion: teilt in Teilprobleme auf, bis das Problem einfach genug ist
   → Methode ruft sich selber mit verringertem Problem auf
   → Teilergebnisse KÖNNEN erforderlich/nach oben weitergereicht werden

Bsp:

Sichtbarkeit Rückgabetyp Name (Parameter){
  if(Abbruchbedingung) return Ergebnis;
  else return Methode(>veränderte< Parameter)(+-*/…Zwischenergebnis);
}

```
public int summe(int anzahlZahlen)
{
    if(anzahlZahlen == 0)
      return 0;                  //Abbruchbedingung
    else
     return (summe(anzahlZahlen - 1) + anzahlZahlen);//Rekursiver Aufruf +
                                                      Teilergebnis
}
```

Klassen

- stellen eine Art Paket zum Zusammenfassen von Daten dar → komplexer Datentyp
- Namen beginnen mit großem Buchstaben
- Sollen kompakte Darstellungsform zum Arbeiten mit größeren Daten bieten
  → Variablen(Speicher) und Methoden(Zugriff, „Aufgaben“) in einem Paket
- idR eine Klasse pro Datei (.java)
- Variablen private deklarieren(nach außen nicht sichtbar)
  →Zugriff über Getter(Lesen) und Setter(Schreiben) Methoden
- durch Instanzen erstellt → Erzeugung mit new (Aufruf des Konstruktors, wie Methode)
- Inhalte(idR Methoden) werden über Punktoperator aufgerufen:
  →`MyClass objekt1 = new MyClass(3);  objekt1.getX();`
- this greift aktuelles Objekt zu → zwar allgemein geschrieben(Klasse), aber spezifisch(Objekt) ausgeführt
- Aufbau:

Sichtbarkeit class Name{
//Membervariablen
Sichtbarkeit Typ Name;
//Konstruktor
Sichtbarkeit Klassenname(Parameter){}
//Getter/Setter
Sichtbarkeit Rückgabetyp Name(Parameter){}
// weitere Methoden
Sichtbarkeit Rückgabetyp Name (Parameter){}
}

```
public class MyClass{
      // Membervariablen deklarieren
      private int x;
      // CTor schreiben – default und/oder advanced
      public MyClass(int neueZahl){
            this.x = neueZahl)
      }
      //Getter/Setter zum Zugriff auf Attribute/Variablen
      public int getX(){    // Getter = Auslesen
            return this.x;
      }
      public void setX(int neuesX){    // Setter = Schreiben
            this.x = neuesX;
      }
      //weitere Methoden zum Arbeiten mit der Klasse
      public int add(int andereZahl){
            return this.x + andereZahl;
      }
}
```

Vererbung

- einsparen von Code durch erweitern einer bereits bestehenden Klasse
- Alle vererbten Eigenschaften der Basisklasse sind vorhanden(sofern sichtbar) wie eigene
- Erweitern einer Klasse (erben von Klasse) durch *extends* (Klassen) ODER *implements* (Interface)
- extends zum erweitern einer Klasse(erbt alle sichtbaren Eigenschaften)
- implements zum Erweitern der eigenen Klasse um die Eigenschaft des Interface (z.B. Comparable)
  → Klasse muss Methoden des Interface überschreiben! (z.B. compareTo)
- Sichtbarkeiten: *private:* nur innerhalb der Klasse (auch Klasse innerhalb Klasse)
  public: nach außen hin sichtbar (überall)
  protected: nur innerhalb der Klasse und für erbende Klassen sichtbar
  -nichts- : Paketsichtbar(Alle Klassen innerhalb eines Package)

- Schlüsselwort *super* ruft Methode/Attribut der Basisklasse auf → ähnlich wie this
- super CTor: ruft CTor der Basisklasse auf, man spart Code,
  → super CTor MUSS im neuen CTor als erstes aufgerufen werden(mit Parametern)

Bsp:

```
public Combi (int volumen, Color color)        //CTor der neuen Klasse
{
      super(color);       //Aufruf der Basisklasse durch super
                          //in den Klammern die Parameter für den CTor
      this.volumen = volumen;       //Variable der neuen Klasse
}
```

- super wird vor allem für Methoden genutzt, wenn diese in der erbenden Klasse überschrieben wurden

Bsp: in beiden Klassen wird eine Methode mit gleichem Namen, Rückgabewert und Parametern definiert
→ wird in der erbenden Klasse überschrieben
→ über super.methodenname() wird die Methode der Basisklasse aufgerufen

```
super.myMethod(a, b);     //Aufruf der Methode der Basisklasse
myMethod(a, b);           //Aufruf der Methode der erbenden(neuen) Klasse
```