# Klassen

Klassen bilden die Basis für die Objektorientierte Programmierung. Die Mainklasse kennst du bereits. **Für jede Klasse wird eine neue Datei mit gleichen Namen angelegt.**

```
public class Main{
    public static void main(String[] args){
        ...
    }
}
```

# Klassen und Objekte

Alle Überlegungen die der Programmierung dienen, orientieren sich an Objekten.

Klassen können Repräsentationen von realen Dingen sein - ein Haus, ein Tisch, eine Tür, oder etwas nicht physisches - eine E-Mail, ein Spiel, ein Newsletter. Ein Objekt ist dann eine Instanz einer Klasse - das Haus von Stefan, die Schlafzimmertür.

## Klassen

**Bsp-Email**

```
public class Email {               //Name of the class

    private String sender;          // attributes
    private String[] receivers;             // of
    private String subject;         // the
    private String content;         // class

}
```

Klassen die nur Attribute halten nennt man Datenklassen. Klassen können aber auch Funktionen beinhalten:

**Bsp-DragCar**

```
public class DragCar {
    private double currentSpeed = 0;
    private double acceleration = 100;
    private boolean isCarRunning = false;

    //Following are the functions of the class

    public void startCar() {
        isCarRunning= true;
    }
    public void stopCar() {
        isCarRunning= false;
    }
    public void accelerate() {
        currentSpeed+= acceleration;
    }
    public void break() {
        currentSpeed-= acceleration;
    }
}
```

Unser modelliertes Auto für ein Drag-Race kann gestartet werden, beschleunigen, nach dem Rennen abbremsen und schlussendlich wieder abgeschaltet werden. Da bei einem Drag-Race keine Kurven vorkommen braucht unser Auto auch nicht lenken können. **Beachte**, dass nur abgebildet werden soll, was auch gebraucht wird.

## Objekte

Um nun eine Instanz von einem Objekt zu erstellen wird das **new** Keyword verwendet.

**Bsp-Drag-Race**

```
public static void main (String[] args) {
    DragCar myDragCar = new DragCar();      // create instance of class DragCar
    DragCar otherCar = new DragCar();       // create another instance

    myDragCar.startCar();               // functioncall to start myDragCar
    otherCar.startCar();                // functioncall to start otherCar

    myDragCar.accelerate();             //functioncall to accelerate myDragCar
    otherCar.accelerate();              //functioncall to accelerate otherCar
}
```

Man kann unendlichviele Objekte einer Klasse erstellen (natürlich nur bis der Speicher vom Programm voll ist^^) und kann auch jedes davon individuell ansprechen.

## Konstruktor

Beide Autos die wir erstellt haben, besitzen die genau gleichen Eigenschaften mit den selben Werten. Sind also gleich schnell wodurch ein Rennen ein bisschen langweilig ist. Mit einem Konstruktor kann man einem Objekt beim initialisieren ein paar Standardwerte mitgeben. **Bsp-Drag-Race**

```
public class DragCar {
    private double currentSpeed = 0;
    private double acceleration = 100;
    private boolean isCarRunning = false;

    //constructor
    public DragCar (double accelerationFromConstructor) {
        acceleration = accelerationFromConstructor;
    }

    //Following are the functions of the class

    public void startCar() {
        isCarRunning= true;
    }
    public void stopCar() {
        isCarRunning= false;
    }
    public void accelerate() {
        currentSpeed+= acceleration;
    }
    public void break() {
        currentSpeed-= acceleration;
    }
}
```

Ein Konstruktor ist eine Methode, die so aufgebaut ist:

```
public <Klassename> ( <parameterType> <parameterName>, <nextParameterType> <nextParameterName> ) {
    // Zuweisung der Parameterwerte an Variablen der Klasse
    // Weitere Schritte die direkt und immer bei der Initialisierung passieren sollen
}
```

Standardmäßig hat jede Klasse einen sogenannten **default constructor** der bei der Initialisierung aufgerufen wird mit der nun bekannten Zeile:

```
DragCar myDragCar = new DragCar();
```

Der **default constuctor** ist komplett leer. Er wird außerdem gelöscht wenn ein eigener Kontruktor geschrieben wird. Um nun unseren neuen Konstruktor aufzurufen müssen wir einfach die von uns angegebenen Parameter übergeben:

```
DragCar myDragCar = new DragCar(100); //im Konstruktor setzen wir somit den Wert von acceleration auf 100

DragCar otherCar = new DragCar(75); //dieses Auto wird nun nur mit 75 beschleunigen und nicht mit 100 wie das andere
```

Man kann auch mehrere Konstruktoren erstellen um Objekte mit verschiedenen zu setztenden Parametern zu erstellen.

**Bsp-Drag-Race**

```
 //Default constructor
public DragCar () {
    ...
}
//Simple acceleration constructor
public DragCar (double acceleration) {
    ...
}
//a bit more complex constructor
public DragCar (double acceleration, Color color) {
    ...
}
//invalid constructor because there already is another constructor with the same parametertypes
public DragCar(double speed) {
    ...
}
```

# Zugriff auf Properties

## Getter

Um die Beschleunigung unseres Autos außerhalb der Klasse benutzen zu können, müssen wir einen **Getter** erstellen.

**Bsp-Drag-Race** ``` public class DragCar { ... private double acceleration; ... //Getter public double getAcceleration () { return this.acceleration; } ... }

```
Ein **Getter** ist eine normale Methode, die aber nur den Wert einer Variable zurück gibt.
Das **this** Keyword wird verwendet um auf die aktuelle Instanz der Klasse(also das Objekt) zuzugreifen.
### Setter
Will man nun den Wert eines Attributes einer Klasse nachträglich ändern verwendet man einen sogenannten **Setter**.

**Bsp-Drag-Race**
 ```
 public class DragCar {
     ...
     private double acceleration;
     ...
     //Setter
     public double setAcceleration (double acceleration) {
         this.acceleration = acceleration;
        }
        ...
}
```

Ein **Setter** ist auch eine normale Methode mit einem Parameter um den Wert des Attributes zu verändern. Hier sieht man auch den Hauptnutzen des **this** Keywords. Heißen Parameter einer Methode nämlich gleich wie ein Attribut einer Klasse will Java wissen welches Feld denn gemeint ist. Mit **this** werden dann immer die Attribute der Klasse, die Methoden der Klasse, etc. angesprochen.

## Allgemeines zu Getter und Setter

1. Sie sind ein **Muss** für gute Codequalität
2. Nur erstellen wenn nötig. (Soll ein Wert nicht geändert werden können braucht er auch keinen **Setter**)
3. **Getter** und **Setter** können auch komplexere Codestücke beinhalten, wenn sich der Wert durch irgendeine Formel berechnet)

# static

Alle Attribute und Methoden die wir bis jetzt erstellt haben, waren Objektattribute und Objektmethoden und benötigen außerhalb der Klasse ein Objekt um darauf zuzugreifen. Es gibt aber auch Fälle wo ein Objekt nicht sinnvoll wäre. Dann braucht man Klassenattribute und Klassenmethoden. Diese werden in einer Klasse direkt beim Attribut oder Methode mit dem **static** Keyword markiert.

**Bsp-File-Helper**
```

```
public static class FileHelper{
    public static String PathSeparator= "/";
    public static String[] getLinesOfFile (String pathToFile) {
        //Pseudocode to get file contents
        return File.ReadAllLines(pathToFile);
    }
}
```

Diese Attribute und Methoden können nun einfach aufgerufen werden.

```
public static void main (String[] args) {
    FileHelper.getLinesOfFile("myFolder" + FileHelper.PathSeparator + "pathToMyFile.txt"); //wird zu "myFolder/pathToMyFile.txt"
}
```

Klassen können sowohl **nonStatic** als auch **static** Elemente beinhalten. In einer **static** Methode können keine Objektattribute verwendet oder Objektmethoden aufgerufen werden.

**Bsp-Person**

```
public class Person {
    public static int PERSON_COUNT = 0;
    private String name;
    public Person (String name) {
        this.name = name;
        PERSON_COUNT++;
    }
    public String getName () {
        return this.name;
    }
}
```

Nun können Personen erstellt werden und wir wissen zu jeder Zeit wie viele bereits existieren.

```
public static void main (String[] args) {
    Person alex = new Person("Alex");
    System.out.printf(alex.getName() + " is our " + Person.PERSON_COUNT + ". person");
    Person denise= new Person("Denise");
    System.out.printf(denise.getName() + " is our " + Person.PERSON_COUNT + ". person");
    Person eric= new Person("Eric");
    System.out.printf(eric.getName() + " is our " + Person.PERSON_COUNT + ". person");
```

**Output** Alex is our 1. person Denise is our 2. person Eric is our 3. person

# Übungen

**Bsp-Custom-Number**

Erstelle eine Klasse die so funktionieren soll:

```
public static void main (String[] args) {
    CustomNumber myNumber1 = new CustomNumber(3);
    System.out.println(myNumber1.add(4));           //Output: 7

    int result = CustomNumber.add(5, 6);
    System.out.println(result);             //Output: 11
}
```

**Bsp-Special-Dates**

Erstelle eine Klasse mit der du spezielle Termine speichern kannst. Ein Termin soll einen **Namen** und einen **Zeitpunkt** haben. Erstelle ein kleines Programm mit dem der Nutzer eine Liste solcher spezieller Termine eingeben kann.

**Bsp-Secure-Password**

Erstelle eine Klasse die das Passwort eines Benutzers speichert. Das Passwort soll nur über den Setter einzugeben sein und nur über den Getter abrufbar sein. Besonderheit soll aber sein das wir das Passwort dazwischen umgekehrt speichern. Von der `main()` soll das aber nicht ersichtlich sein.

```
public static void main(String[] args) {
    Password myPassword = new Password();
    myPassword.setPassword("abc");
    System.out.println(myPassword.getPassword()) // output = abc
    System.out.println(myPassword.toString()) // output = Das Passwort ist abc
}
```

In der Password Klasse sollst du mit einer Methode `reverseString(String s)` das Passwort umdrehen und auch umgedreht speichern. Überschreibe zusätzlich auch die `toString()` Methode und gib darin das Passwort aus.