**EIA – SCRIPT**

**L01 Recap & Foundation**

**L01.1 Grundlagen**

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungQuelle: <https://www.challengecharterschool.net/wp-content/uploads/2017/07/Steve-Jobs.jpg>

**Einblick**

Ein Computer kann, sehr vereinfacht ausgedrückt, eigentlich nicht viel mehr, als Information aus einem Speicher auszulesen, sie mit einer anderen Information zu verknüpfen, beispielsweise mit Hilfe einer mathematischen Operation, und das Ergebnis wieder im Speicher abzulegen. Der Code

**Datentypen**

Die heute gebräuchlichen Computer speichern Informationen als lange Serien von Bits, kleinste Informationseinheiten, die nur zwei Zustände aufweisen können: an und aus. Damit werden aber ganz unterschiedliche Typen von Informationen kodiert: Zahlen, Texte, Bilder, Musik, Videos usw.  
Damit der Rechner nun aber die Informationen korrekt verarbeiten kann, muss bekannt sein, welchen Typ eine Bitfolge darstellen soll.

***Achtung:****Datentypen sind essentiell wichtig! Javascript geht allerdings sehr lax damit um und ändert, wenn man nicht sehr aufpasst, durchaus auch Datentypen während des Programmlaufs, weshalb hier viele Fehler passieren. TypeScript schafft Abhilfe…*

So angenehm es auch ist, dass TypeScript die Typen aus dem Kontext inferieren kann und in der Folge auf Typfehler aufmerksam macht: noch besser ist es, sich bereits bei der Konzeption Gedanken um die gewünschten Datentypen zu machen und diese festzulegen!

**Komplexe Datentypen**

Neben den einfachen Datentypen wie number, boolean und string gibt es noch komplexe Datentypen. Diese können mehrere Informationseinheiten einfachen oder wiederum komplexen Typs enthalten. Man kann dabei unterscheiden zwischen homogenen Datenstrukturen, bei denen die darin verwalteten Informationseinheiten alle vom gleichen Typ sind, und heterogenen, bei denen unterschiedliche Datentypen verwaltet werden. Eine weitere Unterscheidung wird getroffen aufgrund unterschiedlicher Art der Adressierung der Informationen. Diese kann über einen Index erfolgen, also eine Zahl, welche die Position der Information in einer Reihe angibt, oder einen Schlüssel, der mit der Information assoziiert ist und den Zugang darstellt. Meist ist dieser Schlüssel vom Typ string, es sind aber auch andere Typen möglich. Schließlich ist es noch möglich zwischen vordefinierten Schlüsseln und solchen, die zur Laufzeit des Programms erzeugt werden zu unterscheiden.

**Array**

Ein Array ist eine Datenstruktur, in der die enthaltenen Informationen, dann Elemente genannt, mit Indizes adressiert werden. In Javascript sind Arrays grundsätzlich heterogen. Ein Array kann mit den Anweisungen [ ] oder new Array() erzeugt werden. In die Klammern kann bereits bei der Erzeugung eine Liste von Elementen, durch Komma getrennt, angegeben sein.

Es lässt sich also alles Mögliche an beliebige Stellen in das Array ‘reinwerfen’, sogar andere Arrays. Mit TypeScript kann immerhin der Bereich der Typen der Elemente bis zur Homogenität eingeschränkt werden, wodurch eine wichtige Fehlerquelle reduziert wird.

**Assoziatives Array**

Bei einem assoziativen Array werden die Elemente mit sogenannten Schlüsseln verknüpft. Ein solches assoziatives Array wird in der Regel mit Hilfe geschweifter Klammern erzeugt, wobei innerhalb der Klammern bereits Schlüssel-Werte-Paare angegeben werden können. Die Assoziation wird durch den Doppelpunkt : dargestellt.

Nicht nur die Datentypen sind heterogen, sondern auch die Schlüssel können beliebig gewählt werden. Das ermöglicht große Flexibilität, aber auch Fehler, die schwer zu finden sind. Um assoziative Arrays stringenter zu strukturieren, stellt TypeScript interfaces zur Verfügung. Damit lassen sich Schlüssel vordefinieren und die Datentypen für Werte und Schlüssel einschränken.

Im folgenden Beispiel sind die Schlüssel frei wählbar, sind aber auf den Typ stringbeschränkt, und das Array ist auf Wahrheitswerte homogenisiert.

interface MapStringToBoolean {

[key: string]: boolean;

}

let a: MapStringToBoolean = {"wert1": true, "wert2": false};

Im nächsten Beispiel sind die Schlüssel vordefiniert und die zugeordneten Werte müssen von bestimmten Typen sein.

interface VectorWithMeaning {

x: number;

y: number;

meaning: string;

}

let vector: VectorWithMeaning = {x: 12.4, y: -7.2, meaning: "Ortsvektor"};

**Funktion**

Eine Funktion ist ein Stückchen Code innerhalb eines Programms, das ein eigenes kleines Programm darstellt, einen eigenen Namen hat, eingehende Daten verarbeiten und ausgehende Daten erzeugen kann. Eine solche Funktion kann dann von anderen Programmteilen genutzt werden, indem der Name zusammen mit den zu verarbeitenden Daten aufgerufen wird. Liefert die Funktion Daten zurück, können sie zum Beispiel einfach mit Hilfe des Zuweisungsoperators =einer Variablen zugewiesen werden und stehen damit zur Weiterverarbeitung zur Verfügung.  
Eine Funktion definierst Du mit Hilfe des Schlüsselwortes function gefolgt von einem frei wählbaren Namen und einer Liste von Parametern in einer Klammer. Das Ganze ist die sogenannte Signatur. Dann folgt der Funktionskörper, in dem der eigentliche Code steht, in geschweiften Klammern. Soll die Funktion einen Wert zurückliefern, wird dieser nach dem Schlüsselwort return angegeben.

* Schreibe in die Konsole  
  function isDivisible(dividend, divisor) {return (dividend % divisor == 0)}
* Schreibe nun isDivisible(4, 3), welches Ergebnis wird angezeigt?
* Schreibe nun isDivisible(4, 2), welches Ergebnis erhältst Du jetzt?
* Experimentiere mit anderen Zahlen
* Erkläre, wie die Funktion isDivisible arbeitet. Nutze hierzu das Kapitel “Operatoren” im [EIA2-Booklet](https://jirkadelloro.github.io/EIA2-Inverted/X01_Appendix)

Diese Javascript-Funktion ist sehr unsicher, sie liefert zum Beispiel auch ein Ergebnis, wenn Du isDivisible(false, true) aufrufst, was aber keinen Sinn ergibt. Ordentlich, nach unseren Stilregeln und etwas ausführlicher in TypeScript geschrieben, sieht die Funktion so aus:

function isDivisible(\_dividend: number, \_divisor: number): boolean {

let result: boolean = (\_dividend % \_divisor == 0);

return result;

}

MERKE: % bedeutet der Rest der Division und wenn es glatt teilbar ist, kommt eine Null raus!

Hier hat jeder der Parameter eine Typnotation erhalten, TypeScript wird sich also beschweren, wenn die Funktion mit Werten von anderem Typ als number aufgerufen werden soll. Außerdem wird auch festgelegt, dass die Funktion einen Wahrheitswert zurück liefern soll, dazu wurde die Signatur um die Annotation boolean erweitert.

***Hinweis:****Es ist üblich, dass die Signatur als Ganzes eine Funktion identifiziert. Es kann also innerhalb eines Programms zwei Funktionen gleichen Namens geben, die ganz anderen Code beinhalten, solange sie sich durch ihre Parameterlisten unterscheiden. In Javascript/TypeScript ist das nicht so. Hier ist nur der Name entscheidend, er muss also eindeutig sein.*

***Achtung:****Die Anweisung return führt dazu, dass die Funktion an dieser Stelle verlassen wird. Meist steht daher das return am Ende einer Funktion. Man kann sich diese Mechanik aber auch zu Nutze machen, um unter bestimmten Bedingungen z.B. mit Hilfe einer if-Anweisung eine Funktion frühzeitig zu verlassen (“early out”). Das kann auch eine Funktion ohne Rückgabewerte sein, dann steht das return alleine.*

* Verschaffe dir einen Überblick über das Booklet, jetzt insbesondere über das Kapitel “Coding Style”. Bei der Bearbeitung der Aufgaben ist dieser Stil bindend. Vieles wird sich dir jetzt noch nicht erschließen, später solltest Du aber immer wieder hier nachschlagen.

**Objekt**

Ein Objekt ist ein assoziatives Array, dem Funktionen anhaften. Diese Funktionen können die Elemente des Arrays verändern, ohne dass ihnen Informationen zu dem Objekt mitgegeben werden müssen, denn sie sind ja ein Teil davon und haben Zugriff darauf. Um diese Funktionen von den üblichen zu unterscheiden werden sie Methoden genannt. Ein Objekt verfügt also über Methoden, mit der es sich, oder auch seine Umwelt, verändern kann. In den nächsten Lektionen wirst Du vordefinierte Objekte lediglich nutzen und erzeugen, später wirst Du lernen, wie Du ganz neue Objektstrukturen definieren kannst.

***FunFact:****Tatsächlich ist alles in Javascript im Kern vom Typ Objekt. Selbst die primitiven Datentypen gaukeln nur ihre Primitivität vor, wodurch sie sich einsetzen lassen wie in ‘klassischen’ Programmiersprachen.*

**Werte vs. Referenzen**

Ein wichtiger Unterschied zwischen primitiven und komplexen Datentypen ist die Art und Weise, wie Variablen mit Ihnen verknüpft sind.

Nicht zu beachten, dass komplexe Datentypen als Referenzen adressiert werden, mehrere Variablen also auf den gleichen Datenbestand verweisen können, ist wiederum eine häufige Fehlerquelle.

* ***Achtung:****Diese Referenzierung ist kein Problem, sondern eine wichtige Grundlage für die Anwendungsentwicklung.*

**Fehlertypen**

Du hast nun schon ganz unterschiedliche Fehler kennen gelernt. Die korrekte Benennung dieser hilft anderen dabei, dir zu helfen. Ein wichtiges Unterscheidungskriterium ist dabei der Zeitpunkt, zu dem der Fehler erkannt wird.

**Runtime Error**

Tritt zur Laufzeit auf. Das ist ein hässlicher Fehlertyp, da er unter Umständen lange Zeit unentdeckt bleibt oder nur unter bestimmten Bedingungen reproduzierbar ist. Laufzeitfehler werden in der Browserkonsole angezeigt und müssen unbedingt beachtet werden, auch wenn das Programm weiterlaufen sollte.

**Compiletime Error**

Tritt auf, wenn das TypeScript-Programm in ein Javascript-Programm übersetzt wird. Von diesen Fehlern wirst Du zunächst weitestgehend verschont bleiben, sofern Du dein Projekt ordentlich anlegst. Dieser Fehlertyp wird in der Konsole des Compilers, in der Regel also im VSCode-Terminal angezeigt.

**Designtime Error**

Das sind Fehler, die bereits angezeigt werden während Du den Code schreibst und zwar direkt im Code selbst. In VSCode werden hierzu rote Wellenlinien im Text und rote Kästchen am Rand angezeigt. Außerdem wird in der Console der “Problem”-Tab gefüllt. So kannst Du sofort reagieren und schnell experimentieren um den Fehler zu beheben. Außerdem werden beim Zeigen mit der Maus Fehlerbeschreibungen angezeigt und per QuickFix sogar Lösungen vorgeschlagen.

**Logical Error**

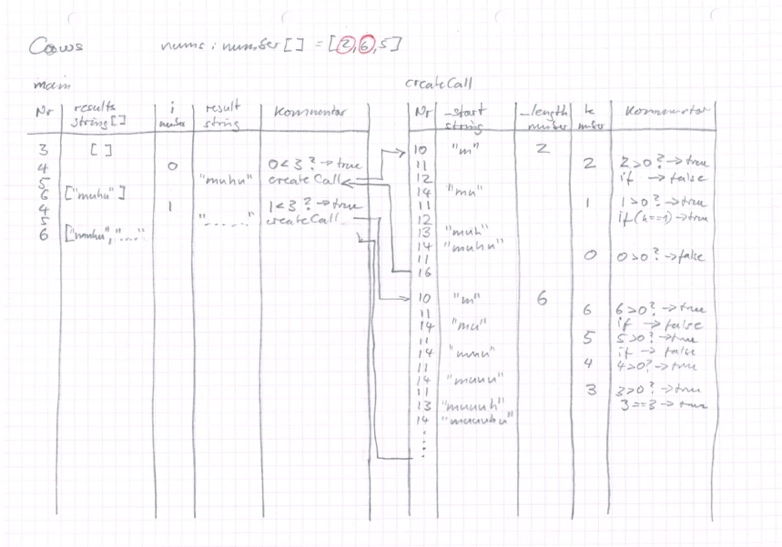
Logikfehler sind nicht auf eine falsche Programmierung zurückzuführen, sondern auf schlechte Konzeption. Diese Fehler kann der Computer meistens gar nicht aufzeigen, da er nicht weiß, was Du eigentlich vorhattest. Solange das Programm ausführbar ist und keiner der anderen Fehlertypen auftritt, wird er es ausführen und davon ausgehen, dass er tut was Du dir vorgestellt hast. Hier hilft nur nachdenken, aber es gibt einige Hilfsmittel, die man dabei einsetzen kann!

***Hinweis:****TypeScript spielt gerade bei der Fehlervermeidung eine seiner großen Stärken aus, indem es viele Fehler zur Entwicklungszeit anzeigt. JavaScript hat hier fast nichts zu bieten, die Fehler zeigen sich erst im Betrieb… besonders gerne beim Nutzer!*

**L01.2 Ablauf und Aktivität**

**Trace Table**

Mit Hilfe einer Verfolgungstabelle kann man den Zustand eines Programms zu jeder Zeit während des Laufs ermitteln und festhalten. Solche Tabellen von Hand zu erstellen ist eine gute Übung um den Programmlauf zu verstehen. Daher exerzieren wir dies im folgenden Video gemeinsam durch. Den Code dazu findest Du hier: [Cows.ts](https://jirkadelloro.github.io/EIA2-Inverted/X00_Code/L01_Recap&Foundation/Cows/Cows.ts)

[](https://jirkadelloro.github.io/EIA2-Inverted/L01_Recap&Foundation/Material/Cows_TraceTable.jpg)

**Debugger**

Natürlich gibt es technische Hilfsmittel etwas ähnliches wie eine Verfolgungstabelle automatisch zu erzeugen. Hierzu verwendest Du den Debugger, also den “Entlauser”.

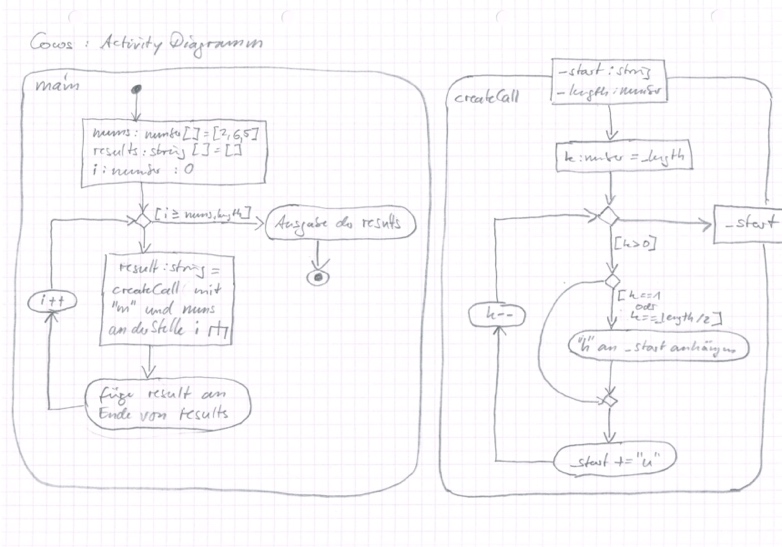
***FunFact:****Der Begriff “Bug” für einen Programmfehler wurde übrigens von der berühmten Computerpionierin Grace Hopper geprägt, die in den 1950 Jahren tatsächlich eine in einem Relais verstorbene Motte für eine Fehlfunktion ihres Großcomputers verantwortlich machen konnte.*

Mit dem Debugger kannst Du auf die Jagd nach Fehlern gehen. Da wir das Programm untersuchen, während es im Browser läuft, ist es sehr praktisch, dass der Browser einen solchen Debugger integriert hat. Unser kleines Programm weist derzeit keine Fehler auf, aber wir schauen trotzdem einmal, ob unsere Verfolgungstabelle mit den Ergebnissen des Debuggers übereinstimmt.

🡪 MERKE: „debugger“ einfach in den Code reinschreiben und dann die Konsole öffnen. Dann hat man verschiedene Möglichkeiten mit dem Debugger umzugehen

**Aktivitätsdiagramm**

Hier wird nun noch das Cows-Programm in ein Aktivitätsdiagramm überführt und dabei die wichtigsten Knoten und Kanten erklärt.

[](https://jirkadelloro.github.io/EIA2-Inverted/L01_Recap&Foundation/Material/Cows_ActivityDiagram.jpg)

## L02 Events

# L02.1 Grundlagen

## Rückblick DOM

Lädt der Browser eine Datei und versucht diese als HTML-Datei zu interpretieren, baut er anhand der Daten im Speicher ein Document-Object-Modell (DOM) auf. Was schließlich im Browserfenster angezeigt wird ist also nicht ein direktes Abbild der Datei, sondern ein Abbild dieses internen Speichermodells.

### DOM-Manipulation

Ein Skript kann das DOM manipulieren, darin Elemente verändern, hinzufügen oder löschen, der Browser kümmert sich automatisch um die Darstellung für den User. Das wurde im ersten Modul dieser Reihe bereits umfassend geübt und auch in der vorangegangenen Lektion wiederholt.

***Achtung:****Die Begriffe Objekt, Element und Knoten können teilweise synonym verwendet werden, es ist aber Vorsicht geboten. ‘Alles’ in Javascript/TypeScript ist Objekt, auch etwas vom Typ number oder string. Ein Knoten ist ein Objekt mit speziellen Eigenschaften und Fähigkeiten, mit dem sich ein Graph aufbauen lässt. Ein Element wiederum ist ein spezieller Knoten, der Eigenschaften eines HTML-Elementes aufweist.*

### Baumstruktur

Das DOM lässt sich, wie auch z.B. die Aktivitätsdiagramme, als Graph mit Knoten, die mit Kanten verbunden sind, darstellen.

Diese Knoten enthalten die Kernfunktionalität zur Bildung des Graphen und damit des DOMs. Jeder Knoten kann auf einen anderen Knoten als parentNode verweisen und auf eine Liste von childNodes. Im DOM ist document der Wurzelknoten, der lediglich eine Referenz auf html in seiner Kinderliste hat. html referenziert über die Eigenschaft parentNode das document und hat in seiner Kinderliste Referenzen auf head und body. bodywiederum referenziert html als Mama bzw Papa und hat wieder verschiedene Kindreferenzen, je nach Inhalt der darzustellenden Seite. Damit ergibt sich eine Baumstruktur, die sich in der Tiefe immer weiter verästeln kann und mit Hilfe der Entwicklertools, wie oben bereits getan, leicht einsehen lässt.

## Ereignisse

Das DOM bietet aber auch ein System für die Interaktion mit dem Nutzer: das Eventsystem. Es stellt äußerst bequem Informationen zu Ereignissen innerhalb der Anwendung zur Verfügung, ohne dass Kenntnisse der Hardware erforderlich sind. Das Betriebssystem und der Browser werten diese Ereignisse bereits aus und bringen die Informationen darüber in eine allgemeine Form.

### Event-Objekt

Events sind spezielle Objekte, die Informationen über ein Ereignis tragen. Ein solches Ereignis kann ein Mausklick sein, ein Tastendruck, eine Berührung des Bildschirms, das Laden einer Datei oder die Beendigung einer Datenübertragung und vieles mehr.

### Target

In der Regel bezieht sich ein Ereignis auf ein bestimmtes Objekt. Zum Beispiel auf den Button, der angeklickt wurde, den Link, der berührt wurde, das Fenster, das den Ladevorgang abgeschlossen hat oder das Textfeld, das verändert wurde. Die Eigenschaft target des Event-Objektes stellt eine Referenz auf dieses Ziel-Objekt zur Verfügung.

### Type

type ist eine simple Zeichenkette und gibt an, was für ein Ereignis beschrieben wird. Hier sind beispielsweise die Werte click, load, change, dragstart und viele weitere vordefiniert. Es ist aber auch möglich eigene, neue Ereignisse zu definieren.

## Event-Handler

Handler sind Funktionen, die ein Ereignis auswerten. Der Umgang damit ist denkbar simpel.

### Handler-Implementation

Um ein Ereignis auszuwerten, implementierst Du einfach eine Funktion, deren Signatur diesem Muster entspricht:

function handlerName(\_event: Event): void {

...

}

Die Funktion nimmt also einen Parameter vom Typ Event entgegen, im Beispiel trägt dieser Parameter den Namen \_event. Auch der Name der Funktion ist frei wählbar, es ist aber zu empfehlen den Prefix “handle” oder abgekürzt “hnd” zu verwenden, z.B. “handleClick”, denn eine solche Funktion, die ein Event verarbeitet, nennt man Handler.

### Listener-Installation

Damit das System weiß, bei welchem Ereignis welcher Handler aufgerufen werden soll, muss der Handler registriert werden. Dies erfolgt mit der bekannten Anweisung addEventListener(...), zum Beispiel so:

document.addEventListener("click", handleClick);

Der erste Parameter ist lediglich die Zeichenkette, die den Typ des Ereignisses beschreibt, der zweite eine Referenz zum Handler. Erhält das document-Objekt nun ein Event-Objekt vom Typ “click”, wird dieses an die Handler-Funktion handleClick weitergeschickt. Das document-Objekt horcht also jetzt in das System hinein, es wurde ihm hierfür ein “Ohr” installiert, ein sogenannter Listener.

***Achtung:****Ein häufiger Fehler in Javascript ist, statt der Referenz einen Funktionsaufruf zu implementieren, z.B. mit addEventListener("click", handleClick()). Die zusätzliche Klammer bewirkt, dass die Funktion bereits bei der Installation aufgerufen wird und deren zurückgeliefertes Ergebnis als Handler-Referenz installiert wird.*

### Beispiel

Das Folgende dürfte das wohl primitivste Beispiel sein, dass wir mit dem Eventsystem darstellen können.

namespace L02\_Load {

window.addEventListener("load", handleLoad);

function handleLoad(\_event: Event): void {

console.log(\_event);

}

}

Hiermit wird das window-Objekt, welches dem Browsertab entspricht in dem die Applikation läuft, angewiesen, die Funktion handleLoad aufzurufen, wenn ein “load”-Event ankommt, und ihr das zugehörige event-Objekt zu übergeben. handleLoad sorgt dann lediglich für die Darstellung des Objektes in der Konsole.

* Untersuche das ausgegebene event-Objekt
* Recherchiere nach dem load-Event, wann genau wird es ausgelöst? Was ist der Unterschied zu DOMContentLoaded?
* Installiere den Listener am document-Objekt, statt am window-Objekt. Was geschieht nun?
* Experimentiere in der gleichen Form mit DOMContentLoaded, wie verhält sich das System nun?

***Hinweis:****Um die Ausführung eines Scripts zu verzögern, steht mittlerweile auch das HTML-Attribut defer zur Verfügung.*

## Event-Phasen

Nicht alle Ereignisse werden allen Objekten im System mitgeteilt. Es ist also nur sinnvoll dort Listener zu installieren, wo sie auch wirken können. Besonders interessant wird das Ganze bei Nutzerinteraktionen, die auf DOM-Objekten ausgeführt werden, wie beispielsweise der Klick auf einen Button. Solche Ereignisse werden nämlich in drei Phasen durch den DOM-Graphen durchgereicht.

### Phase 1: Capture

Das Event-Objekt wird zunächst an das window übergeben. Von dort wandert es zum document, zum html, zum body und weiter in den Baum in Richtung des target.

### Phase 2: Target

Wenn es vom Elternobjekt zum target gereicht wird, befindet sich das Event-Objekt in der Target-Phase.

### Phase 3: Bubble

Schließlich steigt das Event-Objekt im Baum wieder auf, bis es erneut das window erreicht. Es steigt also wie eine Luftblase unter Wasser an die Oberfläche.

### Listener-Options

Bei der Installation des Listeners können mit einem dritten Parameter noch Informationen zur Funktionsweise mitgegeben werden. Wird hier schlicht ein true mitgegeben, reagiert der Listener auf die Capture-Phase. Ansonsten, was üblicher ist, auf die Bubble-Phase. In jedem Fall reagiert er auf die Target-Phase.

### CurrentTarget

Neben dem target trägt das Event-Objekt auch noch eine Referenz auf das Objekt, dessen Listener das Ereignis als letztes gehört hat. Mit currentTarget kann also ausgewertet werden, wo sich das Ereignis gerade im DOM befindet und bearbeitet wird.

### Path

Den kompletten Pfad, den das Event durch das DOM nimmt, kann man im Attribut patheinsehen oder per Skript durch die Methode composedPath() ermitteln.

## CustomEvents

Neben den Ereignissen, die automatisch vom System erzeugt und verschickt werden, ist es auch möglich, explizit Ereignisse durch den eigenen Code erzeugen und verschicken zu lassen. Dabei können auch neue Ereignistypen definiert und beliebige Informationen als detail mitgegeben werden.

// define a custom event that bubbles and carries some information

let event: CustomEvent = new CustomEvent("someSpecialType", {bubbles: true, detail: {someKey: someData}});

// send the event from some dispatcher

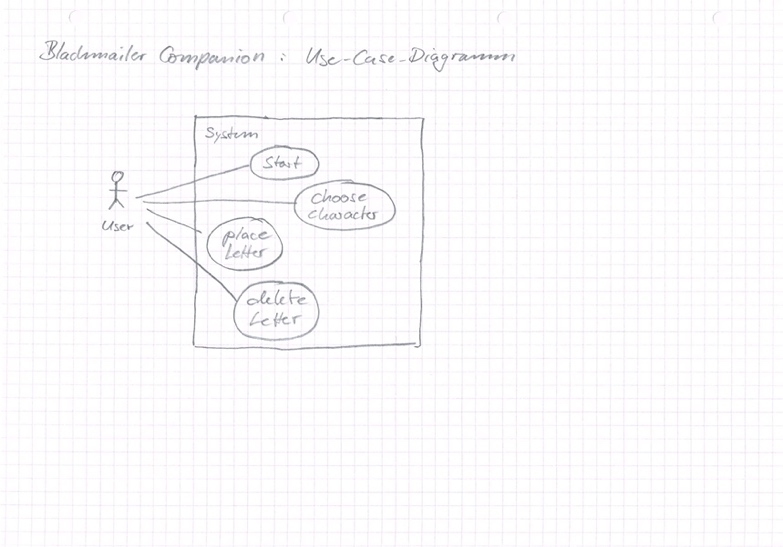
someEventTarget.dispatchEvent(event);

Alle Bezeichnungen, die im Beispiel mit “some” beginnen, können und sollten natürlich mit sinnvollen Bezeichnungen ersetzt werden.

# L02.1 Ereignisgesteuerte Anwendung

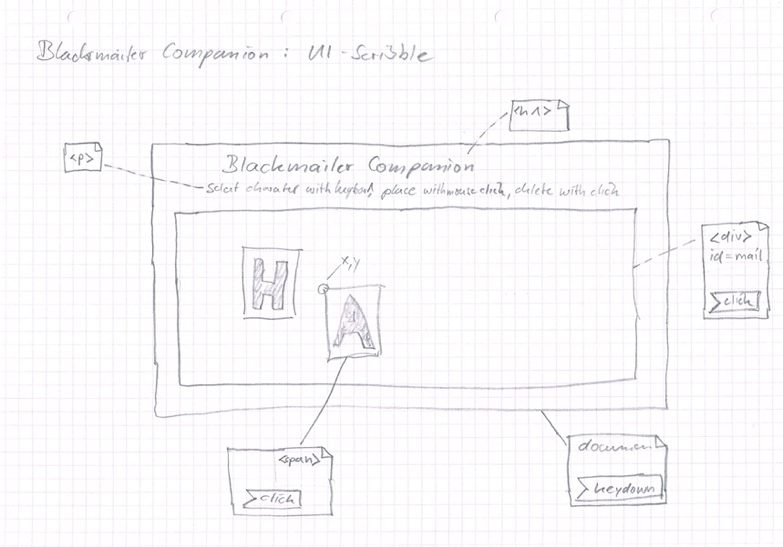
## Anwendungsfalldiagramm (Use-Case-Diagram)

Mit Hilfe des Anwendungsfalldiagramms machst Du dir zunächst einen groben Überblick über die Anforderungen an deine Anwendung. Das geht ganz schnell und hilft ungemein bei der Konzeption.

[](https://jirkadelloro.github.io/EIA2-Inverted/L02_Events/Material/BlackmailerCompanion_UseCaseDiagram.jpg)

## Skizze: User Interface

Als Nächstes machst Du dir eine Skizze des Erscheinungsbildes der Anwendung. Das wird schon einiges über die erforderliche darunterliegende Struktur verraten. Die Skizze versiehst Du schon mit den HTML-Auszeichnungen und Eigenschaften, die dir dabei sinnvoll erscheinen. Unterscheide dabei zwischen statischen und dynamischen Elementen und Eigenschaften. Für die Dynamik trägst Du hier schon ein, an welchen Elementen Listener installiert werden soll und welche Ereignisse dabei mit welchen Aktivitäten verknüpft werden. Prüfe, ob alle Interaktionsmöglichkeiten zur Realisierung der Anwendungsfälle gegeben sind.

[](https://jirkadelloro.github.io/EIA2-Inverted/L02_Events/Material/BlackmailerCompanion_UI-Scribble.jpg)

## Aktivitätsdiagramme

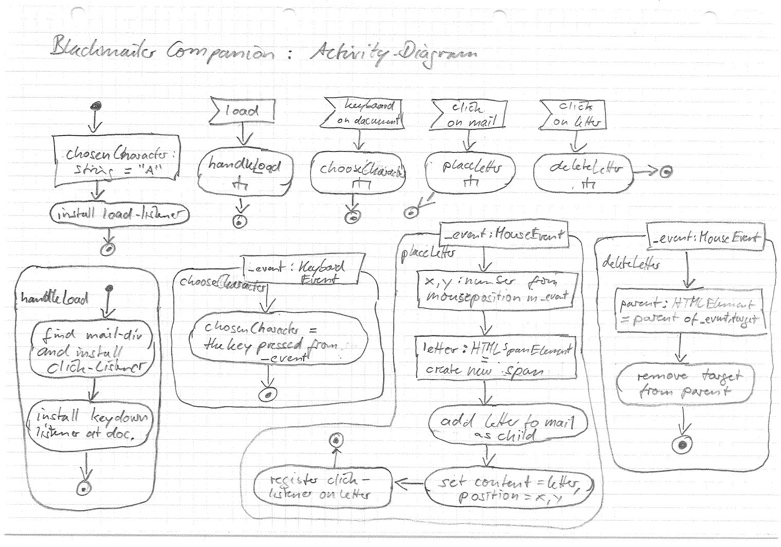
Jetzt hast Du bereits aus der Sicht des Nutzers die wesentlichen Aktivitäten, die beteiligten Elemente und die auszuwerteten Ereignisse festgehalten. Nun wechselst Du auf die Sicht aus dem System heraus und legst fest, wie es arbeiten soll. Dazu nutzt Du jetzt Aktivitätsdiagramme. Ein Event bildet dabei jeweils als Signalempfang einen Startknoten für eine Aktivität.  
Beginne dabei zunächst wieder mit einer Übersicht über die Aktivitäten. Nimm dir dann nacheinander die einzelnen Aktivitäten vor und verfeinere sie. Wiederhole diesen Prozess, bis Du zu den atomaren Aktionen gekommen bist die sich in Programmanweisungen umsetzen lassen. Am Anfang musst Du hierzu wahrscheinlich nach diesen Anweisungen noch etwas recherchieren.

[Ein Bild, das Text, Whiteboard enthält.

Automatisch generierte Beschreibung](https://jirkadelloro.github.io/EIA2-Inverted/L02_Events/Material/BlackmailerCompanion_ActivityDiagram.jpg)

## Implementation

Wenn Du den Eindruck hast, mit deiner Konzeption alles für eine erste Implementation der Anwendung berücksichtigt zu haben, kannst Du dich daran machen.

[](https://jirkadelloro.github.io/EIA2-Inverted/L02_Events/Material/BlackmailerCompanion_ActivityDiagram2.jpg)

## Testing

Während der Implementation wird das Programm immer wieder getestet. Es ist wichtig möglichst so zu implementieren, dass nicht erst ein abschließender Test Fehler zu Tage fördert, sondern dass immer lauffähige Zwischenstände existieren, die entsprechend in das Code-Repository aufgenommen werden.

## Iterative Arbeitsweise

Bedenke, dass Du die Konzeption jederzeit verbessern kannst und solltest, wenn dir auffällt, dass etwas fehlt oder nicht funktioniert. Dasselbe gilt für alle vorangegangenen Schritte. So können Überlegungen beim Zeichnen der Skizze zurück auf das Use-Case-Diagramm reflektieren, oder Erkenntnisse bei der Implementation die Änderung der Skizze nach sich ziehen, was dann wiederum die Aktivitätsdiagramme beeinflusst. Es ist unwahrscheinlich, dass der erste konzeptionelle Ansatz sich unverändert durchführen lässt und zur perfekten Anwendung führt.

***Achtung:****Was Du auf keinen Fall machen darfst, ist auf der letzten Stufe, der Implementation, gravierende Änderungen vorzunehmen, ohne zuvor die Konzeption neu aufbereitet zu haben. Diese Vorgehensweise führt mit großer Sicherheit zum****Misserfolg!***

## Übungen

* Löse zunächst das Problem mit dem erneuten Aufruf von placeLetter nach deleteLetter
  + experimentiere erst mit dem Lösungsoptionen 1 und 2.
  + Plane dann Lösung 3 und implementiere sie.
* Erweitere die Erpresserbrief-App so dass sie auch auf Smartphone anwendbar ist
  + In einem weiteren Bereich sollen alle zulässigen Buchstaben zur Auswahl bereit stehen, so dass man sie per Touch aktivieren kann
  + Der aktuell ausgewählte Buchstabe soll in dieser Auswahl hervorgehoben erscheinen, so dass der Nutzer weiß, was er gewählt hat, bevor er den Brief ändert
* Hinweis: recherchiere nach dem Pointer-Event

# L03 FormElements

# In diesem Kapitel lernst Du die wichtigsten Elemente kennen, erfährst, wie Du sie nutzen kannst und welche Informationen sie liefern. Ähnliche Elemente findet man auch in anderen Entwicklungsumgebungen zur Gestaltung von User-Interfaces.

* *Bringe den Code*[*L03\_FormElements/Overview*](https://github.com/JirkaDellOro/EIA2-Inverted/tree/master/X00_Code/L03_FormElements/Overview)*aus dem Anhang bei dir lokal zum Laufen. In dieser Lektion wirst Du seine Funktionalität nach und nach freischalten und untersuchen. Zunächst erscheint nur die Überschrift “Form-Elements”.*

## Texteingabe

Die wichtigste Eingabemöglichkeit ist die von Text. Mit dem Keyboard-Event könnte man zwar beim Tippen jeden Buchstaben abfangen und damit eine Zeichenkette zusammen setzen, es geht aber sehr viel einfacher. Das generische input-Element bringt in seiner Urform als Textfeld schon diese Grundfunktionalität, kann aber noch viel mehr.

*Beispiel:*

### *Platzhalter*

*Dem Attribut placeholder kann ein Wert zugewiesen werden, der dem Nutzer beispielsweise als Eingabeaufforderung dienen kann. Sobald der Nutzer die Eingabe beginnt, verschwindet der Platzhalter automatisch und wird wieder angezeigt, wenn das Feld geleert wird.*

### *Passworteingabe*

*Der Wert “password” für das Attribut type bewirkt, dass die Eingabe nicht angezeigt wird und auch nicht kopiert werden kann.*

### *Prüfung*

*Zum Beispiel wird durch die Angabe von “email” oder “url” als Wert für das Attribut typeschon eine einfache Prüfung der eingegebenen Zeichenkette vorgenommen um gröbste Eingabefehler zu erkennen und die Ungültigkeit anzuzeigen. Wird das ‘wertlose’ Attribut required angegeben, wird zudem angezeigt, dass die Eingabe zwingend erforderlich ist. Die Darstellung kann per CSS und den Pseudo-Selektoren :invalid, :valid und :required angepasst werden.  
Eine deutlich komplexere Überprüfung kann mit Hilfe von regulären Ausdrücken erfolgen, was allerdings einige Expertise erfordert. Hierzu wird im Attribut pattern der zu verwendende reguläre Ausruck festgelegt. Im Beispiel werden genau 3 Ziffern eingefordert.*

### *Textarea*

*Eine größere Variante des input-Elementes ist die textarea. Hier können längere Texte über mehrere Zeilen eingegeben werden, bei Überlauf erscheint eine Laufleiste und der Nutzer kann das Feld ggf. in seiner Größe vom Nutzer verändern.*

### *Übungen*

* *Aktiviere jetzt in der HTML-Datei das Fieldset mit der id “fsText”. Entferne hierzu die umschließenden Kommentar-Auszeichnungen <!-- und -->. Am einfachsten geht das, indem Du den ganzen Absatz markierst und dann Ctrl+#drückst (Mac: ⌘+K ⌘+U). Speichere die Datei.*
* *Das Fieldset mit der Bezeichnung “Text-Input” sollte nun bei der aktualisierten Anzeige der Datei im Browser erscheinen. Prüfe das Verhalten der Elemente und vollziehe nach, wie es zustande kommt. Bedenke, dass noch kein Skript am Werk ist.*
* *Aktiviere nun das Skript “FormElements.js” in der HTML-Datei. Bediene dann weiter die Elemente und beobachte die Ausgabe in der Konsole. Wann wird das input-Event verschickt, wann change?*
* *Untersuche den Code in der Datei FormElements.ts und kläre, wie er funktioniert.*

## Selektionen

Häufig soll der Nutzer auch zwischen verschiedenen Möglichkeiten auswählen, ohne selbst Text einzugeben. Hierzu müssen ihm die Auswahlmöglichkeiten präsentiert und eine eindeutige Interaktionsmöglichkeit angeboten werden. HTML bietet hier verschiedene Elemente bzw. Elementtypen an.

### Datalist

Hiermit kann einem Textfeld eine Liste von vordefinierten Einträgen zugewiesen werden. Der Nutzer kann per Zeigerinteraktion einen solchen Eintrag auswählen oder durch Eintippen auf der Tastatur die Auswahl einschränken. Er kann aber immer noch auch Freitext eingeben und die Auswahl ignorieren. Die Kopplung von input und datalist geschieht durch die Attribute list und respektive id.

### Radiobutton

Beispiel:   
Dieser Typ ist speziell für Single-Choice-Auswahlen gemacht. Alle input-Elemente des Typs radio, welche den gleichen Wert im Attribut name tragen, werden als Gruppe behandelt. Nur ein Element der Gruppe kann markiert sein.

### Checkbox

Beispiel:   
Mit input-Elementen des Typs checkbox werden dagegen klassischerweise Multiple-Choice-Auswahlen dargestellt.

***Achtung:****Radiobuttons und Checkboxes beziehen sich in der Regel auf einen nebenstehenden Text. Für den Nutzer ist es sehr ärgerlich, wenn diese Beziehung nicht auch zur Unterstützung der Interaktion genutzt wird, er also nicht auf den Text klicken kann um zu interagieren. Dabei ist es äußerst einfach dies zu berücksichtigen, der Text muss lediglich von einem label-Tag umschlossen sein. Die Kopplung von label- und input-Element geschieht dann einfach mit Hilfe der Attribute for und id. Dies sollte****immer****implementiert werden!*

### Select

Beispiel:  1. 2. 3.

Am mächtigsten ist das select-Element, auch ‘Dropdown-Menü’ oder ‘ComboBox’ genannt. Auf engem Raum kann eine vordefinierte Auswahl an Optionen als Single- oder auch Multiple-Choice Auswahl angeboten werden. Dabei können sogar noch Untergruppierungen vorgenommen werden.

### Übungen

* Aktiviere nun das zweite Fieldset, es trägt die id “fsSelection”. Experimentiere im Browser mit den Selektions-Elementen.
* Beobachte in der Konsole genau die Zuordnung von Elementname und Wert. Was fällt dir auf? Wo kommen die Werte her? Vergleiche mit den Texteingaben.
* Aktiviere nun im Skript den Code zu “// Handling checkbox”. Warum ist dieser Code hier, was ist dessen Sinn? Beobachte die Ausgabe in der Browserkonsole.
* Ergänze das select-Tag um das ‘wertlose’ Attribut multiple. Wähle nun im Browser dort bei gedrückter Ctrl- oder Shift-Taste mehrere Optionen. Beobachte erneut die Ausgabe und halte deine Beobachtung fest.

## Weitere Eingabeelemente

Es gibt noch einige Eingabeelemente, die sich gänzlich anders verhalten. Ihre grundlegende Funktionsweise erschließt sich recht schnell und einfach.

* Aktiviere das Fieldset mit der id “fsSpecial”. Experimentiere damit, verändere die Attributwerte und bediene die Elemente im Browser. Beobachte die Konsolenausgaben.
* Beantworte die Frage nach der Wirkung des value-Attributs

## Ausgabeelemente

Es gibt zudem noch ein paar standardisierte Elemente, welche dem Nutzer Rückmeldung geben. Sie dienen also der Interaktion des Systems mit dem Nutzer.

* Aktiviere das letzte Fieldset, es hat die id “fsOutput”.
* Aktiviere auch die letzten Codefragmente. Erkläre, wie die Ausgabeelemente angesteuert werden.

# Formulargesteuerte Anwendung

Ein Bild, das Text, Königin, ClipArt, Schild enthält.

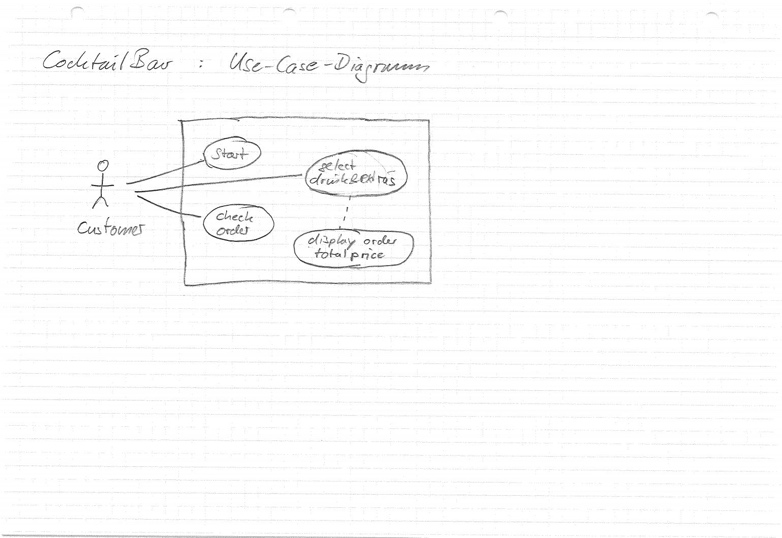
Automatisch generierte Beschreibung

Wie oft sitzt man nicht abends zuhause und wünscht sich einen richtig leckeren Cocktail. Aber die Zutaten fehlen, es ist zu aufwendig damit anzufangen und raus will man auch nicht mehr? Da drängt sich doch die nächste geniale Geschäftsidee auf: die **Online-Cocktailbar!**

Kunden können hier mit Hilfe eines Webformulars spontan einen Cocktail bestellen, der Ihnen wenige Tage später per Post nach Hause geliefert wird.

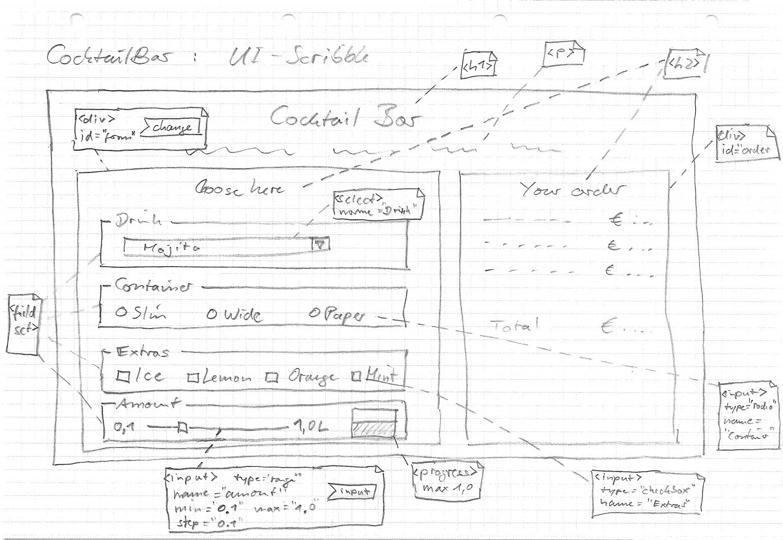
## Anwendungsfalldiagramm (Use-Case-Diagram)

Zeichne wieder ganz schnell ein kleines Anwendungsfalldiagramm. Versuche es selbst, bevor Du dir das Video anschaust.

[](https://jirkadelloro.github.io/EIA2-Inverted/L03_FormElements/Material/CocktailBar_UseCaseDiagram.jpg)

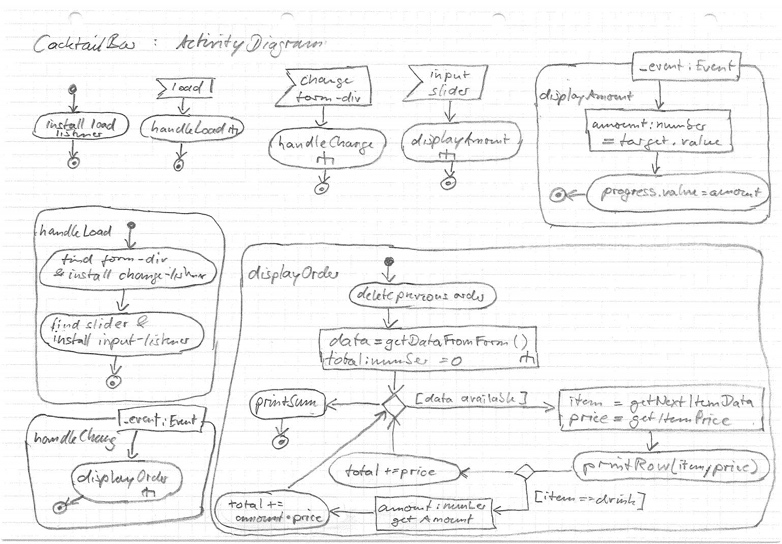
## Skizze: User-Interface

Das User-Interface wird nun schon ein wenig komplexer. Erstelle eine Skizze und überlege, welche Tags, Attribute und Listener Du brauchst. Investiere ein wenig Zeit um selbst einen ersten Entwurf zu gestalten, bevor Du das Video anschaust.

[](https://jirkadelloro.github.io/EIA2-Inverted/L03_FormElements/Material/CocktailBar_UI-Scribble.jpg)

## Aktivitätsdiagramme

Nun ist schon grob konzipiert, was geschehen soll und wie sich das darstellt. Entwickle nun wieder ein Aktivitätsdiagramm mit dessen Hilfe Du festlegst, wie das Ganze funktioniert. Beginne wieder mit den Ereignissen und arbeite dich vom Groben ins Feine. Mach’ dir am Anfang also noch nicht zu viele Gedanken über Details, sondern zerlege große Probleme in kleinere. Versuche es selbst, bevor Du das Video anschaust.

[](https://jirkadelloro.github.io/EIA2-Inverted/L03_FormElements/Material/CocktailBar_ActivityDiagram.jpg)

## Implementation I

Versuche dich nun an der Implementation. Schaue dir das Video spätestens an, wenn Du dich an die Zusammenfassung der Bestellung machst. Hier gibt es noch einige Tipps.

# FormData

Javascript stellt einen einfachen Mechanismus zur Verfügung um Formularelemente automatisch auszuwerten und auf die Ergebnisse zuzugreifen. Hierzu müssen die Formularelemente in der DOM-Laufzeithierarchie einem form-Element untergeordnet sein. Wird bei der Erzeugung eines Objekts des Typs FormData nun ein Verweis auf auf dieses form-Element übergeben, so werden die Werte der name- und value-Attribute als Schlüssel-Werte-Paare zur Verfügung gestellt.

let formData: FormData = new FormData(document.forms[0]);

Das document-Objekt stellt bereits bequem eine Liste aller untergeordneten form-Elemente zur Verfügung. In obigem Beispiel wird also das erste Formular des Dokuments ausgewertet.

## name-Attribut

Bislang hatte das name-Attribut nur für die Gruppierung von Radiobuttons eine Rolle gespielt. Tatsächlich stellt es aber für die automatische Auswertung von Formularen das maßgebliche Zuordnungskriterium dar.

***Achtung:****nicht das Attribut id ist für Formularelemente ausschlaggebend, sondern name*

Zu beachten ist auch, dass das name-Attribut, anders als id nicht eindeutig sein muss. Bei der Auswertung kann derselbe Name also mehrfach als Schlüssel auftauchen und jeweils unterschiedliche Werte tragen.

## Auslesen

### get(...)

Bei eindeutigen und bekannten Namen lassen sich die Werte der value-Attribute mit Hilfe der Objektmethode get(...) wie bei einem assoziativen Array auslesen. Zum Beispiel so:

console.log(formData.get("Drink"));

### entries()

Da aber die Namen nicht zwingend eindeutig sind und sie auch nicht unbedingt im Code reproduziert werden sollen, ist es häufig sinnvoll, über alle Einträge im FormData-Objekt zu iterieren. Die Objektmethode entries() eines FormData-Objektes liefert alle gefundenen Schlüssel-Werte-Paare. Mit einer for..of-Schleife können diese bequem nacheinander behandelt werden:

for (let entry of formData) {

console.log(entry);

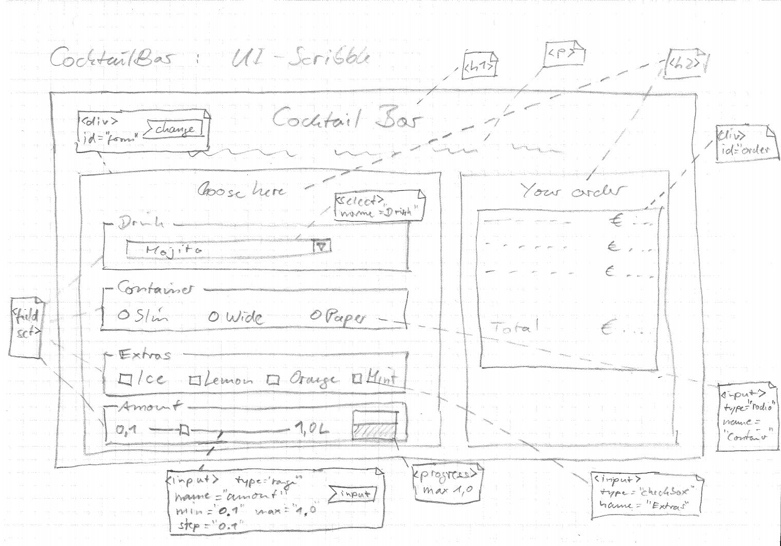
console.log("name: " + entry[0]);

console.log("value: " + entry[1]);

}

Jedes entry-Objekt ist also ein kleines Array mit zwei Einträgen, an Stelle 0 ein String mit dem der Wert des name-Attributs des ursprünglichen Form-Elements, und an Stelle 1 den entsprechenden Wert des value-Attributs. Letzterer ist hier vom Typ FormDataEntryValue, eine etwas besondere Zeichenkette, und muss gegebenenfall konvertiert werden.

## Implementation II

[](https://jirkadelloro.github.io/EIA2-Inverted/L03_FormElements/Material/CocktailBar_UI-Scribble2.jpg)

* Die Ausgabe der Bestellübersicht funktioniert noch nicht richtig. Finde den Fehler und korrigiere ihn.
* Implementiere die Sonderbehandlung der Drinks.
* Implementiere die Berechnung der Gesamtsumme
* Überprüfe die Konzeption und korrigiere gegebenfalls die Diagramme entsprechend der gemachten Erfahrungen.

Hinweis: Ungenauigkeiten bei der Berechnung von Fließkommazahlen können sehr schnell zu hässlichen Zahlendarstellungen führen. Der Typ number verfügt aber über mehrere Methoden zur Umwandlung in eine formatierte Zeichenkette. Recherchiere danach und nutze sie.

## L08 Canvas

# L08.1 Grundlagen

Ein Bild, das Text, Mann, Person, draußen enthält.

Automatisch generierte Beschreibung Quelle: <http://pix4.qmde.de/www.quotenmeter.de/pics/sonstiges/2019/medien/bobross_ox__W1000xh0.jpg>

Die standardisierten HTML-Elemente vereinfachen die Gestaltung einer Web-Applikation enorm, und Formular-Elemente unterstützen sogar komplexe Interaktion in einer den meisten Nutzer bereits geläufigen Form. Allerdings sind die Gestaltungsmöglichkeiten dadurch recht eingeschränkt. Ebenso übliche Formen der Informationsvermittlung, wie beispielsweise Diagramme, lassen sich damit nicht zur Laufzeit erzeugen. Hierfür nutzt man das HTMLCanvasElement!

***FunFact:****Canvas heißt so viel wie Plane, Segel, Markise… aber auch Leinwand.*

Das HTMLCanvasElementstellt eine rechteckige Fläche auf einer Browserseite zur Verfügung, deren einzelne Bildpunkte (Pixel) mit Hilfe von Zeichenbefehlen beliebig eingefärbt werden können. Innerhalb dieser Fläche lassen sich frei Formen und Farben dynamisch generieren.

***FunFact:****Pixel ist ein Kunstwort, dass sich aus****Pic****ture und****El****ement oder****Cell****zusammensetzt. Eine Rastergrafik, wie sie unsere modernen Monitore darstellen, unsere Kameras liefern oder unsere Webseiten aufbauen, sind aus einer Vielzahl solcher farbiger Pixel zusammengesetzt.*

## CanvasRenderingContext

Die Befehle für diese Bildmanipulation stellt das HTMLCanvasElement aber nicht direkt zur Verfügung, sondern es bietet hierfür spezielle Programmierschnittstellen (Application Programming Interfaces, API) an, die in diesem Zusammenhang CanvasRenderingContext genannt werden. Sie können sehr umfangreich sein und es gibt sie sowohl für 2D- als auch für 3D-Grafikdarstellung z.B. mit WebGL. Mit dem folgenden Code wird ein solcher CanvasRenderingContext für die Bildmanipulation mit zweidimensionalen Zeichenkommandos von einem im DOM vorhandenen Canvas angefordert und eine Referenz darauf gespeichert.

let canvas: HTMLCanvasElement = document.querySelector("canvas");

let crc2: CanvasRenderingContext2D = canvas.getContext("2d");

* Erstelle eine HTML-Datei und lege dort mit <canvas> ein Canvas-Element an.
* Erstelle ein TypeScript-Programm, welches den RenderingContext des Canvas-Elementes anfordert und eine Referenz in einer Variable namens crc2 speichert, wie oben angegeben.
* Gib in einer weiteren Zeile crc2. ein und verschaffe dir einen Eindruck von der Menge der Zeichenbefehle, die dir nun zur Verfügung stehen.

Die Pixel des Canvas-Elements werden beim Aufbau des DOMs zunächst mit der Farbe #FFFFFF, also weiß, besetzt. Daher fällt der Canvas auf weißem Hintergrund nicht ins Auge. Der einfachste Befehl, den Du nutzen kannst um das zu ändern, ist fillRect(), wobei zunächst die Füllfarbe als Eigenschaft fillStyle des RenderingContexts definiert werden muss.

crc2.fillStyle = "#FF0000";

crc2.fillRect(0, 0, crc2.canvas.width, crc2.canvas.height);

* Finde heraus, welche Bedeutung die Parameter der Methode fillRect haben. Experimentiere damit.

## Pfade

fillRect(...), clearRect(...) und strokeRect(...) sind die einzigen Zeichenbefehle, die sofort ein sichtbares Ergebnis liefern. Komplexere Formen definierst Du zunächst mit Hilfe eines Pfad-Objektes.

### Vektorgrafik

Wenn auch das entstehende Bild am Ende als Rastergrafik vorliegt, die Definition der Pfade erfolgt nach den Regeln der Vektorgrafik. Das bedeutet, dass Du nicht einzelne Pixel manipulierst, sondern mathematische Beschreibungen der zu zeichnenden Formen und ihrer Koordinaten formulierst. Die Rasterung übernimmt dann der RenderingContext wenn die Zeichenmethoden stroke() und fill() aufgerufen werden. Daraufhin erst werden die Pixel, die innerhalb oder am Rande des Pfades liegen nach der zuvor definierten Füll- oder Linienvorschrift mit Farbe versehen.  
Es gibt allerdings auch Methoden zur direkten Manipulation von Pixeln, die in dieser Lektion aber keine Rolle spielen sollen.

### Arc

crc2.beginPath();

crc2.arc(100, 100, 20, 0, 1.5 \* Math.PI);

crc2.closePath();

crc2.stroke();

* Erkunde, was arc bewerkstelligt und was die Parameter bedeuten.
* Finde heraus, wie Du die Linienfarbe ändern kannst.
* Was geschieht, wenn die Anweisung closePath() nicht ausgeführt wird. Warum?

### Ellipse

Lange Zeit war es erforderlich, Ellipsen als verzerrte Kreise darzustellen oder mit Kurven anzunähern. Mittlerweile gibt es aber auch eine ellipse-Anweisung.

* Finde heraus, wie die Ellipse funktioniert…

### Linienzüge

Mit den Anweisungen moveTo(...) und lineTo(...) kannst Du einen Pfad um Linienzüge erweitern. Eine Linie wird dabei nur durch die Endposition definiert, die Startposition ist die Endposition der vorangegangenen Anweisung.

* Lasse ein Dreieck zeichnen.

### Kurven

Mit moveTo(...) und lineTo(...) kannst Du beliebige eckige Formen darstellen. Wenn es geschmeidiger werden soll, nutzt Du quadratische oder Bezier-Kurven.

* Experimentiere mit [dieser Anwendung](https://jirkadelloro.github.io/EIA2-Inverted/X01_Appendix/Canvas/Curves/Curves/start.html). Beachte, dass für quadratische Kurven außer dem vorangegangenen Endpunkt des aktuellen Pfades zwei weitere, für Bezierkurven drei weitere Punkte angegeben werden müssen.

### Text

Alternativ zu einer Textüberlagerung durch ein weiteres HTML-Element, kann Text auch durch den RenderingContext auf den Canvas gebracht werden. Hierzu dienen die Methoden fillText(...) und strokeText(...) und weitere Methoden zur Steuerung der Textausgabe.

### Weiteres

Der CanvasRenderingContext bietet noch einige Zeichen- und Stilmittel mehr, studiere hierzu dieses [CheatSheet](https://jirkadelloro.github.io/EIA2-Inverted/X01_Appendix/Canvas/HTML5_Canvas_Cheat_Sheet.pdf). Damit erhältst Du schnell einen Überblick und kannst die Syntax nachschlagen. Aber auch das ist nicht vollständig. Im Internet findest Du bei Bedarf noch zahlreiche weitere Quellen.

### Pfadobjekte

Bei Verwendung der Pfad-Methoden direkt auf dem RenderingContext, wird ein globales Pfadobjekt manipuliert. Mit beginPath() wird der darin enthaltene alte Pfad gelöscht und ein neuer angelegt.  
Es ist aber auch möglich, mit new Path2D() individuelle Pfadobjekt zu erzeugen und die Pfadanweisungen darauf auszuführen. So kann der Pfad gespeichert und im Laufe des Programms wiederverwenden werden, ohne dass der Algorithmus zur Pfaderstellung wieder durchlaufen werden muss. Zum Zeichnen eines solchen Pfadobjektes wird es schließlich als Parameter der Zeichenmethode einfach mitgegeben.

let path: Path2D = new Path2D();

path.arc(60, 60, 50, 0, 2 \* Math.PI);

crc2.stroke(path);

## Rendering

“Render” ist ein relativ unspezifisches Wort und bedeutet lediglich so viel wie “ausführen, machen, tun”. Im Zusammenhang mit digitalen Medien ist meist die Erzeugung eines Bildes anhand gegebener Daten gemeint. Es gibt unterschiedliche Ansätze hierfür.

***FunFact:****Render als Substantiv heißt auch “Putz”. Die in der 3D-Grafik häufig genannte “Renderpipe” könnte man also falsch als “Putzpfeife” übersetzen.*

### Retained Mode

Es wird zunächst ein Modell der gesamten darzustellenden Szene erzeugt, was fortwährend manipuliert werden kann. Das Rendering-API kümmert sich dann darum, die Szene als Ganzes zu rendern. Ein dir wohlbekanntes Beispiel für ein solches Modell das DOM. Es wird automatisch gerendert und so die visuelle Darstellung erzeugt, die der Nutzer sieht.

### Immediate Mode

CanvasRenderingContext aber arbeitet im Immediate Mode. Es gibt keinerlei Verwaltung für darzustellende Bildteile. Wann, wo und wie jeder Pfad gerendert wird, obliegt vollständig dem Programm, welches den Context nutzt. Sobald ein Zeichenkommando abgearbeitet wird, verändern betroffene Pixel im Zielbild ihre Farbe.

***Achtung:****Da der Canvas wiederum ein DOM-Element ist, muss der Browser Gelegenheit bekommen, das fertige Bild auf der Seite darzustellen. Eine Endlosschleife beispielsweise wird ihn daran hindern.*

### Maleralgorithmus

Das bedeutet, das nachfolgende Zeichenkommandos die Wirkung der vorangegangenen überschreiben können. Das Bild muss also “von hinten nach vorne” aufgebaut werden. Bildteile, die im Hintergrund liegen sollen, werden zuerst gezeichnet, danach die Bildteile im Vordergrund. Das ist der sogenannte Maleralgorithmus, denn ein Maler arbeitet am effizientesten ebenfalls so.

* Schaue dir Malvideos im Internet an, vielleicht von oben dargestelltem Bob Ross. Beachte, in welcher Folge er die Bildbestandteile malt.

## Koordinatensystem

Das Standard-Koordinatensystem hat seinen Ursprung in der linken oberen Ecke des Canvas, positive Koordinatenwerte steigen horizontal nach rechts an, vertikal nach unten. Angaben sind direkt bezogen auf Pixel, wobei die Koordinate (0.5,0.5) den ersten sichtbaren Pixel ganz links oben referenziert, die Koordinate (canvas.width-0.5, canvas.height-0.5) den letzten sichtbaren Pixel rechts unten.

***Achtung:****Wie immer bei den digitalen Medien beginnen wir die Zählung mit der 0, daher liegt bei einem Canvas mit der Größe 300x200 der Pixel mit der Koordinate (300, 200) nicht mehr im sichtbaren Bereich.*

Als Koordinatenwerte sind nicht nur Ganzzahlen zulässig, sondern auch Fließkommazahlen. Je nach Einstellung des Canvas wird dann beim Rendern der nächste Pixel gefüllt, oder die Farbe halbtransparent auf die umgegebenden Pixel verteilt. Um einen Pixel tatsächlich genau mittig zu treffen, kann es erforderlich sein, Nachkommastellen zu nutzen. Der untenstehende Code erzeugt das nebenstehende Bild (stark vergrößert).

Ein Bild, das Text, ClipArt, Vektorgrafiken enthält.

Automatisch generierte Beschreibung

crc2.beginPath();

crc2.moveTo(2.1, 1);

crc2.lineTo(2.1, 10);

crc2.moveTo(4.5, 1);

crc2.lineTo(4.5, 10);

crc2.moveTo(7.5, 1);

crc2.lineTo(10.5, 10);

crc2.stroke();

Eine schräge Linie verläuft immer durch viele Zwischenpositionen. Würden nur die dabei am stärksten berührten Pixel gezeichnet, entsteht ein Treppen-Effekt (Aliasing). Bei der Farbverteilung (Anti-Aliasing) wird dies kaschiert, es entstehen durch die Transparenzen aber neue Farben im Bild und die Linie wird unscharf gezeichnet.

## Transformation

Sollen ähnliche Bildbestandteile, z.B. Bäume, Häuser, Schneeflocken etc., mehrfach in einem Bild auftauchen, ergibt sich ein Problem. Da die Pfade, mit deren Hilfe diese Bestandteile gezeichnet werden, mit absoluten Koordinaten bezogen auf das Standard-Koordinatensystem konstruiert werden, sind diese Informationen fest im Pfad definiert. Selbst wenn keine literalen Werte als Parameter übergeben sondern Variablen genutzt werden, so dass der Code zur Konstruktion wiederverwendet werden kann, muss der komplette Pfad erneut konstruiert werden, wenn der gleiche Bildbestandteil an einer anderen Stelle auftauchen soll.  
Hier schaffen Transformationen Abhilfe. Anstatt die Koordinaten und Dimensionen neu anzugeben, kann mit Transformationen das Koordinatensystem verändert werden, auf das sich die Angaben beziehen.

[Transformator-App (Flash-Player erforderlich)](https://webuser.hs-furtwangen.de/~del/Apps/Transformation/Transformation.html)

* Experimentiere mit der Transformator-App. Erzeuge drei unterschiedliche und aktive Transformationen und beobachte die Auswirkungen.
* Vertausche sie durch Klick auf das Doppelpfeil-Symbol. Erkläre, warum das Haus des Nikolaus seine Position oder Erscheinung dabei ändert.

Die Transformationen werden also nacheinander ausgeführt und jede weitere Transformation verändert das Ergebnis der vorangegangenen. Tatsächlich existiert nur eine Transformationsmatrix, welche die komplette Transformationskette darstellt. Jedesmal, wenn translate(...), rotate() oder scale() aufgerufen wird, wird diese Matrix verändert. Bei der Erstellung oder beim Zeichnen eines Pfades wird die Gesamtmatrix mit einer einfachen Matrixmultiplikation auf die Koordinaten angewendet.  
Das bedeutet, dass immer wiederkehrende Transformatsaufrufe in der Matrix kumulieren, was in der Regel nicht gewünscht ist. Für dieses Problem gibt es mehrere Lösungsansätze:

* mit resetTransform() die Gesamtmatrix wieder auf den Urzustand bringen
* mit save() den aktuellen Zustand der Gesamtmatrix speichern, und diesen Zustand später mit restore() wieder herstellen. Dabei kann auch mehrfach save() aufgerufen werden, die dabei gespeicherten Zustände werden per restore() in umgekehrter Reihenfolge wieder restauriert.
* mit getTransform() den aktuellen Zustand der Gesamtmatrix speichern und einer Variable vom Typ DOMMatrixzuweisen. Mit setTransform(...) und der Variable als Parameter wird dieser Zustand wieder hergestellt

## Stil

### Farbe

Der CanvasRenderingContext kann Farbangaben als Zeichenketten verarbeiten, wie sie ebenso in CSS3 verwendet werden. Hier hat man also die Auswahl der Farbkodierung und kann die heranziehen, die sich für die Aufgabe am besten eignet. Für eine Serie von Bildelementen, die mit gleicher Farbe aber unterschiedlicher Helligkeit dargestellt werden sollen, ist die Angabe im HSL-Format deutlich leichter zu bewerkstelligen als in den anderen Formaten. Folgende Zeichenketten produzieren alle das gleiche Lachsrot

| **Format** | **Zeichenkette** |
| --- | --- |
| Hexadezimal | #FA8072 |
| RGB | RGB( 250, 128, 114 ) oder RGB( 98%, 50%, 45% ) |
| HSL | HSL( 6, 93%, 71% ) |
| Predefiniert | salmon |

RGB und HSL erlauben (dann auch als RGBA und HSLA bezeichnet) noch einen vierten Parameter, nämlich die Opazität (Alpha) die als Fließkommazahl zwischen 0 und 1 angegeben wird. Zudem ist auch die weniger präzise Angabe mit nur drei hexadezimalen Ziffern möglich.

### Farbverlauf

Zwei Arten von Farbverläufen (Gradients) sind möglich, der lineare und der radiale. Beim linearen werden zwei Punkte auf dem Canvas angegeben, die Farbe verläuft dann vom ersten zum zweiten. Der radiale Farbverlauf wird ebenso mit Hilfe von zwei Punkten aber auch zwei Radien definiert. Damit ergeben sich zwei Kreise zwischen deren Perimetern die Farbe verläuft.  
Welche Farbwerte der Verlauf an welchen Stellen erreichen soll, wird mit Hilfe von Haltepunkten auf der Verlaufsstrecke angegeben. Die Lage der Haltepunkte wird dabei nur mit einem einzigen Wert relativ zur Gesamtstrecke bzw. -fläche des Verlaufs definiert. Dabei entspricht der Wert 0 der Position des Beginns und 1 der des Endes des Verlaufs. Der Haltepunkt mit dem kleinsten Wert bestimmt die Farbe bis zum Beginn und darüber hinaus, der mit dem größten die Farbe bis zum Ende und darüber hinaus. Dazwischen laufen die Farben ineinander.

let gradient: CanvasGradient = crc2.createLinearGradient(0, 0, 0, 100);

gradient.addColorStop(0, "black");

gradient.addColorStop(.5, "red");

gradient.addColorStop(1, "gold");

crc2.fillStyle = gradient;

crc2.fillRect(0, 0, 200, 100);

* Welches Bild liefert der oben angegebene Code?
* Füge weitere Haltepunkte ein um schärfere Kanten zwischen den Farben zu erhalten.

### Pattern

Es ist auch möglich, Füllmuster einzusetzen. Dabei werden verschiedene Quellen für das Muster unterstützt wie Bitmaps, SVG-Bilder oder auch Grafiken, die auf einem Canvas erzeugt werden. Folgender Code liefert nebenstehendes Bild.

Ein Bild, das Platz, Rost enthält.

Automatisch generierte Beschreibung

let pattern: CanvasRenderingContext = document.createElement('canvas').getContext('2d');

pattern.canvas.width = 40;

pattern.canvas.height = 20;

pattern.fillStyle = '#fec';

pattern.fillRect(0, 0, pattern.canvas.width, pattern.canvas.height);

pattern.moveTo(0, 10);

pattern.lineTo(10, 10);

pattern.lineTo(20, 0);

pattern.lineTo(30, 0);

pattern.lineTo(40, 10);

pattern.lineTo(30, 20);

pattern.lineTo(20, 20);

pattern.lineTo(10, 10);

pattern.stroke();

crc2.fillStyle = crc2.createPattern(pattern.canvas, 'repeat');

crc2.fillRect(0, 0, canvas.width, canvas.height);

## Linien

Natürlich lässt sich auch die Gestaltung der Linien, die mit der Anweisung stroke() gezeichnet werden, detailliert kontrollieren. Linien können durchgehend oder unterschiedlich gestrichelt werden, Farben, Gradienten und auch Muster erhalten, sowie die Linienenden und die Ausführung von Knicken in den Linien spezifisch gestalten sein.

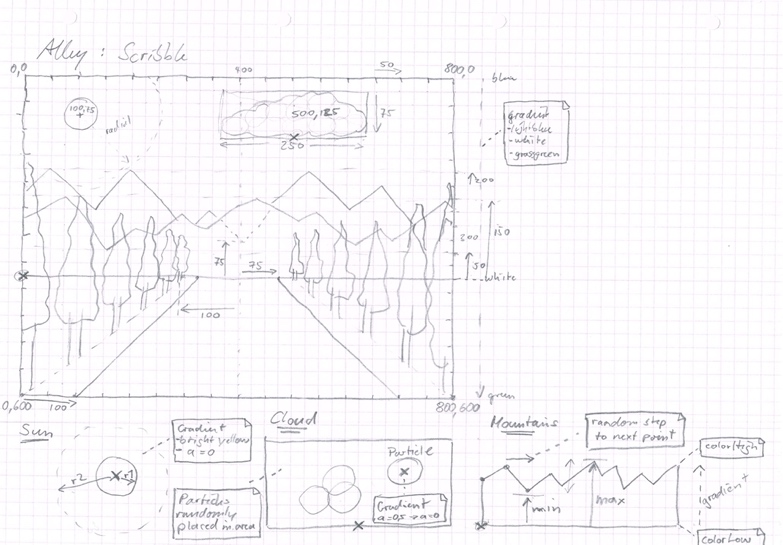
* Recherchiere im Internet nach den Möglichkeiten für die Liniengestaltung.

## Save/Restore

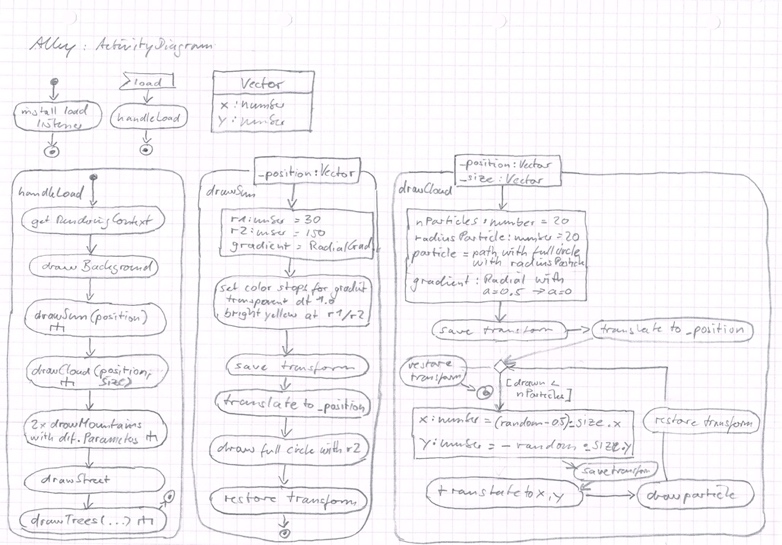
Die Anweisungen save() und restore() speichern nicht nur die aktuelle Transformationsmatrix zwischen, sondern auch die Stilangaben für die Füllungen und Linien und weitere Informationen, die insgesamt den Zustand (state) des RenderingContexts beschreiben. Somit ist es möglich, beispielsweise wieder zum vorangegangenen Stil zurück zu kehren, wenn zwischenzeitlich ein anderer benötigt wurde. save() speichert die Informationen auf einem Ablagestapel (stack), weitere Aufrufe von save() legen weitere solcher Informationen oben auf. Mit restore() wird der Stapel von oben nach unten Stück für Stück geleert und die Zustände des CanvasRenderingContexts in umgekehrter Reihenfolge der Speicherung wiederhergestellt.

# L08.2 Landschaftsbild

## Scribble

[](https://jirkadelloro.github.io/EIA2-Inverted/L08_Canvas/Material/Alley_Scribble.jpg)

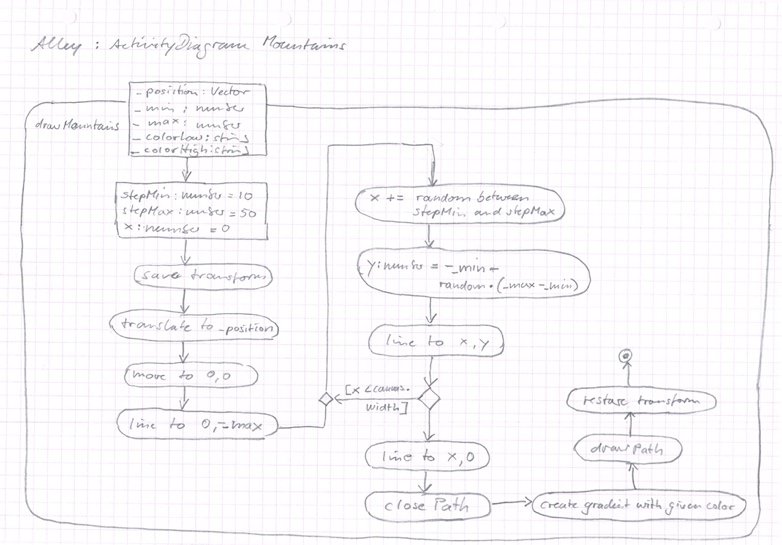
## Aktivitätsdiagramm

[](https://jirkadelloro.github.io/EIA2-Inverted/L08_Canvas/Material/Alley_ActivityDiagram.jpg)

## Implementation des Hintergrund

## Implementation der Wolke

## Aktivitätsdiagramm zu den Bergen

[](https://jirkadelloro.github.io/EIA2-Inverted/L08_Canvas/Material/Alley_ActivityDiagram-Mountains.jpg)

## Implementation der Berge

## Übungen

* Isoliere die Funktionalität eine Zufallszahl zwischen einem minimalen und maximalen Wert zu liefern. Konzipiere sie als Aktivität und implementiere sie als Funktion, welche die beiden Grenzwerte entgegenen nimmt und eine entsprechende Zufallszahl zurück gibt. Ersetze die Stellen im Programm, an denen diese Funktion genutzt werden kann und sollte durch entsprechende Aufrufe derselben.
* Studiere das Aktivitätsprogramm zum Aufbau der Baumreihe und die Implementation. Hier findest Du vielleicht noch einige wertvolle Hinweise für deine weitere Arbeit.
* Studiere den Code der Allee und ergründe, warum noch weitere Variablen vor den Aufrufen der draw-Functionen dazu gekommen sind.
* Experimentiere mit den Parametern der verschiedenen Funktionen und auch mit den hardkodierten Werten innerhalb dieser. Modifiziere so das Bild nach deinem ästhetischen Empfinden.
* Lasse die Position der Straße am Horizont um einen zufälligen Betrag zwischen -200 und 200 Pixeln horizontal verschieben, so dass der Fluchtpunkt bei jedem Aufruf des Programms ein anderer ist. Warum werden die Baumreihen mitbewegt?
* Der CanvasRenderingContext bietet noch weitere Möglichkeiten wie z.B. Schatten. Verschaffe dir einen Überblick über diese.

## L09 Classes

# L09.1 Grundlagen

Ein Bild, das Text, drinnen, Klavier, Computer enthält.

Automatisch generierte Beschreibung

Quelle: https://www.thepositiveencourager.global/alan-kay-on-creativity-3

Als Sohn eines Wissenschaftlers und einer Künstlerin hat Alan Kay nie zwischen Kunst und Wissenschaft unterschieden. Während er als Jazzgitarrist und Gitarrenlehrer Geld verdiente, erwarb er seinen Bachelorabschluss in Mathematik und Molekularbiologie, einen Masterabschluss in Elektrotechnik und den Ph.D. in Informatik. Er prägte maßgeblich den Begriff der objektorientierten Programmierung.

## Objektorientierung

Dieses Entwurfsparadigma für Software erlaubt eine intuitive Modellierung und Konzeption auch komplexer Softwareanwendungen.

Aus deiner Erfahrung weißt Du, dass sich die von dir wahrgenommene Realität aus einer Unzahl von materiellen (Autos, Häuser, Lebewesen, Möbel etc.) und immateriellen (Sprache, Gefühle, Konten, Identitäten etc.) Teilen zusammensetzt. Diese Umwelt vollständig zu beschreiben ist wohl unmöglich, die einzelnen Teile aber kannst Du auf unterschiedlichen Abstraktionsebenen schon besser erfassen. Dabei vereinfachst, klassifizierst und generalisierst Du und baust im Kopf ein Modell der Realität auf, was dir erlaubt Vorhersagen zu treffen, auch wenn Du nicht alle Details deiner aktuellen Umgebung kennst. Du weißt was ein Hund ist, obwohl es den Hund gar nicht gibt. Es gibt immer nur ganz spezifische Exemplare von Hund. Trotzdem gehst Du davon aus, dass ein dir völlig unbekannter Hund bellen kann. Du weißt, dass ein Hund in der Regel vier Beine hat, eine Schnauze und einen Schwanz, dass er neben bellen auch die Fähigkeit hat zu knurren und zu beißen und manchmal furchtbar zu stinken.

## Abstraktion

Ähnlich gehst Du beim objektorientierten Entwurf vor. Wenn Du ein System planst, überlegst Du, aus welchen Teilen es sich zusammensetzt, welche gleichartigen Teile es gibt, wie Du diese beschreiben kannst und wie die Teile mit ihrer Umwelt interagieren. Du abstrahierst und generalisierst und kannst für verschiedene Klassen von Teilen Eigenschaften identifizieren, die das einzelne, tatsächlich existierende Teil näher beschreiben. Bei der Klasse Hund könnte das beispielsweise die Größe oder die Fellfarbe sein. Diese Eigenschaften sind in gewissen Grenzen variabel, ohne dass das beschriebene Tier kein Hund mehr wäre. Es bleibt ein Hund, auch wenn es manchmal schwer zu glauben ist.

Ein Bild, das stehend, Kuh, Hund, Säugetier enthält.

Automatisch generierte Beschreibung  
<https://zaypa.com/wie-wahle-ich-deinen-hund-fur-dich-und-deine-familie.html>

***Abstraktion****ist das erste Prinzip der Objektorientierung!*

## Modellierung

Bislang hatten wir komplexere Datenstrukturen mit Interfaces irgendwo im Code beschrieben. Damit haben wir schon Eigenschaften von Objekten angegeben. Beispielsweise besitzen alle Vektoren aus dem vorangegangenen Kapitel die Eigenschaften x und y, bei jedem einzelnen tatsächlich erschaffenen Vektor sind sie mit individuellen Werten definiert. Was mit diesen Vektoren aber geschieht, ist an ganz anderen Stellen im Code implementiert. So werden Vektoren addiert oder skaliert und der Code dafür muss jedes mal neu geschrieben werden.

***FunFact:****Schau dir einmal im kompilierten Javascript-Code an, was von Interfaces übrig bleibt… sie sind komplett verschwunden. Interfaces sind lediglich ein Anweisung für den Compiler mit der Du angibst, welche Datenstruktur Du willst, damit er dir helfen kann, dich daran zu halten.*

Bei der objektorientierten Modellierung definierst Du nicht nur die Eigenschaften, sondern auch die Aktivitäten, die ein Objekt der beschriebenenen Klasse ausführen können soll. So wie der Hund beißen, bellen, knurren und stinken kann, kann der Vektor vielleicht seine Länge ändern, oder durch Addition mit einem anderen Vektor zu einem resultierenden Vektor werden und so weiter. Im Diagramm schreibst Du hierzu einfach unter dem Feld für den Namen des Typs und dem Feld für die Eigenschaften nun ein drittes Feld, in dem Du die Aktivitäten auflistest.

Bislang hattest Du Interfaces nur als Randnotiz mit in die Aktivitätsdiagramme eingetragen. Nun lernst Du einen neuen Diagrammtyp kennen: **das Klassendiagramm**. Hier wird dargestellt, wie die Klassen aufgebaut sind und miteinander in Beziehung stehen.

* Wo hast Du so etwas bloß schon mehrmals gesehen?

## Implementation

Die im Diagramm dargestellte Vektorklasse kann folgendermaßen implementiert werden

class Vector {

x: number;

y: number;

scale(\_factor: number): void {

this.x \*= \_factor;

this.y \*= \_factor;

}

add(\_addend: Vector): void {

this.x += \_addend.x;

this.y += \_addend.y;

}

}

Vector ist jetzt nicht mehr nur eine Objektstruktur, die sich mit einem Interface beschreiben lässt, Vector ist jetzt eine Klasse. So wird nun statt interface das reservierte Wort class benutzt. Die Aktivitäten werden wie gewohnt als Funktionen implementiert, aber das Schlüsselwort function wird nicht mehr angegeben.

***Hinweis:****Funktionen innerhalb von Klassen werden “Methoden” genannt. Objekte der Klasse verfügen nun über Methoden mit denen sie sich und ihre Umwelt beeinflussen können.*

## Kapselung

Das Programm, das einen solchen Vektor nutzt, muss nun nicht mehr wissen, wie man einen Vektor skaliert oder einen zweiten hinzuaddiert. Es genügt zu wissen, dass man es tun kann und welche Parameter der Vektor hierzu braucht. Den Rest macht der Vektor selbst. Damit wird der Vektor zu einer Black-Box, deren interne Arbeitsweise nicht bekannt sein muss, um sie zu verwenden. Der Vektor ist “gekapselt”.

***Kapselung****ist das zweite Prinzip der Objektorientierung!*

## Instanzierung

Der obenstehende Code beschreibt die Klasse Vektor, es existiert aber noch kein Exemplar eines Objektes dieser Klasse. Die Erzeugung eines solchen erfolgt, ganz anders als bei der Arbeit mit Interfaces, mit der Anweisung new. Diesen Vorgang nennt man auch Instanzierung, denn es wird eine Instanz der Klasse geschaffen.

* Aus welchen Zusammenhängen kennst Du new bereits?
* Implementiere die Vektorklasse in einem neuen namespace und füge darunter die untenstehenden Zeilen an.

let v1: Vector = new Vector();

v1.scale(2);

console.log(v1);

* Teste das Programm im Browser oder mit Node. Welche Konsolenausgabe erhältst Du?
* Verfolge das Programm mit dem Debugger bis hinein in die Methode scale der Klasse. Was hat es mit dem Schlüsselwort this auf sich?
* Zu Beginn der Klasse werden die Eigenschaften x und y deklariert. Ergänze den Code an dieser Stelle so, dass sie auch gleich mit dem Wert 0 definiert werden. Was ändert sich beim nächsten Test und Debuggen?
* Spendiere der Klasse eine neue Methode set(\_x: number, \_y: number) welche den Eigenschaften eines Objektes der Klasse die Werte von \_x bzw. \_y zuweist. Lasse v1 vor dem Aufruf von scale diese Methode ausführen mit Werten die Du dir ausdenkst. Was geschieht jetzt?

## Klasse vs. Objekt

Häufig werden die Begriffe Klasse und Objekt verwechselt oder unscharf gebraucht. Das führt zu Verwirrung und Fehlern. Auch wenn der Unterschied in diesem Text bereits mehrfach herausgestellt wurde, soll an dieser Stelle noch einmal explizit darauf eingegangen werden. Eine Klasse beschreibt die Struktur, die ein Objekt derselben aufweisen soll, und die Methoden, die mit einem solchen Objekt verknüpft werden. Sie ist vergleichbar mit dem Hunde-Genom oder mit dem Hausbauplan eines Architekten. Ist die Klasse definiert, können beliebig viele Objekte der Klasse erzeugt werden, so wie ein ganzer Hundewurf, eine Siedlung oder ein ganzes Array voller Vektoren. Jedes dieser Objekte besitzt die in der Klasse angegebenen Eigenschaften, gegebenenfalls mit individuellen Werten, wie Fellfarbe, Dachziegeltyp oder Koordinate.

Ein Objekt einer Klasse kann zudem die in der Klasse definierten Methoden nutzen, ohne dass eine Referenz auf das Objekt der Methode als Parameter übergeben werden muss. Stattdessen wird das Objekt innerhalb der Methode automatisch mit dem reservierten Wort this referenziert. Der Aufruf der Methode erfolgt mit Hilfe der Punkt-Syntax object.method(...). Da das Objekt eine Instanz einer bestimmten Klasse ist, wird damit die Methode eindeutig identifiziert, auch wenn andere Klassen über eine Methode gleichen Namens verfügen sollten.

## Constructor

Die Hausbau-Metapher führt dich gleich zu einem weiteren interessanten Aspekt. Der Architekt, also Du, macht zwar den Plan, gebaut wird ein Haus aber von einem Bauunternehmer. Im Englischen ist das der Constructor.

Auch bei der Instanzierung von Objekten kommt ein Constructor zum Einsatz, den Du im Deutschen wiederum Konstruktor nennen kannst. Er ist eine besondere Methode, die automatisch aufgerufen wird, wenn die Anweisung new ausgeführt wird und kümmert sich darum, dass das Objekt ordentlich gebaut wird. Ein Standardkonstruktor ist in Javascript immer dabei und tritt nicht in Erscheinung. Du kannst aber für jede Klasse einen eigenen Konstruktor definieren.

Ein Konstruktor ist dann von Bedeutung, wenn weitere Informationen zum Bau des Objektes einfließen sollen oder währenddessen zwingend noch weitere Aktivitäten ausgeführt werden müssen. Bei einem Haus sollte der Constructor vielleicht wissen, welche Ziegel er nun tatsächlich auflegen soll, da diese Eigenschaft des Hauses variabel ist. Wird ein Hund konstruiert, soll vielleicht gleich die Gemeinde informiert werden, damit sie die Hundesteuer eintreiben kann.

Bei der Vektorklasse erscheint es sinnvoll, bei der Konstruktion eines Vektors gleich die Komponenten mitzugeben, so dass die Eigenschaften x und y des Vektors sofort die gewünschten Werte erhalten.

class Vector {

x: number;

y: number;

constructor(\_x: number, \_y: number) {

this.set(\_x, \_y);

}

set(\_x: number, \_y: number): void {

this.x = \_x;

this.y = \_y;

}

...

}

Der Konstruktor verlangt nun zur Instanzierung eines Vektors zwingend die Übergabe zweier Werte. TypeScript wird einen Fehler melden, wenn Du versuchst einen Vektor wie zuvor nur mit new Vector(); zu instanzieren. new Vector(10, -3) dagegen wird akzeptiert und der Vektor wird mit den gegebenen Werten angelegt.

***Hinweis:****Die set-Methode ist übrigens die, welche Du weiter oben schon hättest implementieren sollen. Falls es nicht geklappt hat, kannst Du hier spicken…*

## Konventionen

Im Umgang mit Klassen und Objekten solltest Du dich unbedingt an einige Konventionen halten:

* Jede Klasse wird in einer eigenen Datei implementiert
* Der Dateiname entspricht dem Namen der Klasse (z.B. Vector findet sich in Vector.ts)
* Die Namen der Methoden wiederholen nicht den Klassennamen (scale statt scaleVector)
* Der Klassenname beschreibt das Wesen genau einer Instanz der Klasse, nicht mehrerer (Vector statt Vectors)

# L09.2 Asteroids

Mit diesen Basisregeln des objekorientierte Entwurfs kann man sich schon daran machen, eine komplexe interaktive Anwendung zu konzipieren und zu implementieren. Als Beispiel für diese und die folgenden Lektionen wird das Videospiel “Asteroids” (Atari 1979) in seiner ursprünglichen Arcade-Version herangezogen.

***FunFact:****Asteroids ist einer der größten Erfolge in der Geschichte der Videogames und Ataris meistverkaufter Münzautomat. Es gab allerdings eine dominante Strategie, die es geübten Spielern erlaubte nach Einwurf nur einer Münze stundenlang zu spielen und dabei so viele Bonus-Schiffe anzuhäufen, dass der Automat, dann vom Spieler allein gelassen, noch lange Zeit brauchte um alle Schiffe zu zerstören…*

* Falls dir Asteroids nicht bekannt ist, schau dir den entsprechenden Wikipedia-Artikel und dieses Youtube-Video an: https://www.youtube.com/watch?v=82mbTdpPM58 (ab 0:18)

## Anwendungsfalldiagramm

* Wenn dir das Anwendungsfalldiagramm nicht gleich einleuchtet, schau dir den Wikipedia-Artikel und das Video genau an und versuche, die einzelnen Anwendungsfälle dem Geschehen zuzuordnen.
* Beachte, dass im Diagramm viele Anwendungsfälle aufgeführt sind, die das System selbständig bewältigen muss. Was vom Player gesteuert ist, ist auch mit ihm verbunden.

## UI-Scribble

Um das Spiel interessanter zu machen, insbesondere für die Übungszwecke in diesem Kurs, wird eine etwas andere Verteidigungsmechanik für das Raumschiff konzipiert. Statt einzelner kleiner Projektile, wie die Ufos sie verschießen, werden Laserstrahlen eingesetzt, die in ihrem Kreuzungspunkt ein vernichtendes Kraftfeld erzeugen. Somit kann die Maus als Steuerung besser genutzt und die zusätzliche Herausforderung eines Energiemanagements für den Spieler geschaffen werden.

## Klassendiagramm

Bei der objektorientierte Modellierung kommt ein weiterer Diagrammtyp hinzu, den Du nutzen solltest, bevor Du mit den detaillierten Aktivitätsdiagrammen beginnst. Dabei identifizierst Du zunächst die verschiedenen Objekttypen, die in deiner Anwendung eine Rolle spielen. Sofern es sich dabei um Objekte handelt, die eine sichtbare Repräsentation haben, kannst Du sie im UI-Scribble identifizieren.

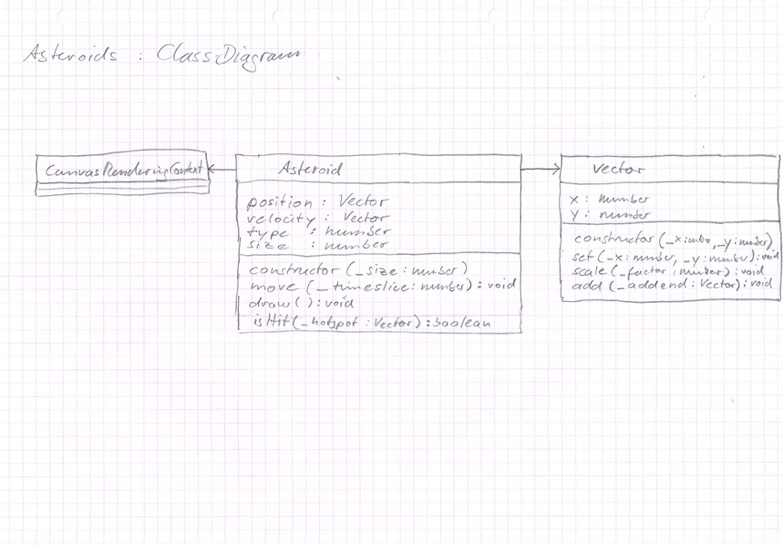
### Fünf Fragen

Für jeden Objekttyp stellst Du dir dann die folgenden Fragen, wobei “es” ein beispielhaftes Objekt des Typs meint.

1. was hat es?
2. was kann es?
3. was weiß es?
4. wer hält es?
5. was ist es?

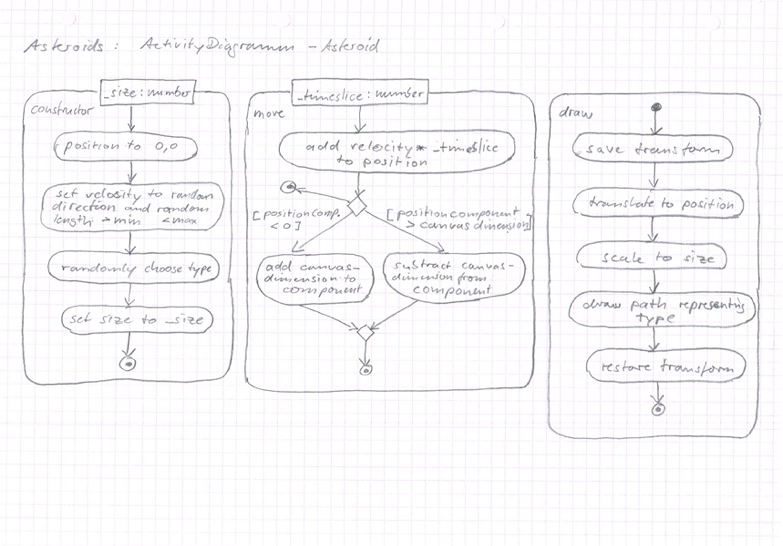
Die ersten beiden Fragen führen dich direkt zur Darstellung der einzelnen Klassen im Diagramm. Kümmern wir uns zunächst um die Asteroiden…

**Video: Definition der Asteroiden als Klassendiagramm**

[](https://jirkadelloro.github.io/EIA2-Inverted/L09_Classes/Material/Asteroids_ClassDiagram.jpg)

## Aktivitätsdiagramm

Jetzt kannst Du die einzelnen Methoden mit Hilfe von Aktivitätsdiagrammen konzipieren.

[](https://jirkadelloro.github.io/EIA2-Inverted/L09_Classes/Material/Asteroids_ActivityDiagram-Asteroid.jpg)

## Implementation

## Zeitsignale

Die Animation mit Hilfe einer Schleife zu bewerkstelligen ist nicht einfach machbar, da der Browser dann nur mit der Schleife beschäftigt ist und keine Zeit mehr hat, das Bild anzuzeigen. Stattdessen soll er das Bild anzeigen und dann warten, bis einige Zeit vergangen ist um das nächste Bild zu zeichnen und anzuzeigen und so weiter.  
Du weißt bereits, dass EventListeners erhalten und aktiv bleiben, auch wenn das Hauptprogramm bereits beendet ist. Das gilt auch für Zeitsignale! Im Aktivitätsdiagramm ist für diese ein eigenes Symbol als Startknoten definiert, das an eine Sanduhr erinnert. Damit kannst Du nun leicht einen “Herzschlag” für deine Anwendung definieren.

In Javascript/TypeScript stehen dir folgende Optionen zur Realisierung solcher Zeitsignale zur Verfügung.

### window.setTimeout (handler, time, args…)

Die Anweisung bewirkt, dass die handler-Funktion nach Ablauf der mit time in Millisekunden angegebenen Zeit aufgerufen wird, gemessen ab dem Zeitpunkt des Anweisungsaufruf. Weitere Parameter (args...) können der Funktion dabei übergeben werden.

### window.setInterval (handler, time, args…)

Die Anweisung bewirkt, dass die handler-Funktion in dem mit time angegebenen zeitlichen Abstand periodisch aufgerufen wird. Weitere Parameter (args...) können der Funktion dabei übergeben werden.

### window.requestAnimationFrame(handler)

Die Anweisung bewirkt, dass die handler-Funktion periodisch in einem für die grafische Aufbereitung sinnvoll erscheinendem Zeitintervall aufgerufen wird. Über dieses entscheidet der Browser, Chrome versucht eine Bildwiederholrate von 60 Bildern pro Sekunde zu erreichen (Frames per second, fps).

## Hauptprogramm

[Ein Bild, das Text, Whiteboard enthält.

Automatisch generierte Beschreibung](https://jirkadelloro.github.io/EIA2-Inverted/L09_Classes/Material/Asteroids_ActivityDiagram-Main.jpg)

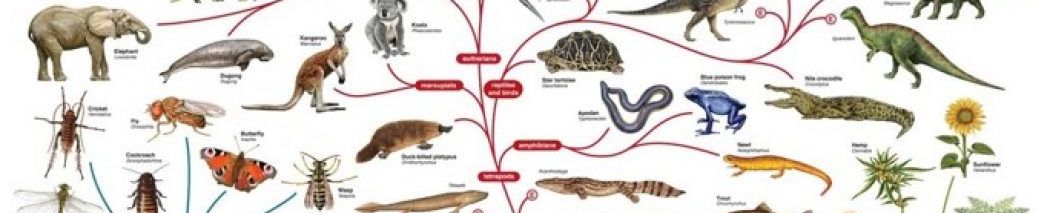
***Hinweis:****Nicht alles lässt sich von vorneherein festlegen, manches musst Du ausprobieren, um ein Gefühl dafür zu bekommen, wie ein gutes Nutzungserlebnis erreicht werden kann. In der Konzeption musst Du abschätzen, welche Tests erforderlich sind und wie Du diese ermöglichst. Du musst zudem selbst Prototypen entwickeln können, um kleinere Sachen alleine zu implementieren und auszuprobieren ohne dafür einen Stab von Leuten zu beschäftigen!*

Ein Problem tritt auf bei der Übergabe der Position an den optionalen Parameter \_position im Constructor der Asteroid-Klasse. Da Vector ein komplexer Typ ist, hält eine Variable dieses Typs lediglich eine Referenz auf ein Vektorobjekt (siehe [Werte vs. Referenzen](https://jirkadelloro.github.io/EIA2-Inverted/L01_Recap%26Foundation#werte-vs-referenzen)). Durch die Zuweisung des Parameterinhaltes an das Objektattribut wurde also kein neues Vektorobjekt erzeugt, sondern nur die Referenz erneut genutzt. Die Bruchstücke referenzieren also das gleiche Vektorobjekt und haben daher die gleiche Position.

* Löse das Problem, in dem Du nicht die Referenz zuweist, sondern mit den Informationen des Parameters ein neues Positionsobjekt instanzierst und dieses zuweist.
* Erweitere die Vektorklasse um eine Methode copy(), welche ein neues Vektorobjekt zurückliefert, das die gleichen Attributwerte hat wie der Vektor der sie ausführt.
* Nutze diese Methode um das obige Problem noch eleganter zu lösen.
* Skaliere die Linienstärke in der draw-Methode der Asteroid-Klasse entsprechend des size-Attributs umgekehrt proportional hoch.

## L10 Inheritance

# L10.1 Vererbung



## Generalisierung vs Spezialisierung

1987 stellte Bararbara Liskov auf einer Konferenz einen Kerngedanken der objektorientierten Modellierung vor. Danach soll ein Objekt einer Klasse durch ein Objekt einer Sub-Klasse dieser Klasse ersetzt werden können, ohne dass das Programm, welches das Objekt verwendet, verändert werden muss. 1993 formalisierte sie zusammen mit Jeannette Wing was heute als das **Liskov’sche Substitutionsprinzip** bekannt ist.

Bezogen auf Hunde kann das Liskov’sche Substitutionsprinzip folgendermaßen beispielhaft dargestellt werden:  
Wenn Hunde grundsätzlich bellen können, dann können auch Subklassen von Hund, wie Dogge, Chihuahua und Pudel bellen. Wenn Du also an einem Haus klingelst, an dem ein Schild “Warnung vor dem Hund” steht, darfst Du davon ausgehen, dass ein Hund bellen wird. Es ist nicht erforderlich, dass auf dem Schild “Warnung vor dem Pudel” steht, auch wenn der Hund in dem Haus ein Pudel sein sollte.

Pudel ist eine Spezialisierung von Hund, Hund ist die Generalisierung von Pudel.

Pudel gelten als besonders gelehrig und werden daher gerne für besondere Tricks und im Zirkus eingesetzt. Dabei ist es wiederum unerheblich, ob es sich um Toy-, Zwerg-, oder Großpudel handelt. Diese sind Spezialisierungen von Pudel, die sich in der Varianz der Größe unterscheiden.

Der Hund kann weiter generalisiert werden, denn er gehört zu der Familie der Hundeartigen, zu denen beispielsweise auch Bären, Robben und Stinktiere gehören. Zusammen mit den Katzenartigen stellen die Hundeartigen die Raubtiere die wiederum zu der Klasse der Säugetiere gehören, von denen auch der Mensch eine Spezialisierung darstellt.

## Darstellung im Klassendiagramm

In der Biologie teilt man ein in Klassen, Rassen, Arten, Gattungen, Familien, Ordnungen und so weiter. Bei der objektorientierten Modellierung muss man lediglich Klassen unterscheiden in Super- und Subklassen.

Die Superklasse stellt die Generalisierung einer Subklasse dar, die Subklasse eine Spezialisierung der Superklasse.

Im Klassendiagramm verweisen Pfeile mit einer dreieckigen und leeren Spitze von der Subklasse auf ihre Superklasse und stellen so diese Beziehung dar. Der oben angeführte Exkurs in die Biologie, der übrigens keiner wissenschaftlichen Überprüfung standhält, könnte im Klassendiagramm folgendermaßen aussehen.

* Studiere das Diagramm und mache dir die Beziehung zwischen Sub- und Superklasse klar.
* Beschreibe nur anhand des Diagramms die Eigenschaften und Fähigkeiten eines Zwergpudels.
* Sollten die verschiedenen Pudel eigene Subklassen bilden? Was wäre eine Alternative?
* Wie hilft dir ein solches Diagramm und die Betrachtung von Generalisierung und Spezialisierung im Umgang mit Lebewesen? Beschreibe es an diesem Beispiel.

## Vererbung

In diesem Zusammenhang spricht man auch von Vererbung (Inheritance). Eine Superklasse vererbt alle ihre Eigenschaften und Methoden an die Subklasse, so wie es bei obigem Stammbaum der Arten gezeigt ist.

***Vererbung****ist das dritte Prinzip der Objektorientierung!*

Wegen der Vererbungsbeziehung werden Superklassen manchmal auch als Elternklassen und Subklassen als Kindklassen bezeichnet. Das führt aber zur Verwirrung, da die manche Laufzeitbeziehungen zwischen Objekten, beispielsweise im DOM, bereits als Parent-Child-Beziehung bezeichnet wird. Weitere Begriffe sind Ober- und Unterklasse, Basis- und Abgeleitete Klasse und weitere. In dieser Lektion werden weiter strikt die Bezeichnungen Super- und Subklasse genutzt.

## Regeln für die Modellierung

Ein Dompteur im Zirkus muss nicht nur die Eigenheiten jeder Hunderasse in seiner Truppe kennen, sondern sogar die jeder einzelnen Instanz. Dem Schalterbeamten “in Zollhafe von modernes Planet” genügte dagegen zu wissen, ob eine Lebensform ein Säuger ist, um das richtige Zimmer zuzuweisen ([siehe Ijon Tichy, Folge 2, 4:23 - 5:33](https://www.youtube.com/watch?v=_DSCXsIR03Q))

Bei der Modellierung eines Systems solltest Du diese zwei Regeln beachten:

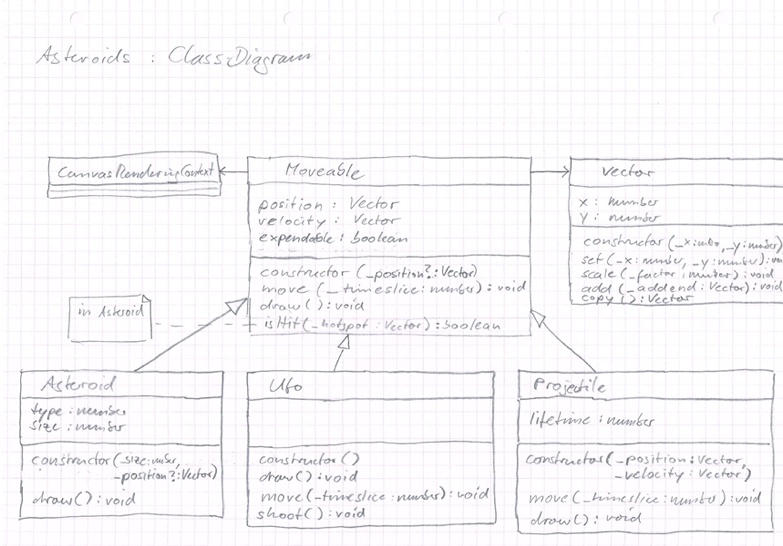
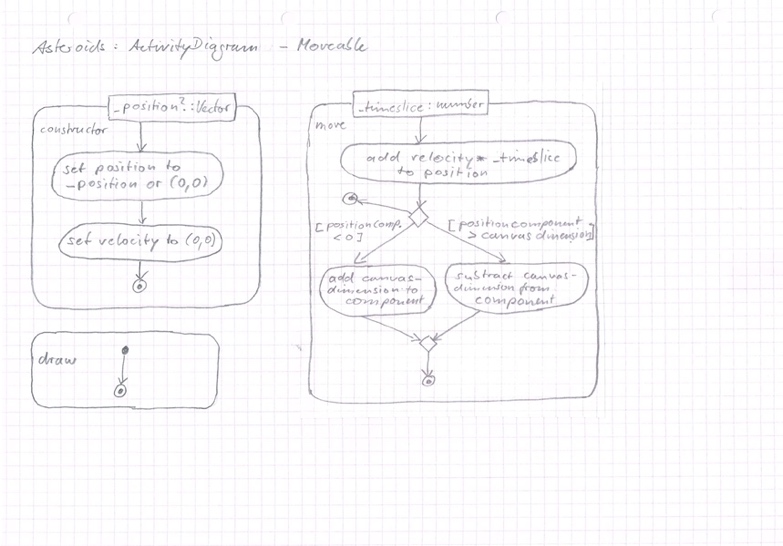
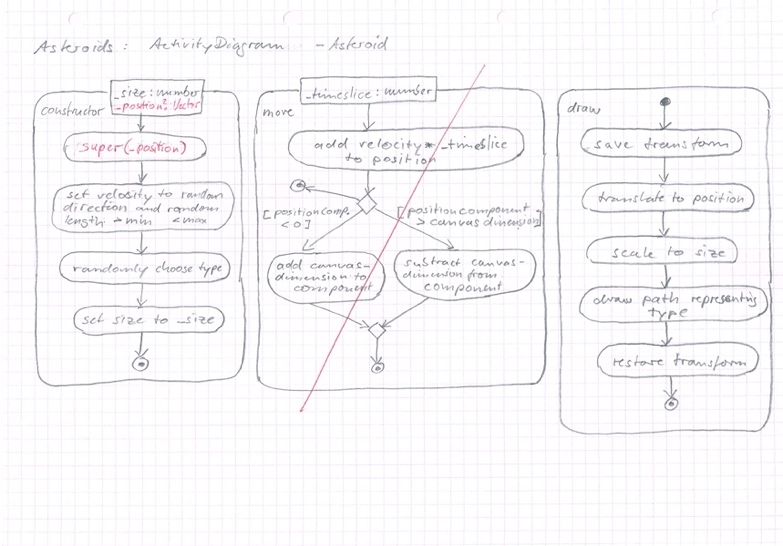
* jeder Teil darf nur so schlau wie nötig sein
* jeder Teil muss so dumm wie möglich sein

Jeder Teil soll also nur das von seiner Umwelt und anderen Teilen wissen, was unbedingt erforderlich für die eigene Funktion ist!

# L10.2 Ufos

Unter Berücksichtigung der oben genannten Prinzipien wird Asteroids weiter entwickelt.

## Erweiterung des Klassendiagramms

[](https://jirkadelloro.github.io/EIA2-Inverted/L10_Inheritance/Material/Asteroids_ClassDiagram.jpg) [](https://jirkadelloro.github.io/EIA2-Inverted/L10_Inheritance/Material/Asteroids_ActivityDiagram-Moveable.jpg) [](https://jirkadelloro.github.io/EIA2-Inverted/L10_Inheritance/Material/Asteroids_ActivityDiagram-Asteroid.jpg)

## Implementation Moveable, Refaktorierung Asteroid

***extends****erweitert eine Superklasse um eine Subklasse!****super(...)****ruft aus der Subklasse den Constructor der Superklasse auf!****super.methode(...)****ruft aus der Subklasse die methode der Superklasse auf!*

## Polymorphie

Griechisch πολύς (polýs) = „viel“ und μορφή (morphé) = “Form, Gestalt”. Hunde sind vielgestaltig, das wurde in diesen Lektionen bereits ausgiebig behandelt. Die einzelnen Hunderassen können aber nicht nur anders aussehen, sondern sich vielleicht auch unterschiedlich verhalten. Ein Jagdhund wird beim Anblick einer Herde Schafe vielleicht gerne hineinrennen und eines reißen, ein Hütehund eher um die Herde herumlaufen und sie beisammen halten. Für jemanden, der mit diesen Hunden spazierengeht, so wie die angeblich 200.000 professionellen Gassigeher in den USA, ist dabei vordringlich wichtig, genug Leinen und Halsbänder zu haben, um die Hunde bei sich halten zu können. Die Rasse ist unerheblich.

***Polymorphie****ist das vierte Prinzip der Objektorientierung!*

Im Asteroids-Programm sind derzeit zwei Klassen implementiert, die von Moveable erben, also zwei Subklassen der Superklasse Moveable. Anders ausgedrückt hat Moveable derzeit drei Gestalten: Moveable, Asteroid und Projectile. Sie unterscheiden sich in Aussehen und Verhalten

| **Klasse** | **Aussehen** | **Bewegung** |
| --- | --- | --- |
| Moveable | keines | Geradlinig mit Umlauf am Rand |
| Asteroid | Felsbrocken | wie Moveable |
| Projectile | kleiner Punkt | wie Moveable aber Reichweite begrenzt |

Damit sich die Instanzen der Klassen bewegen und darstellen können, ruft das Hauptprogramm deren move- und draw-Methoden auf. Dabei ist es dem Hauptprogramm aber völlig egal, ob es sich dabei um Asteroiden oder Projektile handelt, solange sie Moveables sind und somit diese Methoden entweder geerbt haben oder sie neu implementieren (überschreiben). Das Hauptprogramm kann mit allen Gassi gehen …

Nach dem Liskov’schen Substitutionsprinzip muss Folgendes möglich sein, auch wenn Moveable vielgestaltig ist:

let m: Moveable = new Moveable();

let a: Moveable = new Asteroid(1);

let p: Moveable = new Projectile(new Vector(0, 0), new Vector(0, 0));

Beachte, dass die Variablen m, a und p alle mit dem Typ Moveable deklariert wurden, sie aber unterschiedliche Objekt-Typen referenzieren.

* Implementiere diese drei Zeilen und lasse dir die Variablen in der Konsole ausgeben oder schaue sie dir im Debugger an.

Wenn Vorangegangenes möglich ist, dann muss auch Folgendes möglich sein:

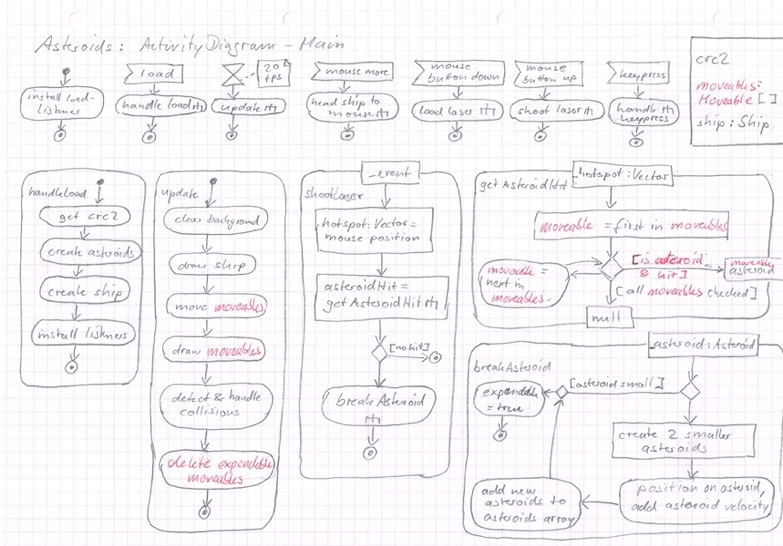
let moveables: Moveable[] = [];

moveables.push(new Asteroid(1));

moveables.push(new Projectile(new Vector(0, 0), new Vector(0, 0)));

* Überprüfe auch dies mit einer entsprechenden Implementation

## Aktivitätsdiagramm Hauptprogramm

[](https://jirkadelloro.github.io/EIA2-Inverted/L10_Inheritance/Material/Asteroids_ActivityDiagram-Main.jpg)

## Implementation bei Nutzung der Polymorphie

***a instanceof B****prüft, ob das Objekt a eine Instanz der Klasse B ist.*

* Konzipiere die Ufos! Dabei musst Du unter anderem folgende Designentscheidung treffen:
  + wer soll die Projektile erzeugen? Wird diese Aufgabe von den Ufos übernommen oder vom Hauptprogramm?
    - wenn Ufo: wie kommen die Projektile in das Moveable-Array um verwaltet zu werden?
    - wenn Hauptprogramm: wie erkennt das Hauptprogramm, dass das Ufo ein Projektil abfeuert?

***Hinweis:****Der Pfad zum Zeichnen der Ufos wird bereits von der Funktion createUfoPath() erzeugt.*

## L11 Advanced

# L11.1 Modifikatoren & Accessoren

Ein Bild, das drinnen, Tisch enthält.

Automatisch generierte Beschreibung

**Einschub**: Eine Lösung für die letzte Übung von Lektion 10 mit CustomEvents, die bereits in [Lektion03](https://jirkadelloro.github.io/EIA2-Inverted/L02_Events/#customevents) kurz angerissen wurden.

Die objektorientierte Modellierung bietet noch einige zusätzliche Möglichkeiten, welche die Entwicklung und Wartung von Softwaresystemen vereinfachen und verbessern. Komplexere Software wird meist im Team entwickelt, verschiedene Teile, Module und Klassen werden von unterschiedlichen Leuten entworfen, implementiert und genutzt. Es ist ungemein hilfreich und praktisch, wenn diese Mechanismen genutzt werden, um Missverständnisse zwischen den Teammitgliedern zu vermeiden oder Fehler bei der Programmierung von vorneherein auszuschließen.

* Entwirf für diese Lektion ein kleines Programm bestehend aus einer Superklasse, einer Subklasse davon, einem Hauptprogramm und einer HTML-Datei, welche deine Skripte lädt. Die Funktion ist zunächst irrelevant, es geht nur um Struktur.

## Abstract

### Abstrakte Klasse

Von der Klasse Moveable bei Asteroids werden keine Objekte instanziert, nur von den Subklassen. Moveable ist somit eine völlig abstrakte Klasse, sie stellt wie “Hund” lediglich ein Konzept, eine Idee, dar (vergleiche Platons Ideenlehre). Mit dem reservierten Wort abstract kann eine Klasse als solche gekennzeichnet werden. Der Versuch, ein Objekt dieser Klasse zu instanzieren, wird nun bereits vom TypeScript-Compiler geahndet.

* Markiere deine Superklasse als abstrakt, in dem Du einfach abstract vor class schreibst. Versuche im Hauptprogramm ein Objekt der Klasse erzeugen zu lassen und beachte die Fehlermeldungen.

Objekte der Subklassen können dennoch weiter die Methoden der abstrakten Superklasse nutzen.

* Überprüfe diese Aussage, indem Du in deiner Superklasse eine Methode implementierst und diese aus einem Objekt der Subklasse heraus aufrufen lässt.

### Abstrakte Methode

Es gibt aber auch Methoden, die in der abstrakten Superklasse nicht sinnvoll implementiert werden können, sondern nur in den Subklassen. draw() in Moveable ist hierfür ein gutes Beispiel. Dennoch muss die Superklasse Moveable die Methode draw aufweisen, damit das Hauptprogramm alle Moveables einfach verwalten und deren draw-Methoden aufrufen kann.

Solche Methoden abstrakter Klassen sollten ebenfalls mit abstract markiert werden. Dann muss dort kein Rumpf implementiert werden und es wird erzwungen, dass die Methode in der Subklasse definiert werden muss.

* Markiere eine neue Methode deiner abstrakten Superklasse als abstrakt. Erhältst Du eine Fehlermeldung, wenn Du diese Methode in der Subklasse nicht definierst?

***Hinweis****: Du kannst die Abstraktion schon beim Entwurf im Klassendiagramm darstellen, hierzu werden die entsprechenden Klassen- und Methodennamen lediglich kursiv geschrieben.*

## Static

In die entgegengesetzte Richtung zielt das Prinzip der statischen Methoden und Eigenschaften von Klassen. Diese können nämlich genutzt werden, ohne dass eine Instanz der Klasse geschaffen werden muss. Die Klasse, oder ein Teil davon, wird also sehr konkret. Hierfür genügt es, das reservierte Wort static der Deklaration einer Methode oder Eigenschaft voran zu stellen.

Bei Asteroids könnten statische Methoden in der Vektorklasse hilfreich sein. Folgende Zeile aus isHit beispielsweise ist recht unschön

let difference: Vector = new Vector(\_hotspot.x - this.position.x, \_hotspot.y - this.position.y);

Es wäre besser lesbar, wenn die Vektorklasse die Differenz zweier Vektoren berechnen und den resultierenden Vektor liefern könnte. Das ist einfach in der Vektorklasse zu implementieren

static getDifference(\_v0: Vector, \_v1: Vector): Vector

let vector: Vector = new Vector(\_v0.x - \_v1.x, \_v0.y - \_v1.y);

return vector

}

Nun steckt die Komplexität der Erzeugung eines neuen Vektors und der komponentenweisen Subtraktion in der Vektorklasse und die neue Klassenmethode getDifference kann einfach und intuitiv verwendet werden.

let difference: Vector = Vector.getDifference(\_hotspot, this.position);

* Definiere in deiner Superklasse eine statische Eigenschaft und eine statische Methode. Lasse das Hauptprogramm die Eigenschaft in der Konsole ausgeben und die Methode aufrufen. Nutze statt einer Objektreferenz nun einfach den Klassennamen hierfür.

***Hinweis****: Im Klassendiagramm werden statische Methoden oder Eigenschaften durch Unterstreichung gekennzeichnet!*

## Gültigkeit

Variablen, die innerhalb eines mit geschweiften Klammern abgegrenzten Code-Block mit let deklariert werden, sind nur innerhalb des entsprechenden Blocks gültig. Dabei können sie andere Variablen gleichen Namens, die ausserhalb des Blocks gültig sind “verdecken”. Dies wird in folgendem simplen Beispiel sichtbar.

let x: string = "I'm valid outside";

{

let x: string = "I'm valid inside";

console.log(x);

}

console.log(x);

Erzeugt wird die Ausgabe

I'm valid inside

I'm valid outside

Es ist erkennbar, dass für die Dauer der Ausführung des Blocks zwei Variablen namens x existierten, denn der Inhalt der “äußeren” wurde nicht verändert. Die “innere” verlor dagegen nach Beendigung des Blocks ihre Gültigkeit. Die verließ ihren “Scope” und der von ihr belegte Speicher wird wieder freigegeben. Die “äußere” aber existiert weiterhin und ihr Inhalt wird ausgegeben.

Solche Blöcke können Schleifenkörper oder if-Blocks abgrenzen, Funktions- oder Methodenrümpfe, Klassen, Objekte und Namespaces. Wie in obigem Beispiel ist es auch möglich, obgleich selten, einen Block zu definieren, ausschließlich um einen Gültigkeitsbereich abzugrenzen.

* Untersuche den Code “Scopes” im Anhang zu dieser Lektion und bringe ihn entweder in der Konsole oder im Browser zum Laufen. In diesem Code wird zehnmal die Variable x als string deklariert und mit unterschiedlichen Zeichenketten definiert. Die Werte bezeichnen den Gültigkeitsbereich.
* Erkläre die Ausgaben und die beiden Abweichungen.

## Sichtbarkeit

Es existieren also mehrere Gültigkeitsbereiche nebeneinander und ineinander verschachtelt. Insbesondere dann, wenn aus einem Bereich in einen anderen zugegriffen werden soll, so wie das Asteroid-Hauptprogramm beispielsweise auf die Eigenschaften oder Methoden der Moveables zugreift, kann schnell Verwirrung entstehen. Denn manche Informationen bieten die Klassen und Objekte tatsächlich für den Zugriff “von Außen” als Schnittstelle an, anderes dagegen brauchen sie lediglich für die interne Funktion. Hier kann ein Zugriff zu Fehlern führen

Bei der objektorientierten Modellierung können durch Sichtbarkeitsmodifikatoren Eigenschaften und Methoden gegen den Zugriff geschützt werden. Das erhöht nicht nur die Sicherheit, sondern macht es bei der Entwicklung im Team für alle sehr viel leichter, die richtigen Zugriffe zu finden und auszuwählen. Diese Modifikatoren sind lediglich reservierte Worte, die bei der Deklaration der Methoden und Eigenschaften voran gestellt werden.

### public

Der Zugriff ist völlig öffentlich. Dies ist auch die Standardsichtbarkeit, wenn kein Modifikator angegeben wird. Daher konnte Asteroids bisher funktionieren. Eine Analogie könnte ein Vorgarten sein, in den jeder hineinlaufen und mit den Dingen dort hantieren kann.

***Hinweis****: Im Klassendiagramm wird public mit dem Zeichen + markiert*

### private

Der Zugriff ist nur Objekten der gleichen Klasse erlaubt. In den meisten Fällen greift also das Objekt selbst auf seine eigenen Eigenschaften und Methoden zu. Eine Analogie könnte das Schlafzimmer der Eltern sein, auf dessen Interieur nur die Erwachsenen im Haus haben.

***Hinweis****: Im Klassendiagramm wird private mit dem Zeichen - markiert*

### protected

Der Zugriff ist nur Objekten der gleichen Klasse und deren Subklassen erlaubt. Eine Analogie ist der Rest des Hauses, den Eltern und Kinder nutzen können, aber anderen, die nicht zur Familie gehören, verschlossen bleibt.

***Hinweis****: Im Klassendiagramm wird protected mit dem Zeichen # markiert*

### readonly

Unterschiedliche Programmiersprachen können noch weitere Modifikatoren anbieten. Erwähnenswert ist für TypeScript noch readonly, das mit den vorangegangenen Modifikatoren kombiniert werden kann. Eine so gekennzeichnete Eigenschaft kann nur direkt bei der Deklaration oder im Konstruktor definiert, danach der Wert aber nicht mehr geändert werden. Es entspricht somit dem Schlüsselwort const für übliche Variablen.

## Zugriffsfunktionen

### set

Manchmal ist es wünschenswert, dass eine Eigenschaft eines Objektes verändert werden kann, so als wäre sie public, aber dass das Objekt selbst von der geplanten Änderung erfährt und noch eingreifen kann oder erforderliche Prozesse dabei auslöst. Beispielsweise könnte das Objekt überprüfen, ob der einzutragende Wert innerhalb bestimmter Grenzen liegt und die Übernahme verweigern, wenn dies nicht so ist. Hierzu könnte es also eine Methode anbieten und das eigentlich Attribut als private deklarieren, um nur selbst darauf Zugriff zu haben.

private value: number;

setValue(\_newValue: number): void {

if (\_newValue < 100)

this.value = \_newValue;

}

Soll nun also die Eigenschaft value des Objektes instance verändert werden, wird die Anweisung instance.setValue(...) genutzt. Da solche Anwendungsfälle häufig auftreten und die Schreibweise instance.value = ... für die Veränderung einer Eigenschaft schlicht einfacher und intuitiver erscheint ist, gibt es einen speziellen Methodentyp: den “Setter”

private valuePrivate: number;

set value(\_newValue: number): void {

if (\_newValue < 100)

this.valuePrivate = \_newValue;

}

Nun wird beispielsweise bei der Anweisung instance.value = 59; automatisch diese Setter-Methode aufgerufen und das, was auf der rechten Seite des Zuweisungsoperators steht, hier 59, als Parameter übergeben. Gleiches geschieht auch bei den kombinierten Zuweisungsoperatoren (siehe Anhang). Zu beachten ist, dass es nun gar keine Eigenschaft value in der Klasse mehr gibt, denn das würde einen Namenskonflikt mit dem Setter bedeuten. Daher wird intern im Beispiel das Attribut intern mit valuePrivate bezeichnet. Von “außen” betrachtet aber verfügt instance über eine Eigenschaft value die nun eine besondere Funktionalität aufweist und nicht unkontrolliert verändert werden kann.

* Die Vektorklasse verfügt bereits über eine Methode set(...). Ist das ein “Setter”?

### get

Das Pendant zu set ist natürlich get. Damit kann beispielsweise eine Eigenschaft gelesen werden, die gar nicht gespeichert ist, sondern erst berechnet wird, wenn jemand danach fragt. Ein Kandidat für einen sogenannten “Getter” könnte ein Attribut length der Vektorklasse sein. Die Länge eines Vektors lässt sich leicht mit dem Satz des Pythagoras berechnen, Javascripts Math-Objekt hält hierfür die Methode hypot bereit. Da die Elemente des Vektors sich ständig ändern, wäre es unnötiger Aufwand, bei jeder Änderung die Länge zu berechnen und zu speichern. Mit

get length(): number {

return Math.hypot(this.x, this.y);

}

erscheint es von außen betrachtet aber so, als wäre diese Eigenschaft ständig vorhanden, denn sie kann einfach mit z.B. let speed: number = velocity.length; abgerufen werden. Die Anweisung verschleiert, dass es sich eigentlich um den Aufruf einer Methode handelt.

***Hinweis:****Zugriffsfunktionen werden im Klassendiagramm standardmäßig nicht markiert, sondern schlicht als Eigenschaft dargestellt.*

# L11.2 Asteroids reloaded

## Aufzählungstypen

Häufig ist es erforderlich, eine Information mit einem Datentyp zu beschreiben, der nur eine enge und diskrete Auswahl an Werten beschreiben kann. Das simpelste Beispiel für einen solchen Datentyp ist boolean. Hier sind nur zwei Werte zulässig true und false. Der Versuch, einer Variablen dieses Typs beispielsweise ein maybe zuzuweisen, scheitert.

Du kannst selbst solche Aufzählungstypen mit dem reservierten Wort enum (Enumeration [engl]: Aufzählung) kreieren.

enum TASK {

WATCH,

PATROL,

CHASE,

SLEEP

}

Der Typ im Beispiel könnte genutzt werden, um die Aufgaben eines Wachhundes zu bezeichnen. Ein Objekt vom Typ des Wachhundes könnte dann über eine Eigenschaft task verfügen, welche nur diese vier Werte annehmen kann.

class Watchdog extends Dog {

private task: TASK = TASK.WATCH;

update(): void {

switch (this.task) {

case TASK.SLEEP:

...

break;

case TASK.WATCH:

...

break;

...

}

}

}

Wird die Methode update des Wachhundes nun zyklisch abgearbeitet, können je nach aktuellem task unterschiedliche Aktivitäten bzw. Verhaltensmethoden aufgerufen werden. Der switch stellt bereits das hierfür erforderliche Konstrukt dar.

* wie könnten die zugehörigen Aktivitäten aussehen? Erstelle grobe Aktivitätsdiagramme hierfür.

Ohne den speziellen Aufzählungstyp wäre task vielleicht einfach vom Typ number. Dann gäbe es die Aufgaben 0, 1, 2 und 3 und man müsste an anderer Stelle festhalten, welche Zahl welche Aufgabe bedeutet. Zudem wären dann auch Zahlen gültig, zu denen gar keine Aufgabe definiert ist. Da sind Fehler vorprogrammiert. task könnte aber auch vom Typ string sein und mit den Werten "sleep", "watch" und so weiter besetzt werden. Dann wäre das Programm wieder lesbar, ist aber sehr fehleranfällig, denn falsche Schreibweisen wie "petrol"schleichen sich ein.

Mit Hilfe des Aufzählungstyps aber kann TypeScript schon beim Schreiben des Codes die richtigen Vorschläge machen, die zulässigen Werte zur Auswahl stellen und Fehler sofort erkennen.

***Hinweis****: Laut UML-Standard wird eine Enumeration im Klassendiagramm durch die Markierung <<enumeration>> gekennzeichnet. Analog gilt das für Interfaces mit <<interface>>*

**L12\_Addition**

[Ein Bild, das Wasser, draußen, Meeresgrund enthält.

Automatisch generierte Beschreibung](https://www.youtube.com/watch?v=PK_yguLapgA) Quelle: https://docplayer.net/docs-images/66/55611304/images/11-0.jpg

* Klicke auf das Bild, wenn Du sehen möchtest, wie ein kleiner Softwarefehler 290 Millionen Euro in Rauch aufgehen lässt.

In diesem Kapitel lernst Du noch einige fortgeschrittene Strukturen kennen, die interaktive Anwendungen flexibler, wartbarer und sicherer machen. Auch wenn Du sie im Kurs vielleicht nicht verwendest, solltest Du sie gesehen und verstanden haben. Zudem werden einige Konstrukte vorgestellt, die nicht unbedingt empfehlenswert, aber weit verbreitet sind, und dich bei deinen Recherchen im Netz irritieren könnten.

**Standardparameter**

Neben dem Fragezeichen gibt es noch eine andere Möglichkeit optionale Parameter bei der Funktions- oder Methodendeklaration vorzusehen. Dabei gibt man einfach nach der Deklaration des formalen Parameters einen Zuweisungsoperator und dann einen Standardwert an. Wird nun beim Aufruf kein Parameter übergeben, erhält der formale Parameter automatisch den Standardwert. Praktisch ist das beispielsweise für den Konstruktor der Vector-Klasse.

constructor(\_x: number = 0, \_y: number = 0) {

this.set(\_x, \_y);

}

Nun kann ein neuer Vector einfach mit dem Aufruf new Vector() erzeugt werden, die Koordinaten haben dann automatisch beide den Wert 0. Es muss also nicht im Methodenrumpf geprüft werden, ob Parameter übergeben wurden um auf diesen Fall zu reagieren. Werden aber Parameterwerte übergeben, werden diese verwendet.

**Eigenschaftsparameter**

Eigenschaften eines Objektes mit Werten zu definieren, die dem Konstruktor übergeben werden, ist ein sehr häufiges Muster. Daher ist in Typescript eine Kurzschreibweise möglich. Dabei werden die Eigenschaften mit der Signatur des Konstruktors durch die Liste der formalen Parameter deklariert.

class Test {

constructor (public a: number, protected b: number, private c: number) {

}

}

Eine Instanz der Klasse Test verfügt nun über die Eigenschaften a, b und c, obwohl diese nicht in der Klasse deklariert wurden, sondern in der Konstruktorsignatur. Erzeugt mit let test: Test = new Test(1, 2, 3); sind die Eigenschaften dann mit den Werten 1, 2 und 3 definiert.

Diese Schreibweise “versteckt” allerdings die Eigenschaften in der Signatur und der Coding-Style der Lektionen kann nicht sinnvoll eingehalten werden, da es sich gleichzeitig um Parameter und Eigenschaften handelt. Daher wurde diese Schreibweise nicht verwendet, soll hier aber erwähnt sein.

**Ausnahmebehandlung**

In den vorangegangenen Lektionen wurde das Thema Fehlerbehandlung weitestgehend ausgeklammert. Lediglich wenn TypeScript durch die strikten Einstellungen des Compilers und des Linters Fehlerquellen angezeigt hat, wurde der Code entsprechend angepasst.

Gerade wenn ein System mit anderen Systemen oder mit Menschen kommuniziert, entstehen viele Fehler zur Laufzeit und sind schwer vorhersehbar. Wird zudem versucht, alle möglichen Fehler durch Verzweigungen im Code abzufangen, steigt die Komplexität des Programms rasant an.

interface Greet {

greet: string;

}

let greets: Greet[] = [{ greet: "Hi" }, { greet: "Hallo" }, { greet: "Servus" }];

let input: string | null = prompt("Lass dich grüßen!", "Gib hier eine Zahl ein");

let greet: string = greets[Number(input)].greet;

alert(greet);

console.log("Done");

* Implementiere das obenstehende Mini-Programm in einem eigenen Namespace und lasse es im Browser laufen. Was geschieht, wenn Du eine Zahl jenseits 0, 1 und 2 eingibst oder beliebige Zeichen?

Wenn das Programm seinen Dienst quittiert, tut es dies leise. Der Nutzer hat keine Ahnung was passiert, sofern er nicht Medienkonzeption in Furtwangen studiert deswegen selbstverständlich die Entwicklerkonsole geöffnet hat. Dort erscheint eine Fehlermeldung.

* Erkläre den Fehler!

Vielleicht ist der Fehler aber gar nicht so gravierend, dass das Programm beendet werden müsste. Oder der Nutzer sollte darüber informiert werden, dass etwas schiefgelaufen ist, so dass er die Ursache prüfen kann. Im Beispiel könnte man natürlich einfach abfragen, ob die Eingabe gültig ist. Es gibt aber noch einen anderen Mechanismus, der auch mit noch unbekannten Fehlerquellen umgehen kann: **Exception-Handling!**

**Error-Objekt**

Javascript kann bei einem solchen Ausnahmefehler (Exception), der zum Absturz des Programms führt, automatisch ein Error-Objekt erzeugen. Ähnlich einem Event, das durch das DOM weitergereicht wird, kann dieses Objekt durch das Programm gereicht, eher sogar “geworfen” werden. Dieses Objekt enthält Informationen über das Problem, die ausgewertet werden können. Damit müssen also nicht alle Fehler präventiv vermieden werden, sondern sie können im Nachhinein behandelt werden.

**try**

Um den Mechanismus zu nutzen wird zunächst der Code-Block eingegrenzt, der den zu behandelnden Fehler provozieren könnte. Dies geschieht, wie üblich, durch Einfassen in geschweifte Klammern, nun aber mit dem reservierten Wort try davor. Damit wird Javascript angewiesen, zu versuchen den Code im Block auszuführen und ein Error-Objekt zu werfen, wenn etwas schief geht.

**catch**

An diesen Block muss sich nun direkt ein Block anschließen, der mit dem reservierten Wort catch versehen ist. Jetzt wird auch die Wurf-Analogie deutlich, denn hier wird das Error-Objekt aufgefangen und kann innerhalb des Blocks verarbeitet werden. Dazu wird nach der catch-Anweisung ein formaler Parameter deklariert, wie es auch bei Funktionen oder Methoden üblich ist. Allerdings kommt eine Eigenheit von TypeScript zum Tragen: wie bei der for..in- und der for..of-Loop ist keine Typ-Annotation erlaubt.

Mit try und catch gestaltet sich obiger Code zum Beispiel so:

namespace L12\_Exception {

interface Greet {

greet: string;

}

let greets: Greet[] = [{ greet: "Hi" }, { greet: "Hallo" }, { greet: "Servus" }];

try {

let input: string | null = prompt("Lass dich grüßen!", "Gib hier eine Zahl ein");

let greet: string = greets[Number(input)].greet;

alert(greet);

} catch (\_error) {

alert("Tschüss!");

console.log(\_error);

}

console.log("Done");

}

* Erweitere deinen Beispielcode entsprechend und experimentiere dann damit.

**finally**

Nach dem catch-Block kann sich noch ein finally-Block anschließen. Der Code darin wird auf jeden Fall ausgeführt, egal ob es eine Exception gab oder nicht. Selbst wenn der Beispielcode in einer Funktion definiert ist, und sowohl im try- als auch im catch-Block eine return-Anweisung steht, die Funktion dort also zurückkehrt, wird zuvor noch der finally-Block abgearbeitet. Die Ausgabe “Done” aber kommt dann nicht mehr.

**throw**

Du kannst auch selbst eine Exception erzeugen und werfen lassen. Das entspricht etwa der Erzeugung eines CustomEvents, das Du durch das DOM schickst. Das Error-Objekt wird die Kette der Funktionsaufrufe zurück geschickt, bis es in einer der aufrufenden Funktionen von einem catch-Block verarbeitet wird. Geschieht letzteres nicht, kommt es als “Unhandled Exception” auf der Konsole raus…

Das bedeutet, dass nun neben return und await eine dritte Möglichkeit zur Verfügung steht, um Funktionen zu beenden oder zu unterbrechen. Je nach Anwendungsfall kann die Verwendung von throw äußerst effizient sein.

* Verfolge den Code “Exception” im Anhang zu dieser Lektion mit dem Debugger!

**Darstellung im Aktivitätsdiagramm**

Es gibt zwei Perspektiven um die Ausnahmebehandlung im Aktivitätsdiagramm darzustellen. Bei der “Außenansicht” wird nur gezeigt, dass eine Aktivität mit einer Ausnahme beendet werden kann und wie der Fluss dann weitergeht. Dazu kann hier - eben ausnahmsweise - die Aktivität zwei Abflüsse haben, die dann aber nicht eine parallele Weiterverarbeitung bedeuten. Denn einer davon ist mit einem kleinen Dreieck gekennzeichnet und gegebenenfalls, wie bei einem bedingten Fluss, mit einer Information zu der Ausnahmebedingung in eckigen Klammer.

Die “Innenansicht” zeigt den inneren Aufbau eines unterbrechbaren Bereiches und die Signale, die eine Ausnahmebehandlung erforderlich machen. Der Rahmen des Bereiches wird gestrichelt dargestellt, der Abfluss vom Signalempfang als gezackte Linie. Neben der gezackten Linie und dem Dreieck an einer Linie gibt es noch die Darstellung mit einem Blitzsymbol an der Linie.

**Funktions-Objekt**

Funktionen sind in Javascript, wie alles andere, Objekte. Sie können beispielsweise mit Hilfe von Variablen referenziert werden. Folgendes ist also möglich:

function doSomething(\_parameter: number): string {

...

}

let doToo: Function = doSomething;

doToo(42);

Das Ergebnis ist das gleiche wie beim Aufruf doSomething(42); denn doToo und doSomething referenzieren das gleiche Funktionsobjekt.

**Anonyme Funktionen**

Die Funktion kann auch gleich bei der Erzeugung einer Variablen zugeordnet werden, dann ist es nicht erforderlich einen Funktionsnamen zu vergeben. Die Funktion selbst ist dann “anonym”.

let doSomething: Function = function(\_parameter: number): string {

...

}

Die Funktionserzeugung ist gleichbedeutend mit der im Beispiel darüber. Allerdings kann doSomething(...) nun erst nach der Definition aufgerufen werden, was eher unschön ist.

Anonyme Funktionen werden häufig dort genutzt, wo sie als Parameter übergeben werden und nicht als mehrfachverwendbare Objekte im Zugriff bleiben müssen. Beispiel:

setTimeout(function (): void {

console.log("Timeout");

}, 2000);

Der Timeout-Funktion wird das anonyme Funktionsobjekt sofort übergeben und nach zwei Sekunden wird es aufgerufen. Danach ist das Funktionsobjekt Müll und wird vom Garbage Collector irgendwann entsorgt.

Für solche kleinen Funktionen ist diese Anonymität akzeptabel, grundsätzlich vermindern solche Konstrukte aber eher die Lesbarkeit und Wartbarkeit des Programms.

**Arrow-Functions**

Seit 2015 ist in Javascript eine weitere Schreibweise für Funktionen üblich, die etwas kryptischer aussieht, aber im Wesentlichen gleichbedeutend ist mit der dir bekannten und etwas intuitiveren Schreibweise mit function.

let doSomething: Function = (\_parameter: number): string => {

...

};

Aus dem reservierten Wort function vor der Parameterliste ist nun der Doppelpfeil zwischen dem Rückgabetyp und dem Funktionskörper geworden.

**this-binding**

Ein Unterschied aber ist die Funktionalität innerhalb von Klassen. Wird eine Arrow-Function als Methode deklariert, wird sie in der Konsole als Eigenschaft angezeigt. Interessanter aber ist der Unterschied bei der Verwendung als Event-Handler.

* Innerhalb klassischer Methoden verweist this dann auf das Event-Target und nicht auf die Instanz der Klasse. Das führt häufig zu unerwarteten Fehlern.
* Bei Arrow-Functions dagegen verweist this weiterhin, wie zu erwarten, auf die Instanz. Daher ist diese Schreibweise in dem Fall zu bevorzugen.
* Es gibt auch die Möglichkeit, eine klassische Methode so zu modifizieren, dass auch sie als Event-Handler die Instanz mit this referenziert. Hierzu existiert die Funktion bind des Funktionsobjektes.
* Wenn Du dies besser verstehen möchtest, experimentiere etwas mit dem Code “FunctionObject” im Anhang.

**Drag&Drop**

Eine wichtige Standardfunktionalität des DOM wurde in diesem Kurs bislang noch nicht behandelt. Insbesondere bei Desktopanwendungen ist es sehr üblich, Objekte direkt zu manipulieren, nicht also über Formularelemente, sondern beispielsweise per drag & drop. Die Verwendung ist recht einfach:

* draggable ist eine Eigenschaft, definiert in der Klasse HTMLElement. Erhält sie den Wert true verändert sich das Maus-Verhalten auf dem entsprechenden Element.
* wird die Maustaste gedrückt, während sich der Mauscursor über dem Element befindet, wird ein Event vom Typ dragstart gefeuert.
* das Event-Objekt ist vom Typ DragEvent und verfügt über eine Eigenschaft dataTransfer, die ein Objekt vom Typ DataTransfer referenziert.
* die Methode setData(key, value) fügt dataTransfer Informationen hinzu.
* ein HTMLElement, auf dem dann der Drop stattfinden können soll, muss diesen zulassen. Dazu muss es auf ein dragover-Event hören und mit event.preventDefault() sein Standardverhalten unterbinden, welches die Ablehnung des Drops ist.
* nun kann es auf ein drop-Event reagieren, wobei dem entsprechendem Handler das beim drag erzeugte und mit zusätzlichen Informationen versorgte Event übergeben wird.
* mit getData(key) können die Informationen ausgelesen und verarbeitet werden, zum Beispiel um das Drop-Ziel zu verändern.
* Im Anhang befindet sich ein kleines Beispielprogramm “DragDrop” mit einer rudimentären Implementation des DOM-Drag&Drop-Mechanismus, mit dem noch viel mehr möglich ist. Schaue es dir an!

**Garbage Collection**

Wenn eine Variable ihren Gültigkeitsbereich verlässt, wird der von ihr belegte Speicher freigegeben und steht wieder für andere Informationen zur Vefügung. Bei Variablen, die auf Objekte verweisen, wird aber nur der Verweis gelöscht, das referenzierte Objekt dagegen bleibt im Speicher, denn es könnten noch andere Variablen darauf verweisen. Diese würden dann ins Leere deuten und Fehler erzeugen. Werden die Objekte allerdings nie gelöscht, wird der Speicher immer mehr zugemüllt und es kommt irgendwann zu einem Programmversagen.

Bei einem Javascript-Programm wird der Speicher daher von einem Algorithmus, dem sogenannten “Garbage Collector” überwacht. Dieser Müllmann durchforstet in unregelmäßgen Zeitabständen den Speicher, findet Objekte die nicht mehr gebraucht werden und löscht diese. In den meisten Fällen geschieht das völlig unmerklich, bei Animationen allerdings kann dieser Vorgang zu sichtbaren Störungen führen, wenn ein Bild einmal etwas länger stehen bleibt als andere. Durch Wiederverwendung von Objekten kann der Effekt minimiert werden.

**Debugger in VSCode**

**Node-Programme**

In der Lektion zum Server wurde bereits kurz der interne Debugger von VSCode vorgestellt. Da bei Node-Programmen der Browser nicht beteiligt ist, kann dessen Debugger nicht genutzt werden um den Programmablauf zu prüfen. Beim Umgang mit dem Debugger von VSCode sind noch einige Informationen sehr nützlich. Da VSCode gleichzeitige mehrere Projekte verwalten kann, braucht der Debugger beim Start die Information, welches Programm ausgeführt und verfolgt werden soll. Im einfachsten Fall, wenn der Debugger “No Configurations” anzeigt, startet er das Programm, das gerade im fokussierten Editor-Fenster angezeigt wird. Mit Hilfe des Menüsystems lassen sich aber auch gezielt Konfigurationen definieren, diese werden in einer Datei launch.json im Ordner .vscode gespeichert, die der Debugger automatisch dabei anlegt.

* Öffne den Ordner X01\_Appendix mit VSCode, navigiere zur Debug-Ansicht und darin zu Open launch.json bzw. Open Configurations. Nun sollte dir die Datei launch.json angezeigt werden.
* Studiere den ersten Eintrag im Array configurations. Beachte, dass ${workspaceFolder} ein Platzhalter ist, an dessen Statt der Pfad zu dem Verzeichnisses eingesetzt wird, dass Du mit VSCode geöffnet hattest. In unserem Fall also der Pfad zu X01\_Appendix.
* Finde den Namen dieser ersten Konfiguration an anderer Stelle im Debug-View. Stelle dann sicher, dass diese Konfiguration genutzt wird, wenn Du den Debugger startest.
* Starte nun den Debugger z.B. mit F5 oder Klick auf den grünen Pfeil.
* Beobachte die Ausgabe im Debug-Terminal und im Editor-Fenster.

Zeile 3 enthält die bekannte Anweisung debugger deswegen wurde hier der Programmfluss angehalten und dir wird das Control-Panel mit den Optionen angezeigt, die Du schon vom Browserdebugger kennst. Statt diese Anweisung in den Code zu schreiben, kann man natürlich auch, wie üblich, Breakpoints einfach durch Klick vor der Zeilennummer einsetzen.

* Führe zwei mal einen “Step over” aus um die nächste Konsolenausgabe zu sehen.
* Erkläre, woher der Gruß kommt, der nun erscheint.

**Browser-Programme**

Aber auch bei Browser-Programmen muss man nicht auf den Komfort des VSCode-Debuggers verzichten. Über einen lokalen Server kann VSCode mit dem Browser kommunizieren und das laufende Programm kontrollieren. Hierfür ist noch ein wenig Einrichtung erforderlich. Das folgende Beispiel beschreibt dies für die Nutzung von Chrome.

* Installiere die Extension “Debugger for Chrome” in VSCode.
* Betrachte den zweiten Eintrag der configurations in launch.json. Analog zu programm gibt es hier den Eintrag url. Hier ist der Pfad zu der auszuführenden HTML-Datei hinterlegt, bezogen auf das Wurzelverzeichnis eines lokalen Fileservers, dem Verzeichnis, dass Du mit VSCode geöffnet hattest.
* Starte den Fileserver im Wurzelverzeichnis mit npx serve oder, wenn Du “serve” bereits global installiert hattest, einfach mit serve.
* Aktiviere die Debug-Konfiguration “Launch DebugBrowser with Chrome” von launch.json.
* Starte den Debugger. Beachte, dass nun Chrome mit der Web-App gestartet wird.
* Erkläre, woher jetzt die Grußnachricht kommt.

**Teamarbeit**

**Git** hat noch mehr Funktionen als die bisher genutzten, sehr wichtig für die Teamarbeit sind dabei “Branches” und “Forks”.

**Branches**

Dabei können mehrere Versionen des gleichen Repositories gleichzeitig gespeichert werden, sie werden als Branches bezeichnet. Bislang hast Du nur mit dem master-Branch gearbeitet. Insbesondere, wenn mehrere Leute am oder mit dem Code arbeiten, ist es äußerst wünschenswert, einen Branch mit einem stabilen und getesteten Zustand zu halten (oft master), während es weitere Branches gibt, auf denen die Fortentwicklung und Experimente etc. stattfinden. Ist dann in einem Branch etwas entstanden, das in den master einfließen soll, so kann **git** diese Verschmelzung, im Englischen “merge” genannt, übernehmen. Nur wenn es Konflikte gibt, also in beiden zu verschmelzenden Branches die gleiche Datei verändert wurde, muss ein Mensch entscheiden, welcher Code erhalten bleiben soll.

**Forks**

Mit diesem System ist auch die Open-Source-Entwicklung in einer Community möglich, ohne dass die Teammitglieder in einer festgelegten Struktur arbeiten und sich noch nicht einmal kennen müssen. Jeder kann sich auf Github einen Klon (**Fork**) eines öffentlichen Repositories auf seinen Github-Bereich erstellen lassen, und daran arbeiten. Entsteht dabei etwas, das für das Original nützlich sein könnte, kann man einen “Pull-Request” verschicken, in dem automatisch die Veränderungen enthalten sind. Die Entwickler des Originals können diese begutachten und bei Bedarf in ihr Repository “mergen”.

**Plattformen**

Die technische Basis, die für diese Modulreihe “Entwicklung interaktiver Anwendungen” herangezogen wird, entstammt der Webtechnologie. Das bedeutet aber nicht, dass deine Konzeption sich auf Internetseiten beschränken muss. Mit dem, was Du jetzt gelernt hast, kannst Du vollwertige native Apps und Desktop-Programme konzipieren und entwickeln. Hierzu verwendest Du Techniken, die deine Software auf eine erweiterte Plattform hieven, im Inneren arbeitet dabei weiter ein Browser und/oder Node, der Nutzer bekommt davon aber nichts mit.

**Electron**

Wird von GitHub selbst entwickelt und verwendet Node.js sowie Chromium, die Entwicklerversion des Chrome-Browsers. Der Funktionsumfang wird erweitert um Systemfunktionen wie Zugriffe auf das Dateisystem, so dass aus deiner Web-App eine vollwertige, native Desktopanwendung für Windows, Mac und Linux werden kann.

Du arbeitest übrigens schon eine Weile mit einer solchen Anwendung, den Visual Studio Code ist eine Electron-App!

**Cordova/PhoneGap**

Cordova und PhoneGap sind zwei im Wesentlichen identische Produkte, die denselben Ursprung haben und von Apache beziehungsweise Adobe weiterentwickelt werden. Ihnen liegt das gleiche Prinzip zugrunde wie Electron, allerdings für mobile Android- und iOS-Geräte. So kann deine WebApp über die entsprechenden Stores vertrieben werden und im Betrieb auf GPS, Beschleunigungssensoren, Kamera etc. zugreifen.

Die Wikipedia mobile App ist eine PhoneGap-Anwendung!

**Progressive Web Apps**

Seit 2015 schickt sich ein weiteres Modell an den App-Markt aufzumischen: Progressive Web Apps. Diese kommen zunächst wie eine gewöhnliche Web-App daher, installieren aber beim ersten Besuch einen “Service-Worker” auf dem Smartphone. Diese Software lädt gezielt Inhalte herunter und speichert sie auf dem Gerät, um die Funktionalität der App auch zu erhalten, wenn keine Internetverbindung besteht. Es wird zudem angeboten, eine Verknüpfung auf dem Home-Screen zu erstellen, sie kann im Vollbildmodus gestartet und Push-Nachrichten vom Server angezeigt werden, auch wenn die App gerade nicht aktiv ist. Dadurch wirkt die PWA wie eine vollwertige native App bei minimalem Mehraufwand gegenüber der reinen Web-Entwicklung. Der riesige Vorteil: ist erst einmal ein potenzieller Nutzer über eine Suchmaschine auf dem Angebot gelandet, muss er es nicht erst wieder verlassen, in den Store geleitet werden um dort die Installation einer App und später den Start derselben in die Wege zu leiten, sondern die Nutzung der App hat bereits begonnen!

Der mobile Twitter Client ist eine Progressive Web App

**Abschluss**

**eiaSteroids**

Das Beispielprojekt hat nun auch diesen eigenen und eigenwilligen Namen bekommen. Schau es dir genau an, insbesondere die finale Dokumentation. Es ist beispielhaft für deine Abschlussarbeit und Du solltest dich daran orientieren! Hier geht’s also zu [**eiaSteroids**](https://jirkadelloro.github.io/EIA2-Inverted/X01_Appendix/eiaSteroids)