Javascript Klassen

SPENGERGASSE

```
class Person {
  constructor(name, age) {
    this.name = name; // 'name' is a property
    this.age = age;   // 'age' is a property
  }
}

const person = new Person('Alice', 25);
console.log(person.name); // 'Alice' (accessing a property)
person.age = 26;          // Modifying a property
```

constructor

Ist eine function, die `this` Referenz steht bereits zur Verfügung. Unterschied zu Java: Der Konstruktor heisst *immer* `constructor`!

get / set

Getter und Setter sind lustig in Javascript und funktionieren analog zum Konzept in C#:

```
class Temperature {
  constructor(celsius) {
    this._celsius = celsius;
  }
  get celsius() {
    return this._celsius;
  }
  set celsius(value) {
    if (value < -273.15) {
      console.log("Temperature cannot be below absolute zero.");
    } else {
      this._celsius = value;
    }
  }
}

const temp = new Temperature(25);
temp.celsius = -300; // Invalid, value not updated
console.log(temp.celsius); // 25
```

Vorsicht: Wenn so ein getter oder setter resourcen-intensiv ist, könnte diese Syntax dazu verführen, unperformanten Code zu schreiben!

properties / attributes

In der Web-Welt wird unter ”property” üblicherweise der ”key” oder das ”field” eines Objektes bezeichnet, wohingegen ”attribute” sich üblicherweise auf Eigenschaften von HTML-Elementen bezieht.

Ich verwende diese Worte im Unterricht oft synonym und sage zu den Properties auch gerne ”member variablen”.

Grundsätzlich kann man im Nachhinein jedem Objekt auch im Nachhinein (nach Erzeugung) jede beliebige Variable neu setzen.

Nerd Stuff: `Object.preventextensions(), Object.freeze(), Object.seal(), Proxy`

private / public

1. private ”by convention”: propertyname beginnt mit ”_”: `this._privateVar`. Man sollte Alarmglocken hören bei z.B. `o._privateVar = 7`!
2. really private: propertyname beginnt mit '#'. `this.#privateVar` dies ist sicherer und wird mittlerweile voll unterstützt.

Vererbung

- Wie in Java: ”class Dog extends Animal”

- wie in java kann (und soll) auch `super()` aufgerufen werden.
- Subklasse kann methoden überschreiben
- subklasse kann auch komplett neue Methoden bekommen

ich gehe nicht auf prototypal inheritance ein. Dies ist Schnee von gestern und spielt in zeigenössischen codebases keine Rolle mehr. Allerdings ist die Sache mit dem prototypes immer noch, wie javascript unter der Motorhaube Vererbung implementiert.

throw / try / catch / finally

Hat in dieser Folie eigentlich nix zu suchen. Evtl. sei erwähnt es gibt keine ”checked exceptions” wie in Java. Man kann zu jeder Zeit `throw new Error("Error Message")` aufrufen.

Man kann von `Error` erben.

Man kann im `catch` Block mit `instanceof` abfragen, wo der Error in der Hierarchie ist, wenn man das möchte. Es ist auch möglich, beliebige Objekte zu `throw()`-en, z.B. Strings oder Numbers. Ist aber nicht empfehlenswert.

Der `finally` Block ermöglicht es, bestimmte Aktionen auszuführen, egal ob es eine Exception gab oder nicht (um aufzuräumen, üblicherweise).

methoden

analog wie java

static

auch dies sehr analog wie Java. Dient zum Schreiben von utility-Funktionen. Man braucht keine Instanz, um die Methode zu verwenden. Innerhalb der static Methode ist natürlich `this` nicht möglich. Beispiel wäre `Array.from()`. ”Array” ist die Klasse, ”from” die Methode.