Web-Entwicklung

Objektorientierte Programmierung

Inhalte der Vorlesung

- Objektorientierte Programmierung
  - Objekterzeugung
  - Prototypen
  - Vererbung
    - Prototypische Vererbung
    - Pseudoklassische Vererbung
    - Klassensyntax
  - Getter/Setter und statische Eigenschaften
  - Datenkapselung
    - Private Eigenschaften
    - Module
      - Entwurfsmuster: Immediately Invoked Function Expression, Revealing Module
      - CommonJS-Module
      - Native ES-Module (ESM)

Objekterzeugung

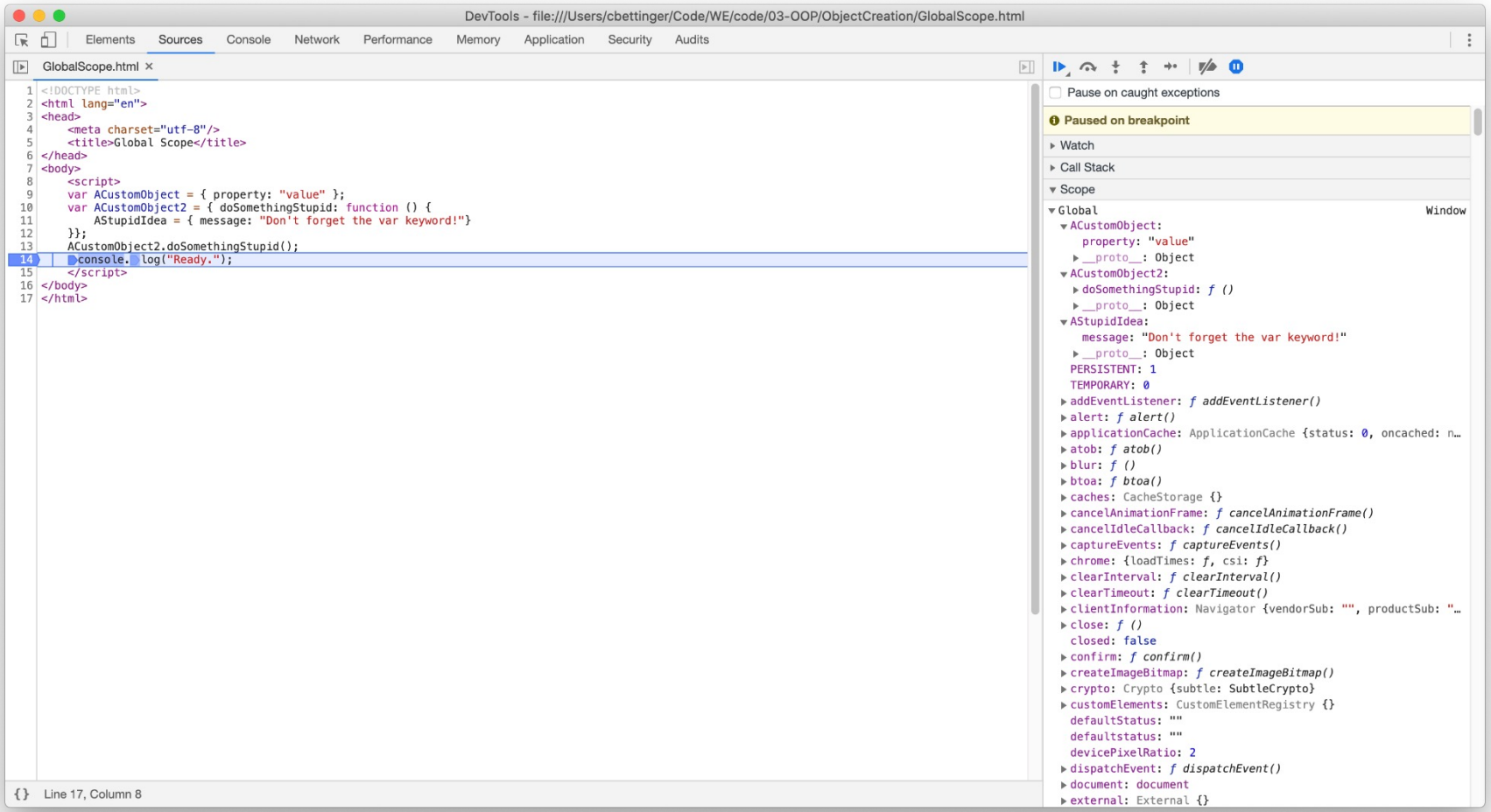
Globaler Gültigkeitsbereich (*global scope*)

- Standardobjekte: Im ECMAScript spezifiziert und in allen Laufzeitumgebungen vorhanden
  - `Object, String, Number, Array, Math, Date, ...`
- Host-Objekte: Von einer spezifischen Laufzeitumgebung bereitgestellte (also nicht-standardisierte) Objekte, z.B.
  - `console` und `process` in Node.js
  - `console`, `window` und `document` im Browser
- Benutzerdefinierte globale Variablen (sollten möglichst vermieden werden)
  - Deklaration von Variablen mit Schlüsselwort `let/var` im äußersten Gültigkeitsbereich einer Browser-Anwendung (d.h. außerhalb jeder Funktion)
  - Zuweisung an Variable ohne vorherige Deklaration mit `let/var` in beliebigem Gültigkeitsbereich

Globales Objekt (*global object*)

- Node.js: Jedes Modul spannt eigenen Gültigkeitsbereich auf
  - Es gibt einen globalen Gültigkeitsbereich mit Standard- und Host-Objekten, aber kein globales Objekt und keine benutzerdefinierten globalen Variablen
- Im Browser sind die Objekte im globalen Gültigkeitsbereich als Eigenschaften des sog. globalen Objekts zusammengefasst
  - Host-Objekt `window`
  - rekursiv: `window === window.window === window.window.window`

Globaler Gültigkeitsbereich



Objekterzeugung

- Objekt-Literal
- Konstruktorfunktion
- `Object.create()`
- Klassensyntax

Objekt-Literal

```
let p1 = {
   firstName: "Max",
   lastName: "Mustermann",
   print: function () {     // method
      console.log("%s %s",
         this.firstName, this.lastName);
   }
};
p1.print();
```

```
let p2 = {
   firstName: "Hans",
   lastName: "Wurst",
   car: {          // nested object
      manufacturer: "ACME",
      model: "Pinky"
   },
   print: function () {
      console.log("%s %s (Car: %s %s)",
      this.firstName,
      this.lastName,
      this.car.manufacturer,
      this.car.model);
   }
};
p2.print();
```

Konstruktorfunktion

```
function Person(firstName, lastName) {
   this.firstName = firstName;                // properties of created objects
   this.lastName = lastName;
   this.print = function () {
      console.log("%s %s", this.firstName, this.lastName);
   };
}

let p1 = new Person("Hans", "Wurst");
p1.print();

let p2 = new Person("Max", "Mustermann");
p2.print();

new Person("Eva", "Meier").print();
```

Object.create()

```
let p1 = Object.create({});
p1.firstName = "Max";
p1.lastName = "Mustermann";

console.dir(p1);
```

```
let p2 = Object.create({}, {
   firstName: {
      value: "Hans",
      writable: false,
      enumerable: true,
      configurable: false
   },
   lastName: {
      value: "Wurst",
      writable: false,
      enumerable: true,
      configurable: false
   }
});

console.dir(p2);
```

Object.create()

```
let Person = {
   firstName: null,
   lastName: null,
   print: function () { console.log("%s %s", this.firstName, this.lastName); }
};

let p3 = Object.create(Person);
p3.print();

let p4 = Object.create(Person);
p4.firstName = "Max";
p4.lastName = "Mustermann";
p4.print();

let p5 = Object.create(Person, { firstName: { value: "John" }, lastName: { value: "Smith" } });
p5.print();

let p6 = Object.create(Person);
p6.firstName = "Christian";
p6.lastName = "Bettinger";
p6.age = 40;            // dynamically added property
console.dir(p6);
```

Object.create()

```
let Person = {
   firstName: null,
   lastName: null,
   print: function () { console.log("%s %s", this.firstName, this.lastName); }
   init: function (firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
      return this;
   }
};

let p7 = Object.create(Person).init("Jane", "Doe");
p7.print();
```

Klassensyntax

- Populärste Neuerung in ES6: Alternative Syntax zur Realisierung der sog. pseudoklassischen Vererbung
  - Schlüsselwörter und Syntax lehnen sich an gewohnte Konstrukte aus Java/C++/C# an
    - Klassen mit Konstruktoren
    - Ableitung von Klassen
    - Getter-/Setter-Methoden
    - Statische Eigenschaften, Methoden und Initialisierer
    - `class`, `extends`, `constructor`, `super`, `get`, `set`, `static`
  - Zur Laufzeit gibt es weiterhin nur Objekte – keine Klassen!

Klassensyntax

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   print() {
      console.log("%s %s", this.firstName, this.lastName);
   }
}

let p1 = new Person("Hans", "Wurst");
p1.print();

let p2 = new Person("Max", "Mustermann");
p2.print();

new Person("Eva", "Meier").print();
```

Prototypen

Prototypen

- JavaScript kennt keine klassenbasierte Objektorientierung
  - Es gibt keine Klassen, nur Objekte!
- Stattdessen: Prototypische Objektorientierung
  - Jedes Objekt kann als Vorlage (Prototyp) für andere Objekte dienen
  - Konvention: Bezeichner der Prototypen beginnen mit einem Großbuchstaben, Bezeichner anderer Objekte mit einem Kleinbuchstaben
- Genauer:
  - Jedes Objekt besitzt eine Referenz auf ein Prototyp-Objekt
    - Lesender Zugriff via `Object.getPrototypeOf()`
    - Eigenschaft `__proto__` erst seit ES6 standardisiert
  - Wird auf Eigenschaften eines Objekts zugegriffen und diese existieren nicht, werden diese Eigenschaften im Prototyp (bzw. dessen Prototyp usw.) gesucht („Prototypenkette“)
    - So „erben“ alle Objekte bspw. die Methode `toString()` des Wurzelobjekts `Object.prototype`
  - Prototyp ist `null`, falls
    - Wurzel des Standard-Prototypenbaums erreicht (`Object.prototype`)
    - Prototyp eines Objektes wurde explizit auf null gesetzt wurde (`Object.create(null)`)

Prototypen (Object.create())

- Bei der Verwendung von `Object.create()` wird der Prototyp explizit als erstes Argument übergeben
- Kann auch explizit auf `null` gesetzt werden
  - Sofortiges Ende der Prototypenkette

```
let Person = {
   firstName: null,
   lastName: null,
   print: function () {
      console.log("%s %s",
         this.firstName, this.lastName);
   }
};

let p = Object.create(Person);
console.log(Object.getPrototypeOf(p));
          // { firstName: null,
          //   lastName: null,
          //   print: [Function: print] }

let o = Object.create(null);
console.log(o.toString());    // TypeError: o.toString
                              // is not a function
```

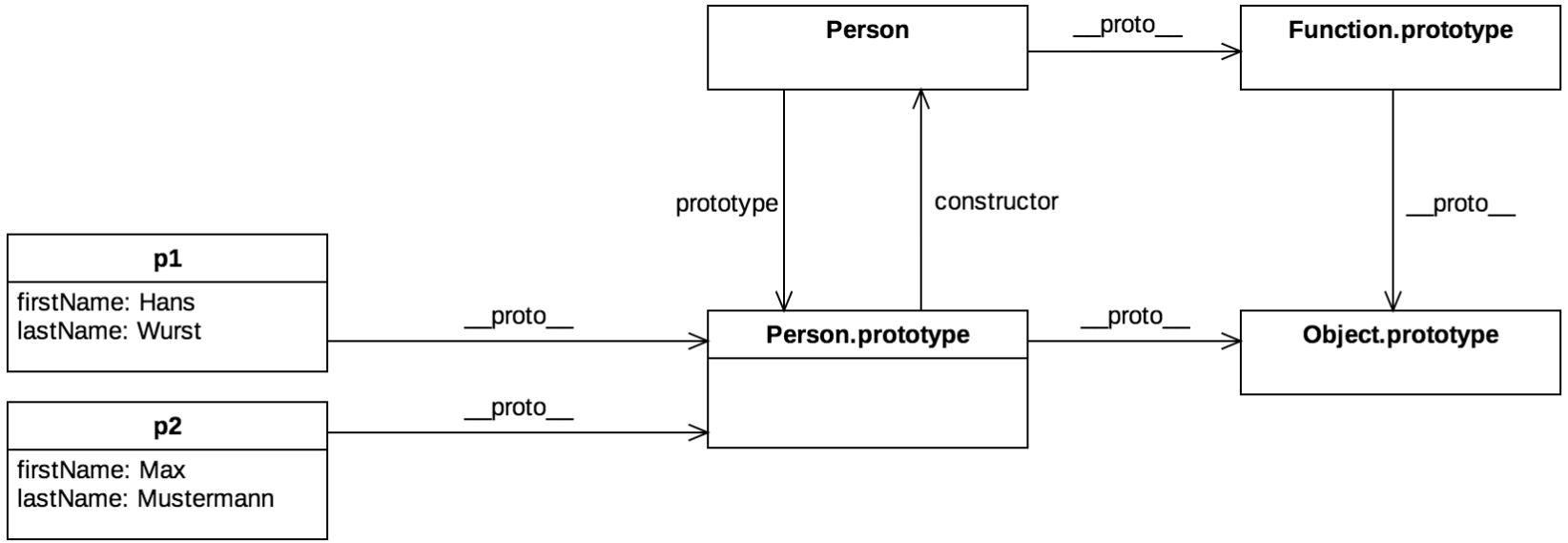
Prototypen (Konstruktorfunktion)

- Konstruktorfunktionen verwenden ein implizit erzeugtes Prototyp-Objekt
  - Zugriff über
    - Eigenschaft `prototype` der Konstruktorfunktion
    - Anwendung der Funktion `Object.getPrototypeOf()` auf die darüber erstellten Objekte
    - Eigenschaft `__proto__` der darüber erstellten Objekte
- Eigenschaft `constructor` des Prototyps verweist umgekehrt auf die Konstruktorfunktion
- `instanceof-Operator` prüft, ob die `prototype`-Eigenschaft einer Konstruktorfunktion in der Prototypenkette eines Objekts vorkommt
  - Syntax: `p1 instanceof Person`

Prototypen (Konstruktorfunktion)

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}

let p1 = new Person("Hans", "Wurst");
let p2 = new Person("Max", "Mustermann");

console.log(Object.getPrototypeOf(p1));                          // Person {}
console.log(Object.getPrototypeOf(p2));                          // Person {}
console.log(Person.prototype);                                   // Person {}
console.log(Object.getPrototypeOf(p1) === Person.prototype);     // true
console.log(Person.prototype.constructor === Person);            // true

console.log(p1 instanceof Person);      // true
console.log(p1 instanceof Animal);      // false
```

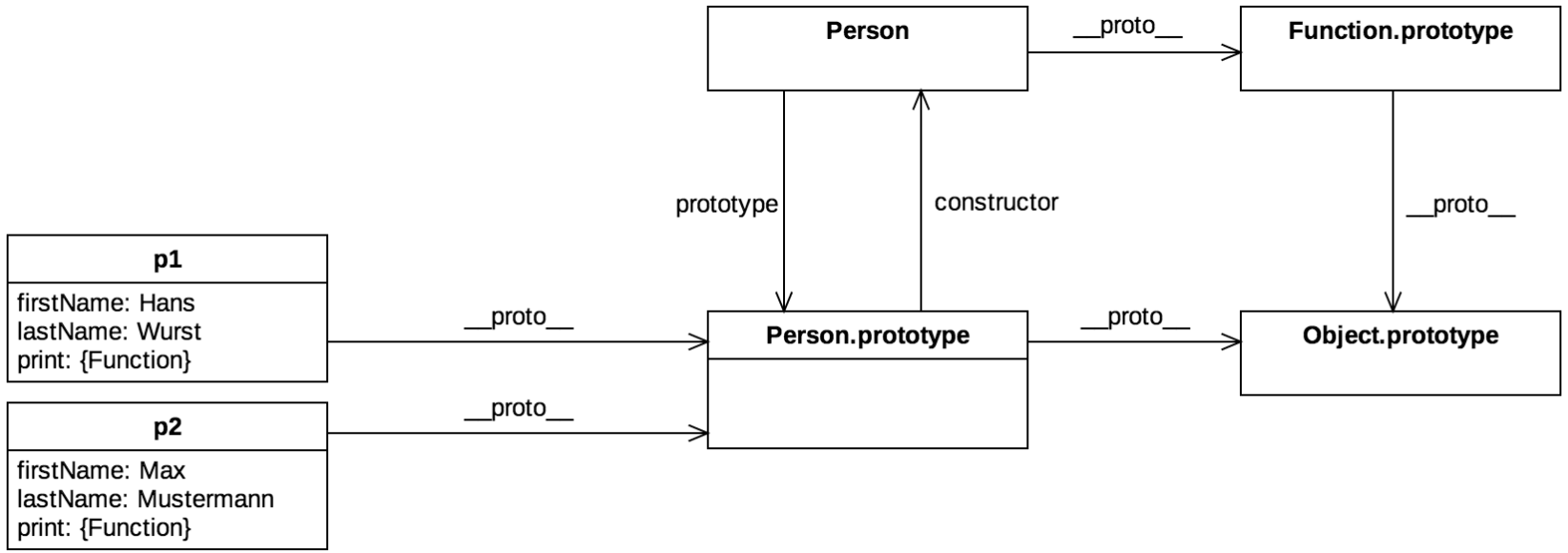
Prototypen (Konstruktorfunktion)



Prototypen (Konstruktorfunktion)

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.print = function() {
      console.log("%s %s", this.firstName, this.lastName);
   }
}

let p1 = new Person("Hans", "Wurst");
let p2 = new Person("Max", "Mustermann");
```

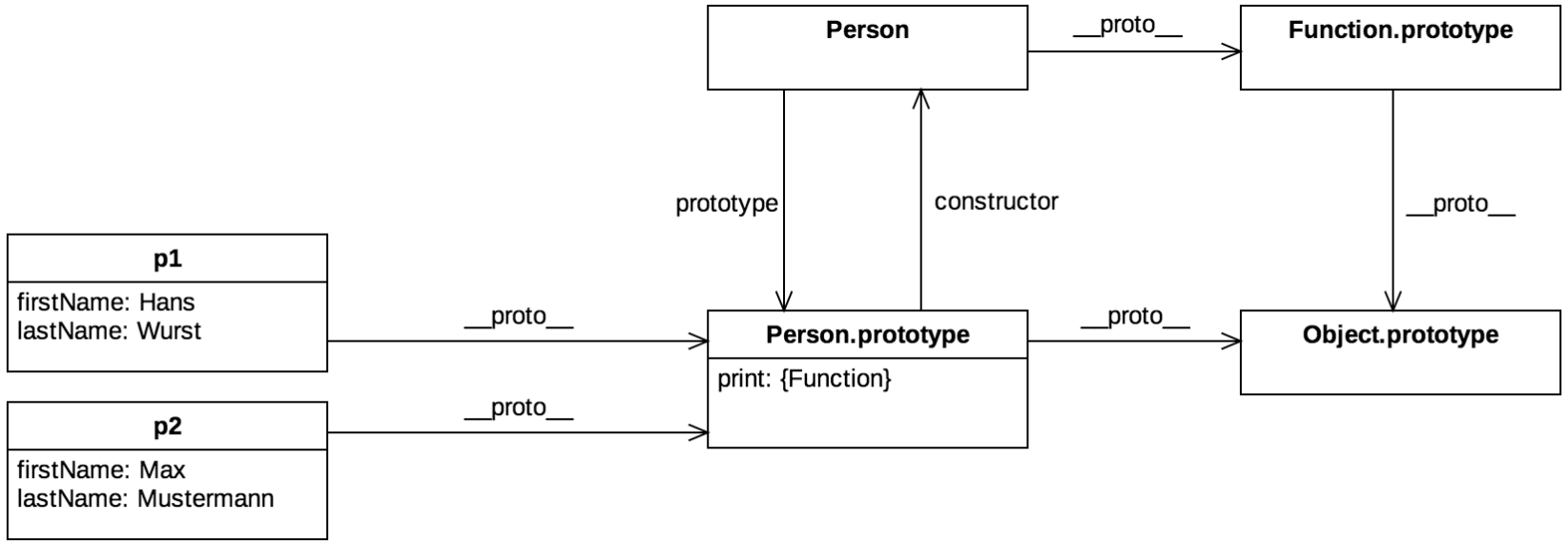
Prototypen (Konstruktorfunktion)



Prototypen (Konstruktorfunktion)

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}

Person.prototype.print = function() {
   console.log("%s %s", this.firstName, this.lastName);
};

let p1 = new Person("Hans", "Wurst");
let p2 = new Person("Max", "Mustermann");

p1.print();         // Hans Wurst
p2.print();         // Max Mustermann
```

Prototypen (Konstruktorfunktion)



Prototypen (Objekt-Literal)

- Bei Objekt-Literalen ist der Prototyp stets das Objekt `Object.prototype`

```
let p1 = {
   firstName: "Max",
   lastName: "Mustermann“
};

console.log(Object.getPrototypeOf(p1)
      === Object.prototype);  // true
```

Prototypen (Klassensyntax)

- Klasse (hier: `Person`) ist eine implizit erzeugte Konstruktorfunktion
- Implizit erzeugter Prototyp (hier: `Person.prototype`)

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   print() {
      console.log("%s %s", this.firstName, this.lastName);
   }
}

console.log(typeof Person);                      // function
console.log(Person.prototype);                    // Person {}

let p1 = new Person('Hans', 'Wurst');
console.log(Object.getPrototypeOf(p1));           // Person {}
console.log(Object.getPrototypeOf(p1) === Person.prototype);
                                                  // true
console.log(Person.prototype.constructor === Person);
                                                  // true
```

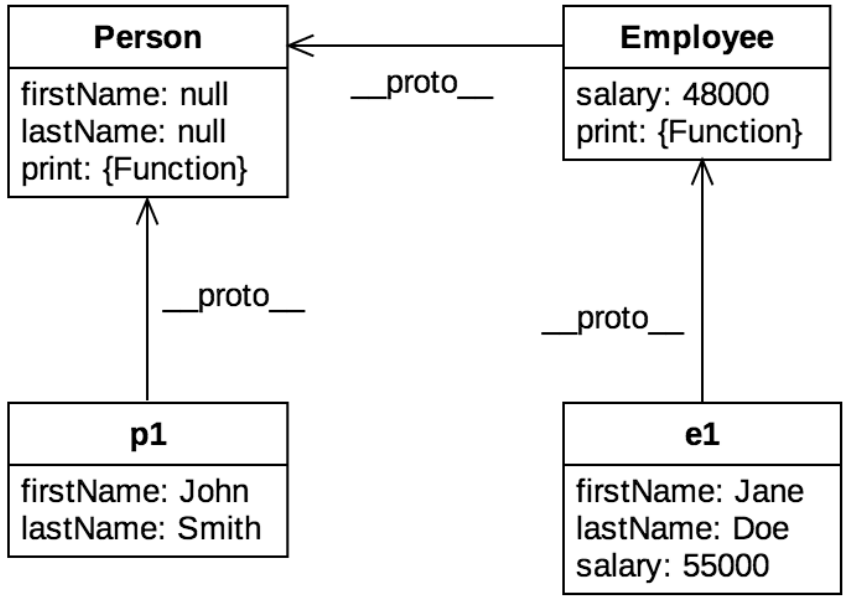
Vererbung

Prototypische Vererbung

```
let Person = {
   firstName: null,
   lastName: null,
   print: function () {
      console.log("%s %s",
         this.firstName, this.lastName
      );
   }
};

let p1 = Object.create(Person);
p1.firstName = "John";
p1.lastName = "Smith";
p1.print();
```

```
// inherited object
let Employee = Object.create(Person);
// new property with default value
Employee.salary = 48000;
// override method
Employee.print = function () {
   // calling overridden method
   Person.print.call(this);
   console.log("Salary: %s", this.salary);
};

let e1 = Object.create(Employee);
e1.firstName = "Jane";
e1.lastName = "Doe";
e1.salary = 55000;
e1.print();
```

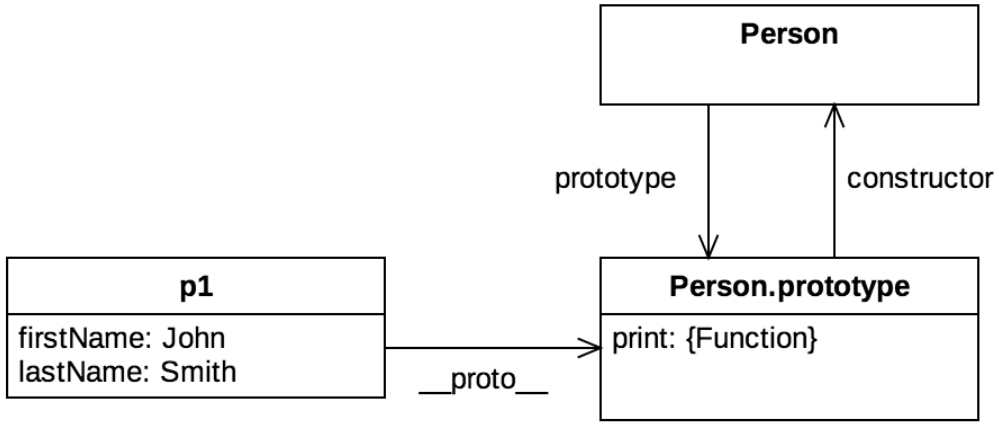
Prototypische Vererbung



Pseudoklassische Vererbung (Konstruktorfunktion)

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}

Person.prototype.print = function () {
   console.log("%s %s", this.firstName, this.lastName);
};

let p1 = new Person("John", "Smith");
p1.print();
```

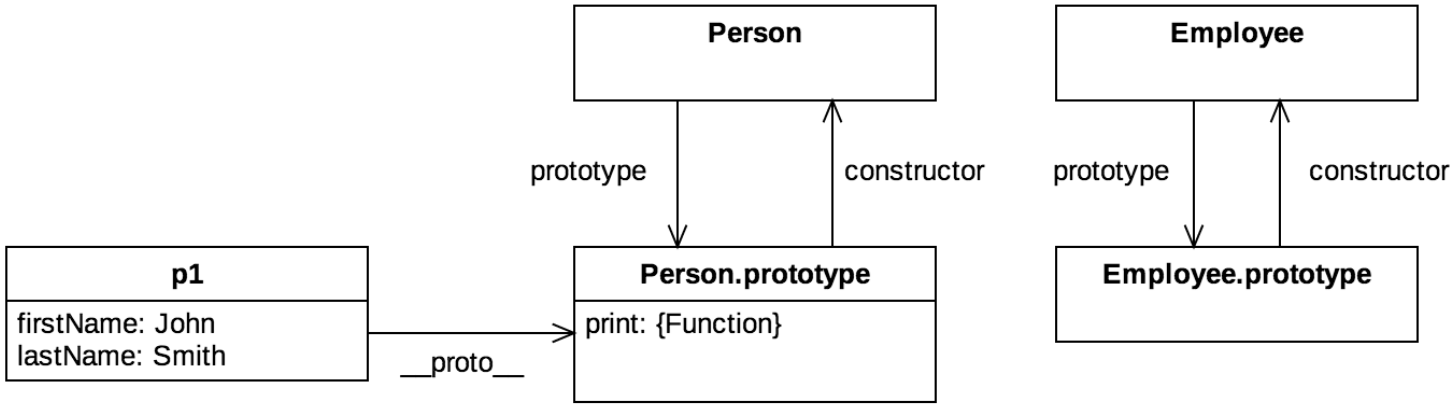
Pseudoklassische Vererbung (Konstruktorfunktion)



Pseudoklassische Vererbung (Konstruktorfunktion)

```
function Employee(firstName, lastName, salary) {
   // call superclass constructor
   Person.call(this, firstName, lastName);
}
```

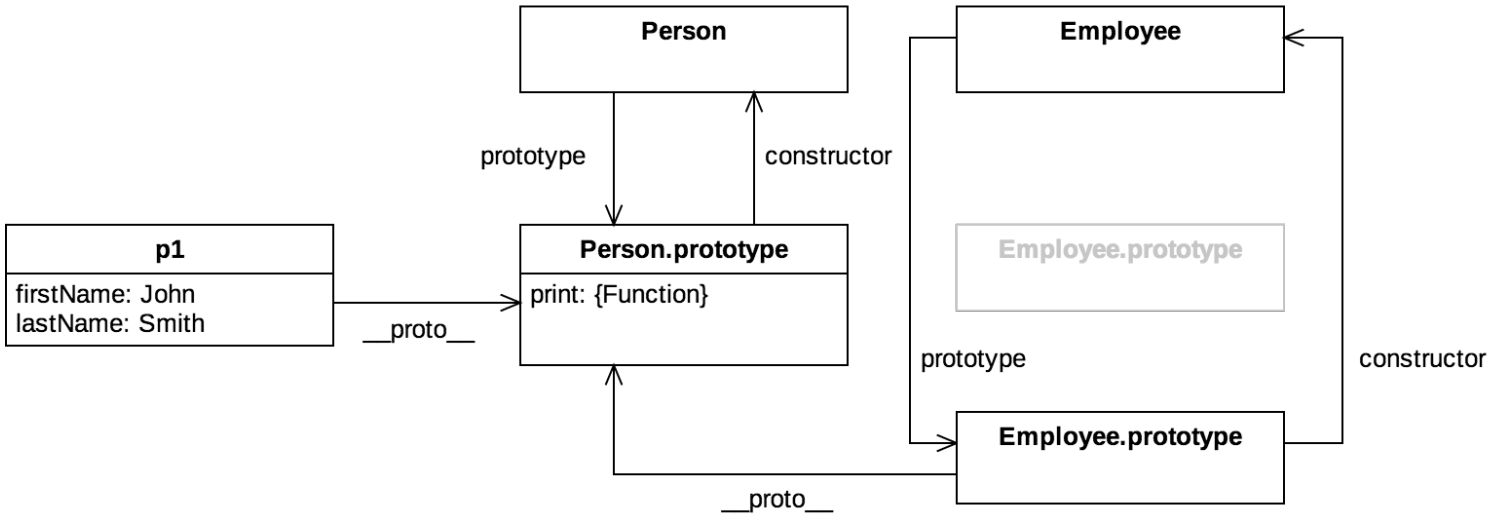
Pseudoklassische Vererbung (Konstruktorfunktion)



Pseudoklassische Vererbung (Konstruktorfunktion)

```
function Employee(firstName, lastName, salary) {
   // call superclass constructor
   Person.call(this, firstName, lastName);
}

// overwrite implicit created prototype
Employee.prototype = Object.create(Person.prototype);
// link prototype
Employee.prototype.constructor = Employee;
```

Pseudoklassische Vererbung (Konstruktorfunktion)

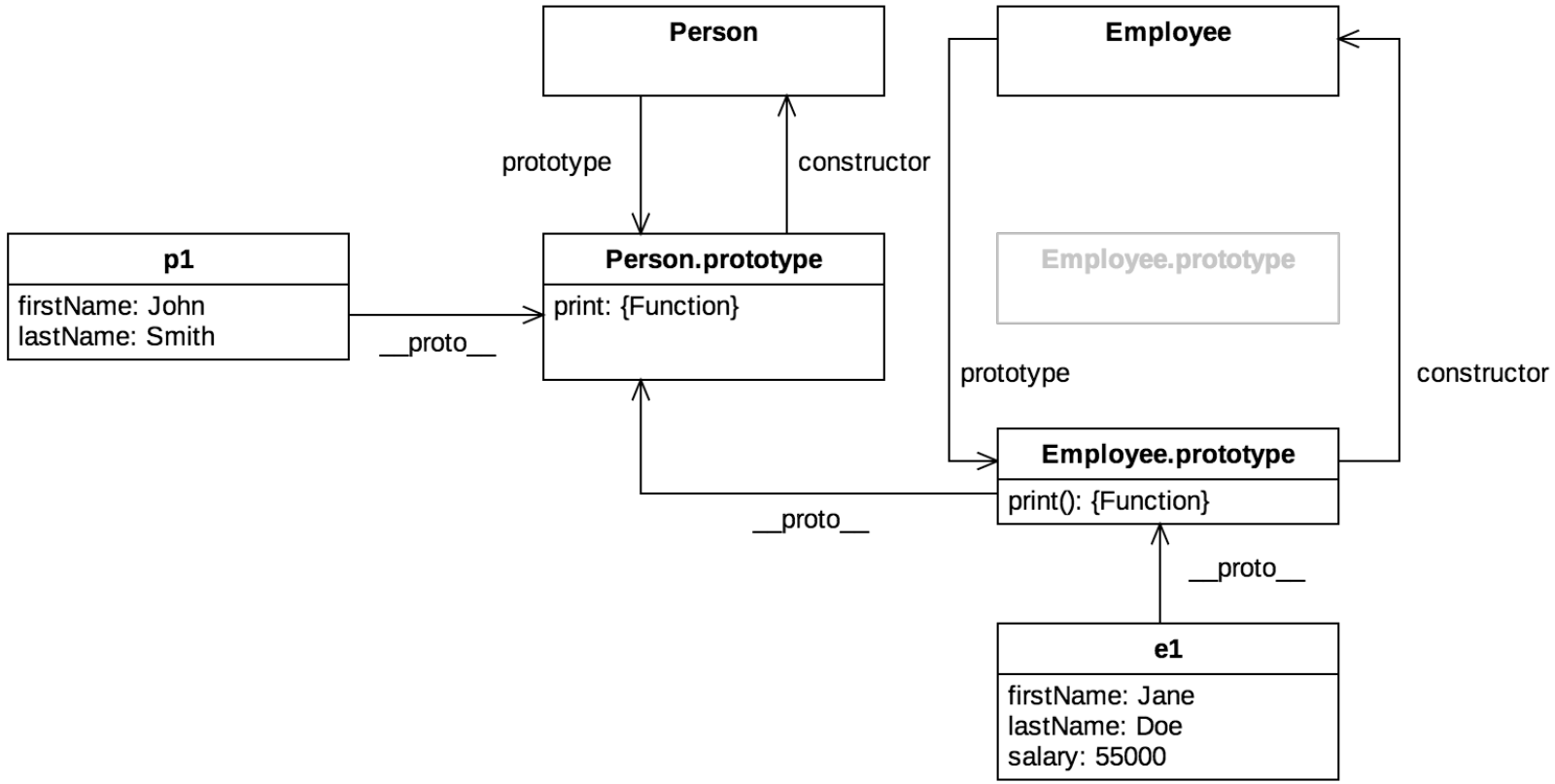


Pseudoklassische Vererbung (Konstruktorfunktion)

```
function Employee(firstName, lastName, salary) {
   // call superclass constructor
   Person.call(this, firstName, lastName);
   // new property with default value
   this.salary = salary || 48000;
}

// override implicit created prototype
Employee.prototype = Object.create(Person.prototype);

// link prototype
Employee.prototype.constructor = Employee;
```

```
// override method
Employee.prototype.print = function () {
   // calling overriden method
   Person.prototype.print.call(this);
   console.log("Salary: %s", this.salary);
};

let e1 = new Employee("Jane", "Doe", 55000);
e1.print();
```

Pseudoklassische Vererbung (Konstruktorfunktion)



Vererbung (Klassensyntax)

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   print() {
      console.log("%s %s", this.firstName,
            this.lastName);
   }
}

class Employee extends Person {
   constructor(firstName, lastName, salary = 48000) {
      // call superclass constructor
      super(firstName, lastName);
      this.salary = salary;             // new property
   }

   print() {               // override method
      super.print();       // calling overriden method
      console.log("Salary: %s", this.salary);
   }
}
```

```
let p1 = new Person("John", "Smith");
p1.print();

let e1 = new Employee("Jane", "Doe", 55000);
e1.print();

class Animal {}

console.log(e1 instanceof Employee);      // true
console.log(e1 instanceof Person);        // true
console.log(e1 instanceof Animal);        // false
```

Getter/Setter und statische Eigenschaften

Getter und Setter

- Sprachkonstrukt zur Definition von Getter-/Setter-Methoden
- Schlüsselwörter `get` bzw. `set` vor Funktionsdefinitionen
- Lesender und schreibender Zugriff auf Eigenschaften wird in Funktionen gekapselt
  - Alternative zu normalen Methoden mit Bezeichnerpräfix "`get`" bzw. "`set`"
- Aber: Verwendung wie eine Dateneigenschaft, nicht wie eine Methode
  - Zugrundeliegende Eigenschaft muss ggf. anderen Namen tragen – sonst Endlosrekursion!

Getter und Setter

```
class Person {
     constructor(firstName, lastName) {
        this._firstName = firstName;
        this._lastName = lastName;
     }

     get firstName() { return this._firstName; }

     set firstName(value) {
        if (value.length !== 0) { this._firstName = value; }
     }

     get lastName() { return this._lastName; }

     set lastName(value) {
        if (value.length !== 0) { this._lastName = value; }
     }

     get _fullName() {
       return `${this.firstName} ${this.lastName}`;
     }

     print () {
       console.log(this._fullName);
     }
}

const p = new Person('Hans', 'Wurst');
p.firstName = 'John';
p.print(); // John Wurst
```

Statische Eigenschaften

- Es gab vor ES6 kein spezielles Sprachkonstrukt für statische Eigenschaften und Methoden
- Emulation über Eigenschaften der Konstruktorfunktion
- Gängige Konvention für statische Eigenschaften (seltener für statische Methoden): Nur Großbuchstaben und Unterstriche
- ES6: Schlüsselwort `static`

Statische Eigenschaften (vor ES6)

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
   Person.COUNT++;            // instance counter
}

// initialize "static" property as a property
// of a constructor function
Person.COUNT = 0;

// "static" method as a property of a constructor function
Person.printCount = function () {
   console.log("Number of Persons: %d", Person.COUNT);
};
```

```
new Person(…);
new Person(…);
new Person(…);
new Person(…);
new Employee(…);

Person.printCount();
```

Statische Eigenschaften (ES6: static)

```
class Person {
   static COUNT = 0;
   static printCount () { console.log('Number of Persons: %d', Person.COUNT); }
   static { console.log(`Person.COUNT: ${this.COUNT}`); }

   constructor(firstName, lastName) {
      this._firstName = firstName;
      this._lastName = lastName;

      Person.COUNT++;
   }

   print() { console.log("%s %s", this._firstName, this._lastName); }
}

class Employee extends Person { ... }

let p1 = new Person("John", "Smith");
let e1 = new Employee("Jane", "Doe", 55000);
Person.printCount();            // ?
```

Datenkapselung

Private Eigenschaften

- Es gab ursprünglich kein spezielles Sprachkonstrukt für private Eigenschaften und Methoden
- Bei allen bisherigen Beispielen können "von außen" die Eigenschaftswerte geändert oder die Eigenschaft gelöscht werden
- ES13+: Bezeichnerpräfix #

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}

Person.prototype.print = function () { ... };

let p1 = new Person('Hans', 'Wurst');
p1.print();         // Hans Wurst
p1.firstName = 'John';
p1.print();         // John Wurst
```

Quasi-Private Eigenschaften per Konvention

- Implementierung von Getter-/ Setter für alle Eigenschaften, die gelesen bzw. geschrieben werden können
  - Idee: Zustandsänderung sollte nur über Methoden erfolgen, niemals über direkten Zugriff auf eine Eigenschaft
- Gängig: Präfix "_" oder "__" für quasi-private Eigenschaften und Methoden

```
class Person {
  constructor (firstName, lastName) {
    this._firstName = firstName;
    this._lastName = lastName;
  }

  get firstName () { return this._firstName; }

  set firstName (value) {
    if (value.length !== 0) { this._firstName = value; }
  }

  // Weitere Getter/Setter analog

   print () {
    console.log(this._fullName);
  }
}
```

Quasi-Private Eigenschaften per Konvention

- Lediglich eine Absichtsbekundung durch den Entwickler
- Verhindert die Modifikation von Eigenschaften keineswegs

```
let p1 = new Person('Hans', 'Wurst');
p1.print();         // Hans Wurst
p1.firstName = 'John‘;
p1.print();         // John Wurst
p1._lastName = 'Doe';
p1.print();         // John Doe
```

Private Eigenschaften per Function Level Scope

- Jede Funktion spannt einen eigenen Gültigkeitsbereich auf – auch Konstruktorfunktionen
  - Lokal deklarierte Variablen und Funktionen sind außerhalb des Konstruktors nicht sichtbar (also privat)
- Verwendung von sog. privilegierten Methoden anstelle von öffentlichen Methoden
  - Zugriff auf private sowie öffentliche Eigenschaften und Methoden
  - **Aber: Redundante Erzeugung eines Funktionsobjekts für jede Instanz**
    - **Gilt auch für die ES13+-Syntax (#)**
- https://www.crockford.com/javascript/private.html

Private Eigenschaften per Function Level Scope

```
function Counter () {
  let seed = Math.floor(Math.random() * 100);  // private property

  this.increments = 0;  // public property

  function increment () { this.increments++; }  // private method

  // privileged method - access to public and private properties
  // and methods but redundant in every object
  this.getNextValue = function () {
    increment.call(this);
    return seed + this.increments;
  };
}

// public method - no access to private properties or methods
// but no redundandy
Counter.prototype.toString = function () {
  return this.increments.toString();
};
```

```
let counter = new Counter();

console.log(counter.seed);  // undefined

console.log(counter.increments);  // 0
console.log(counter.toString());  // 0

console.log(counter.increment)  // undefined

console.log(counter.getNextValue());  // 8
console.log(counter.getNextValue());  // 9
console.log(counter.getNextValue());  // 10
console.log(counter.toString());  // 3
```

Private Eigenschaften

- Langersehntes Sprachkonstrukt zur einfachen Realisierung privater Eigenschaften und Methoden
  - Einführung erst mit ES13 (2022)
- Präfix # für Eigenschaftsnamen
  - Der Präfix ist Teil des Namens, muss also bei jeder Referenzierung angegeben werden
- Private Dateneigenschaft (nicht: Methode) muss vor der Initialisierung im Konstruktor deklariert werden
  - Unmittelbare Initialisierung möglich, sonst `undefined`
- Eigenschaften sind nur in Methoden des deklarierenden Objekts sichtbar
  - Auch nicht in davon abgeleiteten Objekten (also nicht „protected“)
  - `SyntaxError` beim Versuch des Zugriffs
- Kann auch mit statischen Eigenschaften und Methoden kombiniert werden

Private Eigenschaften

```
class Person {
  #firstName;
  #lastName;

  constructor (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get firstName () { return this.#firstName; }

  set firstName (value) {
    if (value.length !== 0) { this.#firstName = value; }
  }

  get lastName () { return this.#lastName; }

  set lastName (value) {
    if (value.length !== 0) { this.#lastName = value; }
  }

  get #fullName () {
    return `${this.firstName} ${this.lastName}`;
  }

  print () { console.log(this.#fullName); }
}

const p1 = new Person('Hans', 'Wurst');
// p1.#fullName();  // SyntaxError
p1.print(); // Hans Wurst
// p1.#lastName = 'Doe';  // SyntaxError
p1.lastName = 'Doe';
p1.print(); // Hans Doe
```

Vermeidung von Namenskonflikten

- Bei der Realisierung umfangreicher Projekte oder Bibliotheken mit vielen Prototypen gilt es, Namenskonflikte zu vermeiden
  - Prototypen sind in Browser-Anwendungen globale Variablen, also Eigenschaften des *global object*
  - Java u.ä.: Packages
  - Vor ES6: Kein entsprechendes Sprachkonstrukt
- Realisierung von Modulen durch
  - Revealing Module-Entwurfsmuster
  - De-facto-Standards zur Modulspezifikationen
    - Asynchronous Module Definition (AMD)
    - Universal Module Definition (UMD)
    - CommonJS-Module
  - Native ES-Module (ESM)
    - Einführung mit ES6
    - Unterstützung in Node.js seit Version 13 (2020)

Immediately Invoked Function Expression (IIFE)

- Grundlage vieler Ansätze: Jede Funktion spannt einen eigenen Gültigkeitsbereich auf
- Entwurfsmuster: *Immediately Invoked Function Expression*
- Einmaliger Aufruf einer anonymen Funktion unmittelbar nach deren Definition

```
(function () {
   console.log("Ready.");
})();

(function (name) {
   console.log("Hello, %s!", name);
})("World");
```

Revealing Module-Entwurfsmuster

- Zusammenfassung der Klassen in einer IIFE
- IIFE liefert assoziatives Array mit Referenzen auf die inneren Konstruktorfunktion(en)
- Ablegen dieses Objekts als Eigenschaft mit möglichst eindeutigem Schlüssel (hier: `MyModule`) im *global object*
- Oder-Verknüpfung verhindert das Überschreiben eines bereits existierenden gleichnamigen Moduls

```
let MyModule = MyModule || (function () {
   class Person { … }

   class Employee extends Person { … }

   return { Person, Employee };
})();

let p = new MyModule.Person("Hans", "Wurst");
let e = new MyModule.Employee("Jane", "Doe", 55000);
```

De-facto-Standards

- Module-Entwurfsmuster wurde im Detail unterschiedlich umgesetzt
- Wunsch nach (De-facto-)Standardisierung
  - Asynchronous Module Definition (AMD), Universal Module Definition (UMD)
    - Keine native Browser-Unterstützung
  - CommonJS-Module
    - Keine native Browser-Unterstützung
    - Setzt Implementierung der Funktion `require()` sowie das vordefinierte Objekt `module` (mit Eigenschaft `exports`) voraus, z.B. durch die Laufzeitumgebung Node.js
    - Kann über den sog. Bundling-Mechanismus (z.B. mithilfe von esbuild, browserify oder webpack) auch im Browser genutzt werden

CommonJS-Module

- Wert der Eigenschaft `exports` des vordefinierten Objekts `module` wird beim Laden des Moduls zurück gegeben
- Bei Verwendung von Node.js spannt jedes Modul einen eigenen Gültigkeitsbereich auf
- Bei Verwendung eines Bundlers für den Einsatz im Browser übernimmt dieser das Schachteln der Module in IIFEs zur Erstellung eines jeweils eigenen Gültigkeitsbereichs

```
class Person { … }

class Employee extends Person { … }

module.exports.Person = Person;
module.exports.Employee = Employee;

// alternativ:
// module.exports = { Person, Employee };
```

CommonJS-Module

- Globale Funktion `require()` zum Importieren von Modulen
  - Implementiert in der Node.js-Laufzeitumgebung
    - Relative Pfade (ohne Dateiendung)
    - npm-Modulnamen (abgelegt in Ordner `node_modules`)
  - Nicht implementiert in den Browsern
    - Wird vom Bundler in die erzeugte JavaScript-Datei eingefügt

```
const MyCommonJSModule =
      require("./lib/MyCommonJSModule");

let p = new MyCommonJSModule.Person("Hans", "Wurst");
p.print();

let e = new MyCommonJSModule.Employee("Jane", "Doe", 55000);
e.print();

/* Alternativ:
const { Person, Employee } =
      require('./lib/MyCommonJSModule');

let p = new Person('Hans', 'Wurst');
p.print();

let e = new Employee('Jane', 'Doe', 55000);
e.print();
/*
```

Native ES-Module (ESM)

- Native ES-Module sollten dank der Standardisierung zumindest in neuen Projekten bevorzugt werden
  - In Node.js seit Version 14 stabil
  - Für Browser-Anwendungen ggf. Anpassung der Build-Pipeline nötig (siehe nächste Vorlesungseinheit)
    - Bspw. unterstützt der Bundler browserify keine ES-Module
- Neue Schlüsselwörter: `export`, `default`, `as`, `import`
- Empfehlung: Dateien, die ES-Module exportieren, sollten die Endung `*.mjs` (anstatt `*.js`) aufweisen
- Pfadangaben bei Imports
  - Relative Pfade (inkl. Dateiendung)
  - npm-Modulnamen (abgelegt in Ordner `node_modules`)
- Jedes ES-Modul spannt einen eigenen Gültigkeitsbereich auf

Native ES-Module (ESM): Benannte Exports

```
export const MAX_INPUT_LENGTH = 255;

export function removeWhitespace (str) {
   return str.replace(/\s/g, "");
}

export function removeChildNodes (node) {
   if (node) {
      while (node.hasChildNodes()) {
         try {
            node.removeChild(node.firstChild);
         }
         catch (error) { // intentionally left blank }
      }
   }
}

export class Person { … }

export class Employee extends Person { … }
```

```
export const MAX_INPUT_LENGTH = 255;

export function removeWhitespace (str) {
   return str.replace(/\s/g, "");
}

export function removeChildNodes (node) {
   if (node) {
      while (node.hasChildNodes()) {
         try {
            node.removeChild(node.firstChild);
         }
         catch (error) { // intentionally left blank }
      }
   }
}

class Person { … }

class Employee extends Person { … }

export { Person, Employee };
```

Native ES-Module (ESM): Import benannter Exports

```
import { removeWhitespace, Employee } from './NamedExports.mjs';

console.log(removeWhitespace(' abc '));

new Employee('Jane', 'Doe').print();
```

Native ES-Module (ESM): Umbenannte Exports und Imports

```
function removeWhitespace (str) {
  return str.replace(/\s/g, "");
}

export { removeWhitespace as trim };
```

```
import { trim } from './RenamedExports.mjs';
import { trim as removeWhitespace } from './RenamedExports.mjs';

console.log(trim(' abc '));
console.log(removeWhitespace(' abc '));

import * as MyModule from './RenamedExports.mjs';

new MyModule.Employee('Jane', 'Doe').print();
```

Native ES-Module (ESM): Default-Export

```
export default class Person { … }
```

```
import Person from './Person.mjs';

new Person('Hans', 'Wurst').print();
```

Fragen?