Web-Entwicklung

Objektorientierte Programmierung

Inhalte der Vorlesung

- Objektorientierte Programmierung
  - Objekterzeugung
  - Prototypen
  - Vererbung
    - Prototypische Vererbung
    - Pseudoklassische Vererbung
  - Statische Eigenschaften
  - Datenkapselung und Module
    - Private Eigenschaften
    - Module
      - Entwurfsmuster: Immediately Invoked Function Expression
      - CommonJS-Module

Objekterzeugung

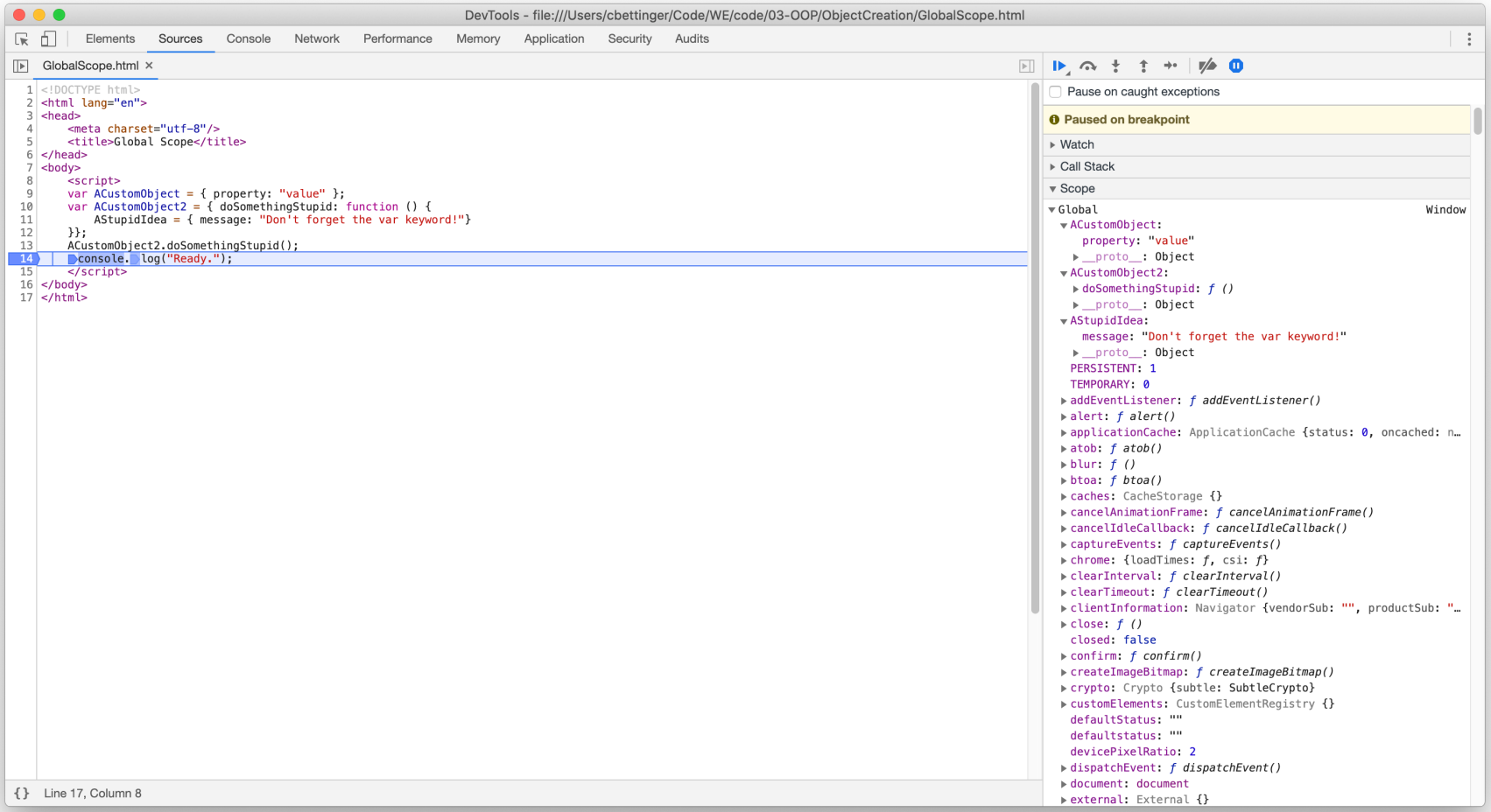
Globaler Gültigkeitsbereich (*global scope*)

- 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`, `window` und `document` im Browser
  - `console` und `process` in Node.js
- Benutzerdefinierte globale Variablen (sollten vermieden werden)
  - Deklaration von Variablen mit Schlüsselwort `var` im äußersten Gültigkeitsbereich einer Browser-Anwendung (d.h. außerhalb jeder Funktion)
  - Zuweisung an Variable ohne vorherige Deklaration mit `var` in beliebigem Gültigkeitsbereich

Globales Objekt (*global object*)

- 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`
- 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

Globaler Gültigkeitsbereich



Objekterzeugung

- Objekt-Literal
- Konstruktorfunktion
- `Object.create()`
- ES6-Klassensyntax (nächste Vorlesungseinheit)

Objekt-Literal

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

```
var 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);
   };
}

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

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

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

Object.create()

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

console.dir(p1);
```

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

console.dir(p2);
```

Object.create()

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

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

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

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

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

Object.create()

```
var 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;
   }
};

var p7 = Object.create(Person).init("Jane", "Doe");
p7.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` (Ausnahme: `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 Protypenkette

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

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

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

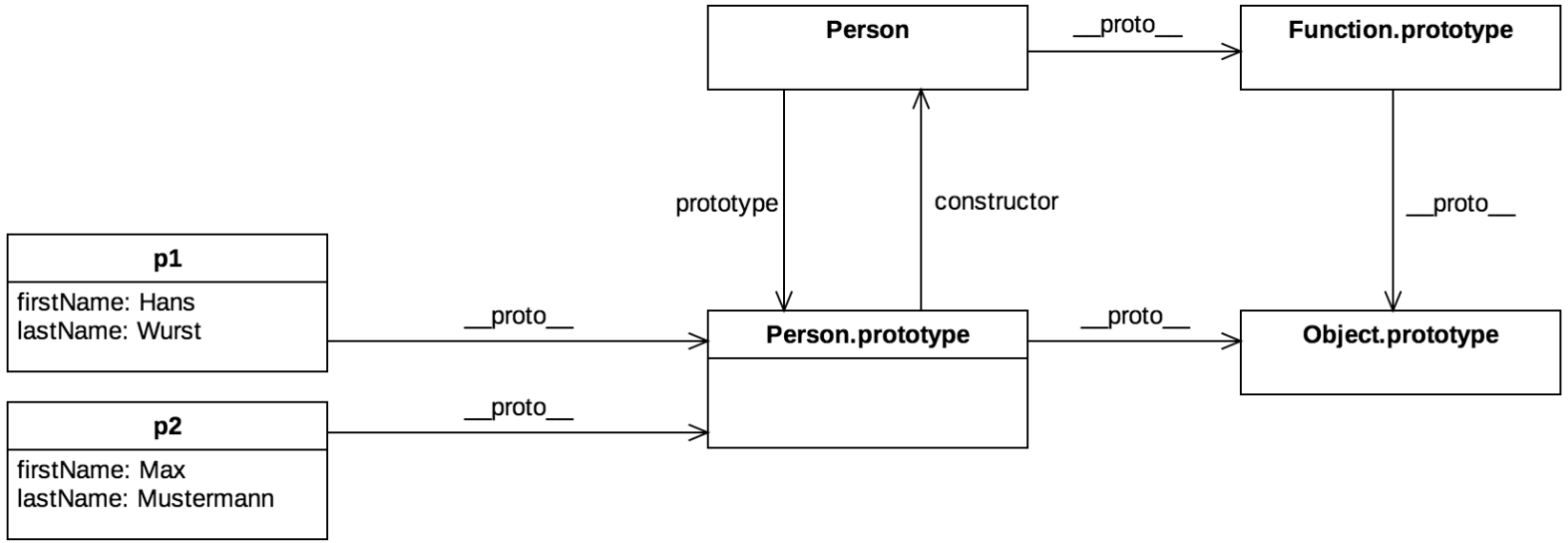
Prototypen (Konstruktorfunktion)

- 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;
}

var p1 = new Person("Hans", "Wurst");
var 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
```

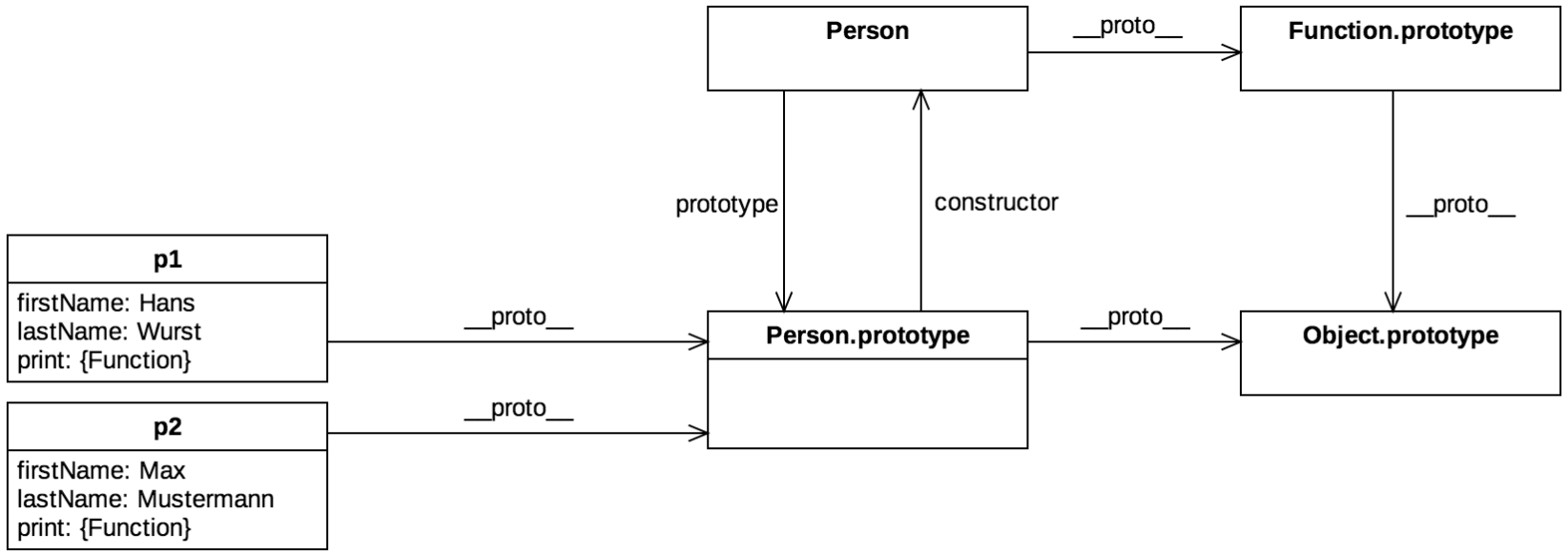
Prototypen (Konstruktorfunktion)



Prototypen (Konstruktorfunktion)

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

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

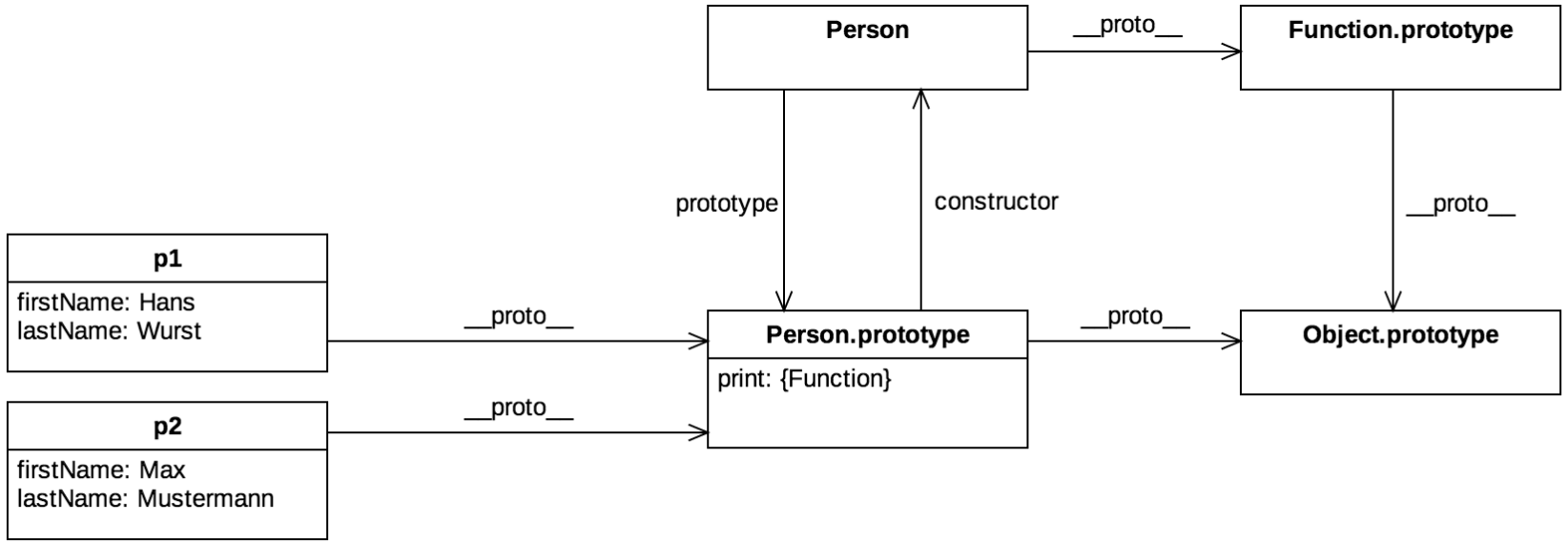
Prototypen (Konstruktorfunktion)



Prototypen (Konstruktorfunktion)

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

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

var p1 = new Person("Hans", "Wurst");
var 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`

```
var p1 = {
   firstName: "Max",
   lastName: "Mustermann“
};
console.log(Object.getPrototypeOf(p1)
      === Object.prototype);  // true
```

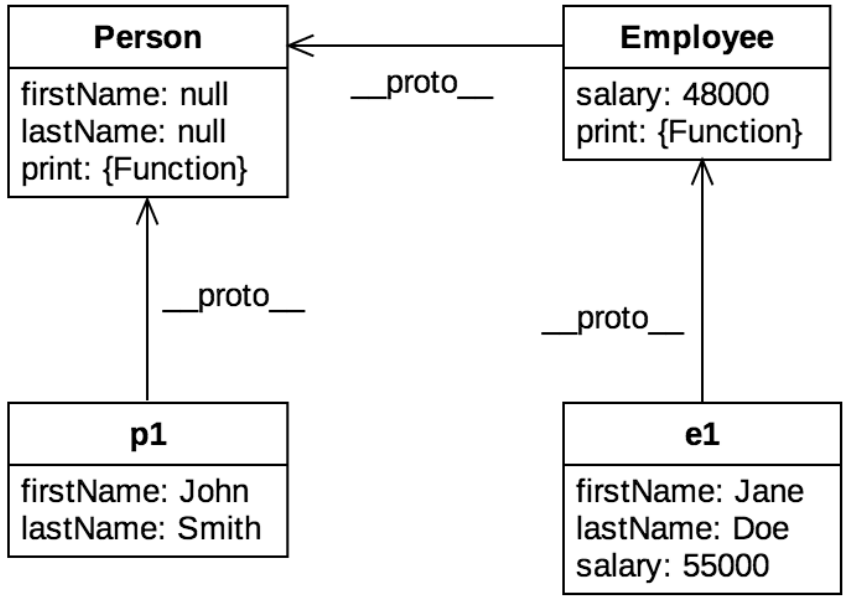
Vererbung

Prototypische Vererbung

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

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

```
// inherited object
var 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);
};

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

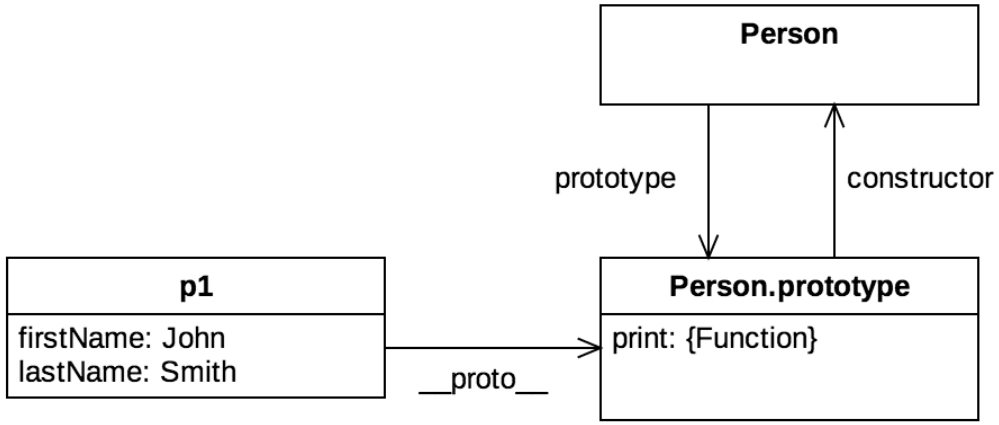
Prototypische Vererbung



Pseudoklassische Vererbung

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

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

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

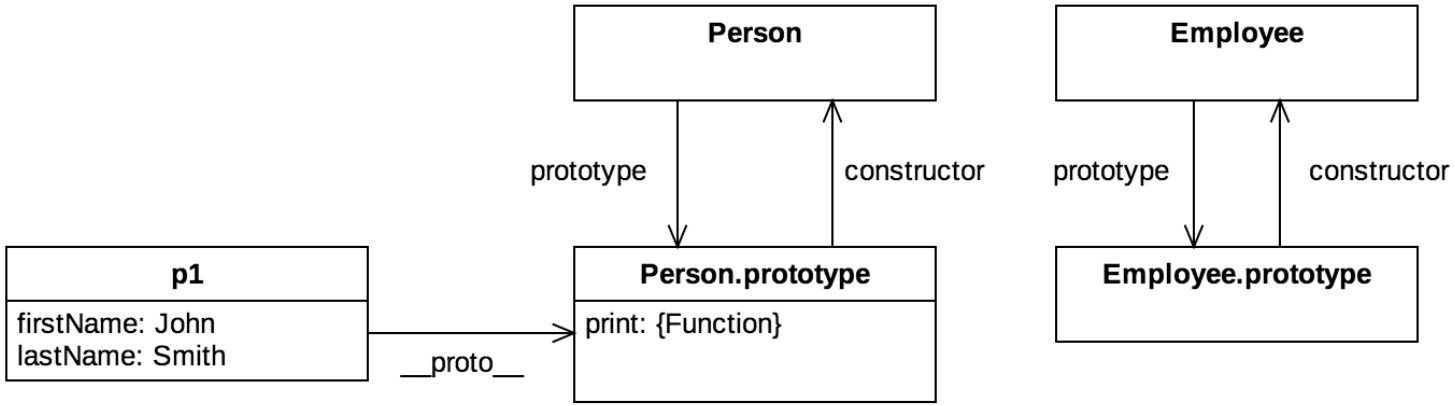
Pseudoklassische Vererbung



Pseudoklassische Vererbung

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

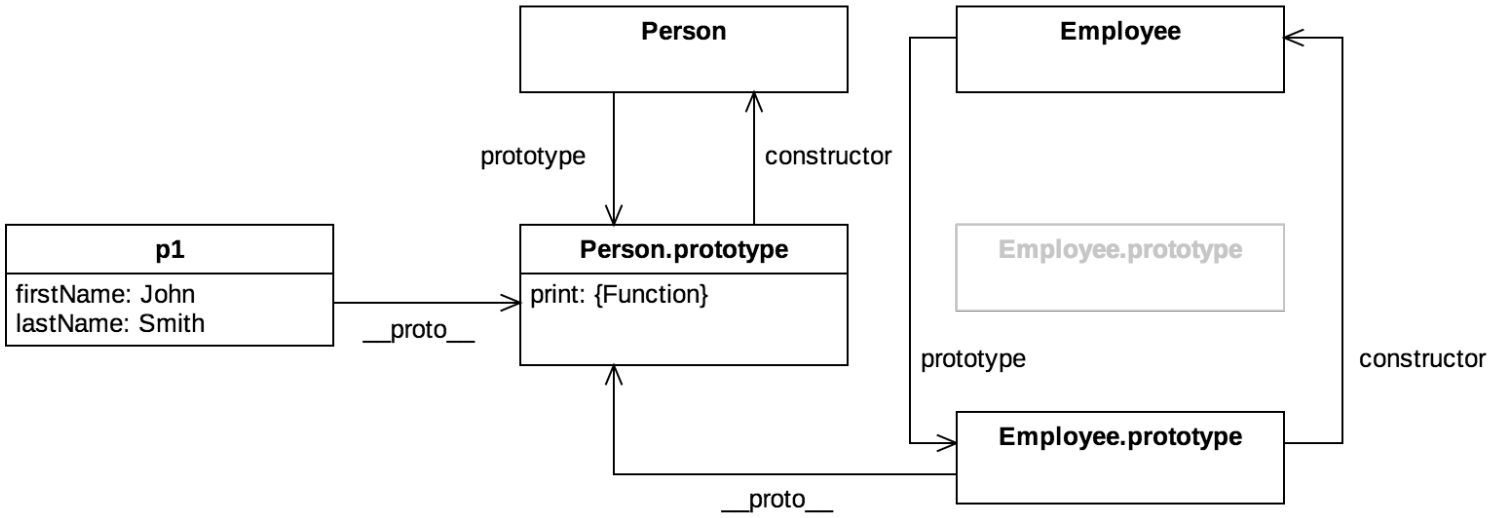
Pseudoklassische Vererbung



Pseudoklassische Vererbung

```
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

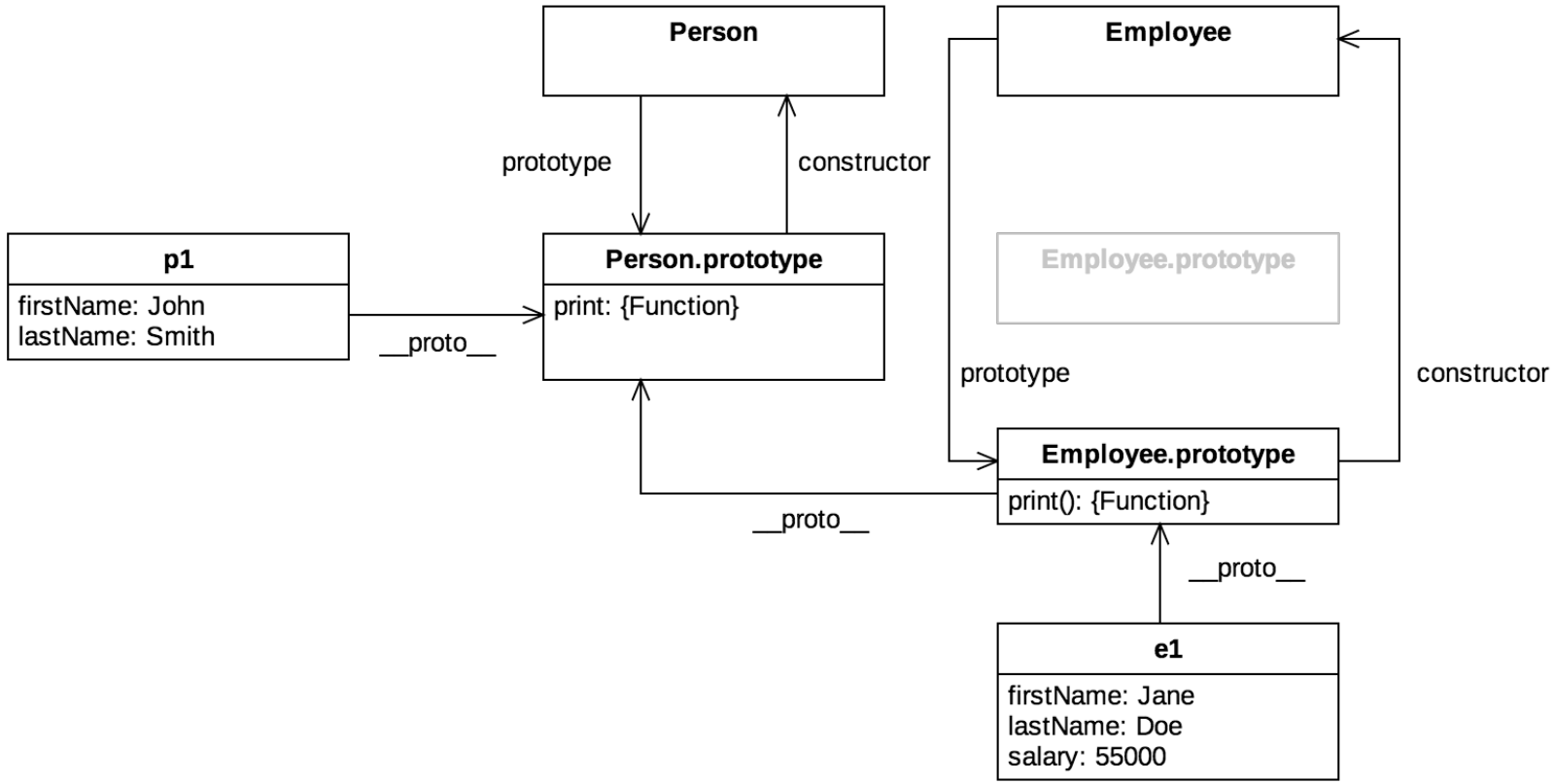


Pseudoklassische Vererbung

```
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);
};

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

Pseudoklassische Vererbung



Statische Eigenschaften

Statische Eigenschaften

- Es gibt kein spezielles Sprachkonstrukt für statische Eigenschaften (inkl. Methoden)
- Emulation über Eigenschaften der Konstruktorfunktion
- ES6: Schlüsselwort `static` für statische Methoden
- Konvention für statische Eigenschaften (nicht statische Methoden): Nur Großbuchstaben und Unterstriche

Statische Eigenschaften

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

// "static" properties as properties of constructor
function
Person.COUNT = 0;

// "static" method as property of 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();
```

Datenkapselung und Module

Private Eigenschaften

- Es gibt kein spezielles Sprachkonstrukt für private Eigenschaften und Methoden
- Bei allen bisherigen Beispielen können "von außen" die Eigenschaftswerte geändert oder die Eigenschaft gelöscht werden

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

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

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

Private Eigenschaften

- In Literatur und Internet finden sich verschiedene Ansätze zur Realisierung privater Eigenschaften
- Alle Ansätze sind mit Nachteilen verbunden
  - „unschön“
  - Erschwertes Debugging
  - Nicht idiomatisch
  - Nicht effektiv
- ECMA-Arbeitsgruppe arbeitet an Sprachfeature: https://github.com/tc39/proposal-class-fields
- Empfehlungen
  - Auf Konvention basierender, nicht-effektiver aber redundanzfreier Ansatz
  - Auf Gültigkeitsbereichen basierender, idiomatischer, effektiver aber redundanzbehafteter Ansatz

Quasi-Private Eigenschaften per Konvention

- Implementierung von Getter-/ Setter-Methoden 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
  - Empfehlung: Präfix nur bei privaten Methoden, aber nicht bei anderen Eigenschaften

```
function Person(firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}
Person.prototype.getFirstName = function () {
   return this.firstName;
};
Person.prototype.getLastName = function () {
   return this.lastName;
};
Person.prototype.setFirstName = function (value) {
    if (value.length) { this.firstName = value; }
};
Person.prototype.setLastName = function (value) {
   if (value.length) { this.lastName = value; }
};
Person.prototype.print = function () {
   console.log(this.__getFullName());
};
Person.prototype.__getFullName = function () {
   return this.firstName + " " + this.lastName;
};
```

Quasi-Private Eigenschaften per Konvention

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

```
var p2 = new Person("Hans", "Wurst");
p2.setFirstName("John");
p2.lastName = "Doe";
p2.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 für jedes Objekt
- https://www.crockford.com/javascript/private.html

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

   // public property
   this.updates = 0;

   // private method
   function update() {
      this.updates++;
   }

   // privileged method
   this.getValue = function () {
       update.call(this);
       return seed + this.updates;
   };
}

// public method
Counter.prototype.toString = function () {
   return this.updates.toString();
};
```

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
  - ES5: 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 ES6-Module
    - Node.js: Bisher experimentelle Unterstützung

Immediately Invoked Function Expression (IIFE)

- Grundlage der drei folgenden 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 Konstruktorfunktion(en) sowie der Prototypmethoden 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

```
var MyModule = MyModule || (function () {
   function Person(firstName, lastName) {…}
   …
   function Employee(firstName, lastName, salary) {…}
   …

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

var p = new MyModule.Person("Hans", "Wurst");
var 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
    - 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 Browserify) auch im Browser genutzt werden
      - Erlaubt einheitliche Erstellung von Modulen für Node.js und Browser
      - Erlaubt Verwendung der umfangreichen npm-Moduldatenbank (https://www.npmjs.com) auch im Browser

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 jeweils einen eigenen Gültigkeitsbereich auf
- Bei Verwendung von Browserify übernimmt ein Werkzeug das Schachteln der Module in IIFEs zur Erstellung eines jeweils neuen Gültigkeitsbereichs

```
function Person(firstName, lastName) {
   …
}
…
function Employee(firstName, lastName, salary) {
   …
}
…
module.exports.Person = Person;
module.exports.Employee = Employee;
```

CommonJS-Module

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

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

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

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

Fragen?