Construction of User Interfaces (SE/ComS 319)

Ali Jannesari

Jinu Susan Kabala

Department of Computer Science

Iowa State University, Spring 2021

JAVASCRIPT CONCEPTS

Outline

- JavaScript Memory Management
- Other concepts (Hoisting, Closures, …)
- Asynchronous operations (Callback, Promises, async & await)
- JavaScript prototype-based inheritance

JAVASCRIPT MEMORY MANAGEMENT

Memory management in JavaScript

- Managed code vs. unmanaged?
  - Java vs. C?
- Memory life cycle
  - Allocate the memory you need
  - Use the allocated memory (read, write)
  - Release the allocated memory when it is not needed anymore

- **Automatic garbage collection in JavaScript**
  - Opposite to low-level memory management primitives like malloc() and free() (e.g. in C/C++ language)

Static memory allocation vs. dynamic memory allocation

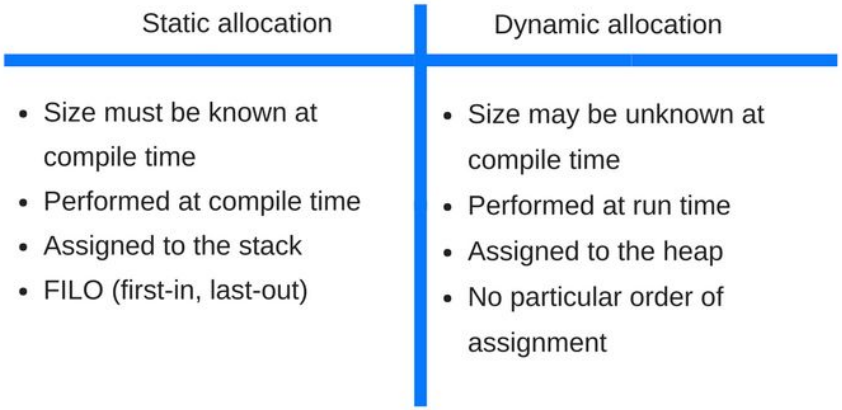
- Static (28 bytes): int n; // 4 bytes

  int x[4]; // array of 4 elements, each 4 bytes

  double m; // 8 bytes

- Dynamic (runtime): int n = readInput(); // reads input from the user

  ... // create an array with "n" elements



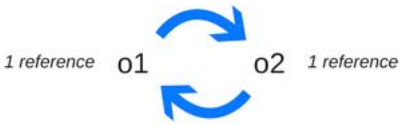
Reference-counting garbage collection (1)

- Reference-counting garbage collection algorithm
  - An object has no other objects referencing it
  - It is considered garbage collectible if there are zero references pointing at this object.
- Problem
  - **Memory leak:**
    - Memory that is not needed by an application anymore that for some reason is not returned to OS or the pool of free memory.

Reference-counting garbage collection (2)

- **Limitation** of Reference-counting garbage collection
  - Cycles (causing memory leak)
  - Example:

```
function f()
{
        var o1 = {};
        var o2 = {};
        o1.a = o2; // o1 references o2
        o2.a = o1; // o2 references o1
        return 'azerty';
}

f();
```



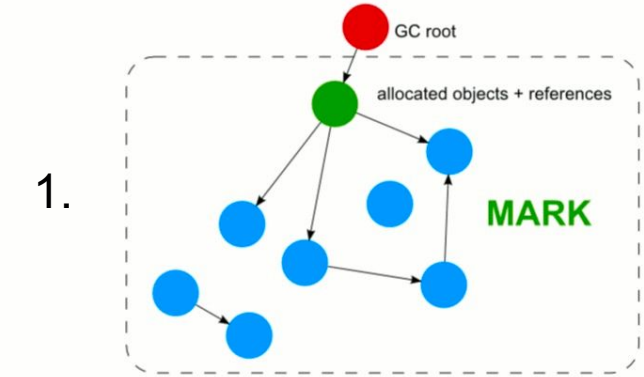
  - Internet Explorer 6 and 7 are known to have reference-counting garbage collectors

Mark-and-sweep algorithm (1)

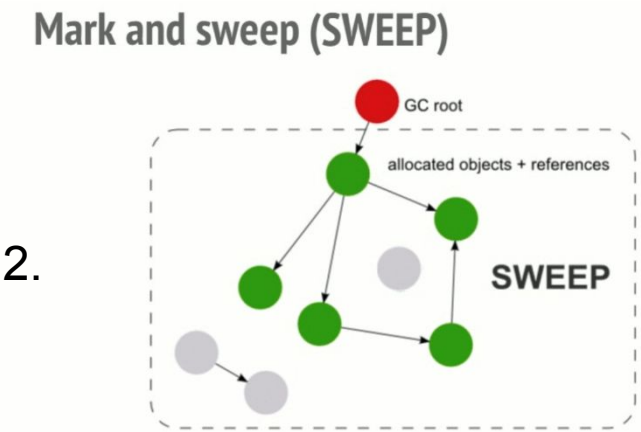
- Mark-and-sweep algorithm
  - an object is unreachable Garbage
  - knowledge of a set of objects called **roots**
    - In JavaScript, the root is the **global object**
  - Periodically, the garbage-collector will start from these roots
    - Finds all objects that are referenced from these roots
    - The garbage collector will find all reachable objects and collect all non-reachable objects.
- This algorithm is **better** than Reference-counting garbage collection
  - Cycles are not a problem
  - In our example, after the function call returns, the 2 objects are not referenced anymore (not reachable from the global object)

Mark-and-sweep algorithm (2)

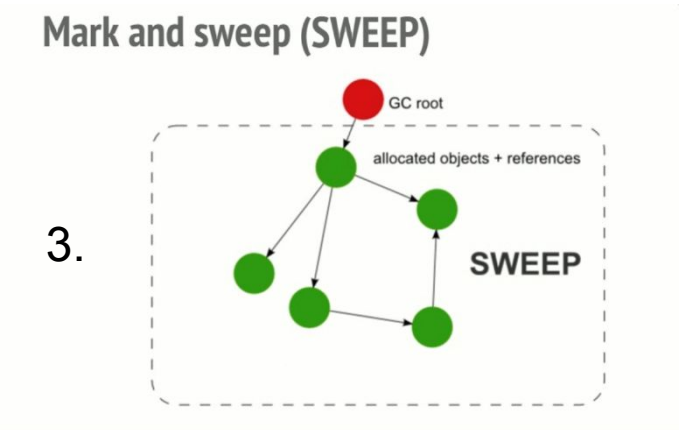
1.



2.



3.



- All modern browsers ship a mark-and-sweep garbage-collector

Source: https://blog.sessionstack.com/

Garbage collection…

- Managed code (e.g. Java/JavaScript code) benefits from automatic garbage collection
  - Comfortable for programmers!
- But with overhead!
  - Compared to C/C++ with manual garbage collection by programmer – no automatic garbage collection! More efficient code!

JAVASCRIPT ASYNCHRONOUS OPERATIONS

Arrow Functions – Recap

- Functions:

```
hello = function() {
  return "Hello World!";
}
```

- With Arrow Functions:

```
hello = () => {
  return "Hello World!";
}
```

- With Arrow Functions (Return Value by Default):

```
hello = () => "Hello World!";
```

- Arrow Function With Parameters:

```
hello = (val) => "Hello " + val;
```

- Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

IIFE (Immediately Invoked Function Expression)

- IIFE: A function is immediately invoked and executed as soon as it is defined
- Variables declared within the IIFE cannot be accessed by the outside world
- Immediately execute privacy

```
> let paintColor= 'red'

  const paint = (() => {
     return {
      changeColorToBlue: () => {
        paintColor: 'Blue';
        return paintColor;
      },
      changeColorToGreen: () => {
        paintColor: 'Green';
        return paintColor;
      }
     }
  })();

  console.log(
    paint.changeColorToBlue()
  );
  red
```

Hoisting – JavaScript Interpreter

- Hoisting: JavaScript interpreter always moves the variables and function declaration to the top of the current scope (function scope / global scope) before the code execution
- Example:

```
> function cowSays(sound){
    console.log(sound);
  }
  cowSays('moo');
  moo
< undefined
```

```
> cowSays('moo');
  function cowSays(sound){
    console.log(sound);
  }
  moo
< undefined
```

- Same output without error!

Closures

- Closures extend behavior such as pass variables, methods, or arrays from an outer function to an inner function
- ‘second()’ extends the behavior of the function ‘first()’ and has access to the variable ‘greet’
- The parent scope won’t have the access of child scope variable ‘name’
- Achieve **object-oriented behavior** through closures
  - **const ‘newFunc’** as an object having property ‘greet’ and ‘second()’ a method

```
> const first = () => {
    const greet= 'Hi';
    const second = () => {
    const name= 'john';
      console.log(greet);
    }
     return second;
  }

  const newFunc= first();
< undefined
> newFunc();
  Hi
< undefined
>
```

Asynchronous operations in JavaScript

- Functions running in parallel with other functions are called asynchronous:
- Callback
- Promises
- async & await

Callbacks

- Callback: a function that is passed to another function as a parameter (invoked or executed inside the other function)

```
> const greeting = (name) => {
    console.log('Hello ' + name);
  }

  const processUserName= (callback) => {
      name = 'GeeksforGeeks';
      callback(name);
  }
  processUserName(greeting);
  Hello GeeksforGeeks
```

- 'greeting' passed as an argument (callback) to the 'processUserName' function.
- Before the 'greeting' function executed it waits for the event 'processUserName' to execute first.

- A function needs to wait for another function to execute or return value
- This makes the chain of the functionalities

Callbacks – Example

Without callback:

```
function myDisplayer(some) {

document.getElementById("demo").inn
erHTML = some;
}

function myCalculator(num1, num2) {
  let sum = num1 + num2;
  myDisplayer(sum);
}

myCalculator(5, 5);
```

With callback:

```
function myDisplayer(some) {

document.getElementById("demo").inn
erHTML = some;
}

function myCalculator(num1, num2,
myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

Promises

- **Promises** avoid recursive structure of callback – ‘callback hell’
- A promise is in three possible states:
  - Fulfilled: When the operation is completed successfully.
  - Rejected: When the operation is failed.
  - Pending: initial state, neither fulfilled nor rejected.
- Chaining operations with promise

```
> const promise = new Promise((resolve, reject) => {
  isNameExist = true;
  if(isNameExist) {
       resolve("User name exist")
     } else {
       reject ("error")
     }
  })


  promise.then(result => console.log(result))
  .catch(()=> {
    console.log('error !')
  })
  User name exist
< ▶ Promise {<resolved>: undefined}
> |
```

- *A Promise is an object representing the eventual completion or failure of an asynchronous operation*.
- *A promise is a returned object to which you attach callbacks*
- Promise Object arguments are two function resolve and reject

Promise – Syntax

- "Producing code" is code that can take some time
- "Consuming code" is code that must wait for the result
- A Promise is a JavaScript object that links producing code and consuming code:

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Async & Await

- Async & Await: provide a way to maintain asynchronous operation more synchronously
  - Example: where you want the data to fully load before pushing it to the view
- Increases the code readability and syntactic improvement

```
> const showPosts = async () => {
   const response = await fetch('https://jsonplaceholder.typicode.com/posts');
   const posts = await response.json();
   console.log(posts);
  }

  showPosts();
  ▶ Promise {<pending>}
```

- wrap ‘await’ inside an ‘async’ function (notify JS that we are working with promises)
- (a)wait for two things: response and posts.
- We need to make sure we have the response fetched (before we can convert the response to JSON format)

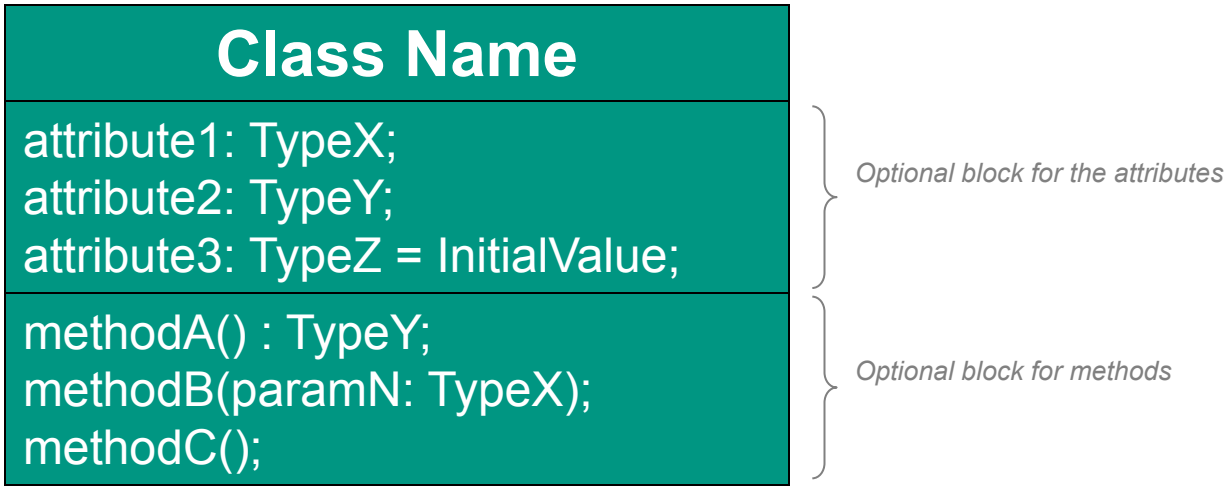
Async & Await – Syntax

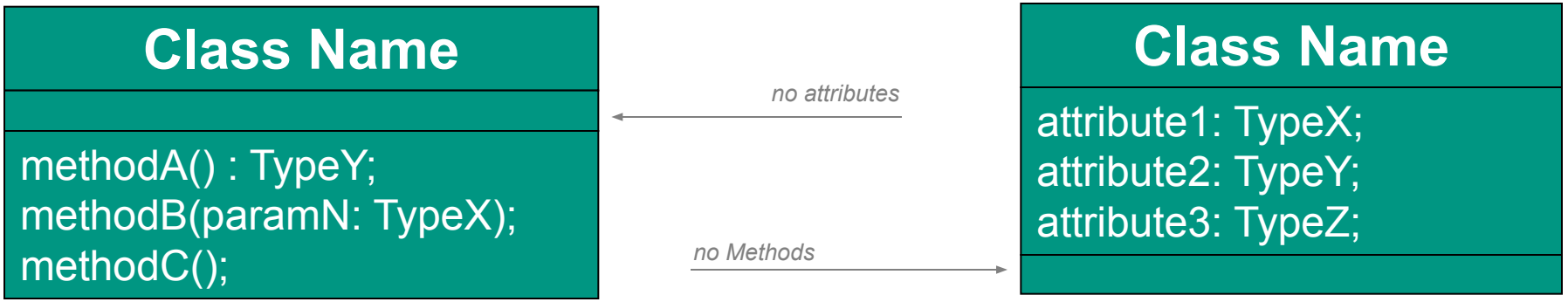
- **async** before a function makes the function return a promise
- **await** before a function makes the function wait for a promise

```
async function myDisplay() {
  let myPromise = new Promise(function(myResolve, myReject) {
    myResolve("I love You !!");
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
```

JAVASCRIPT PROTOTYPE-BASED INHERITANCE

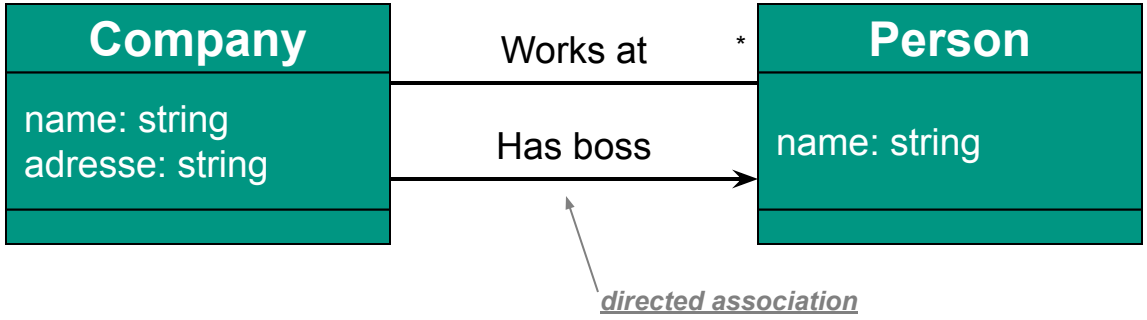
Classes – UML class diagram



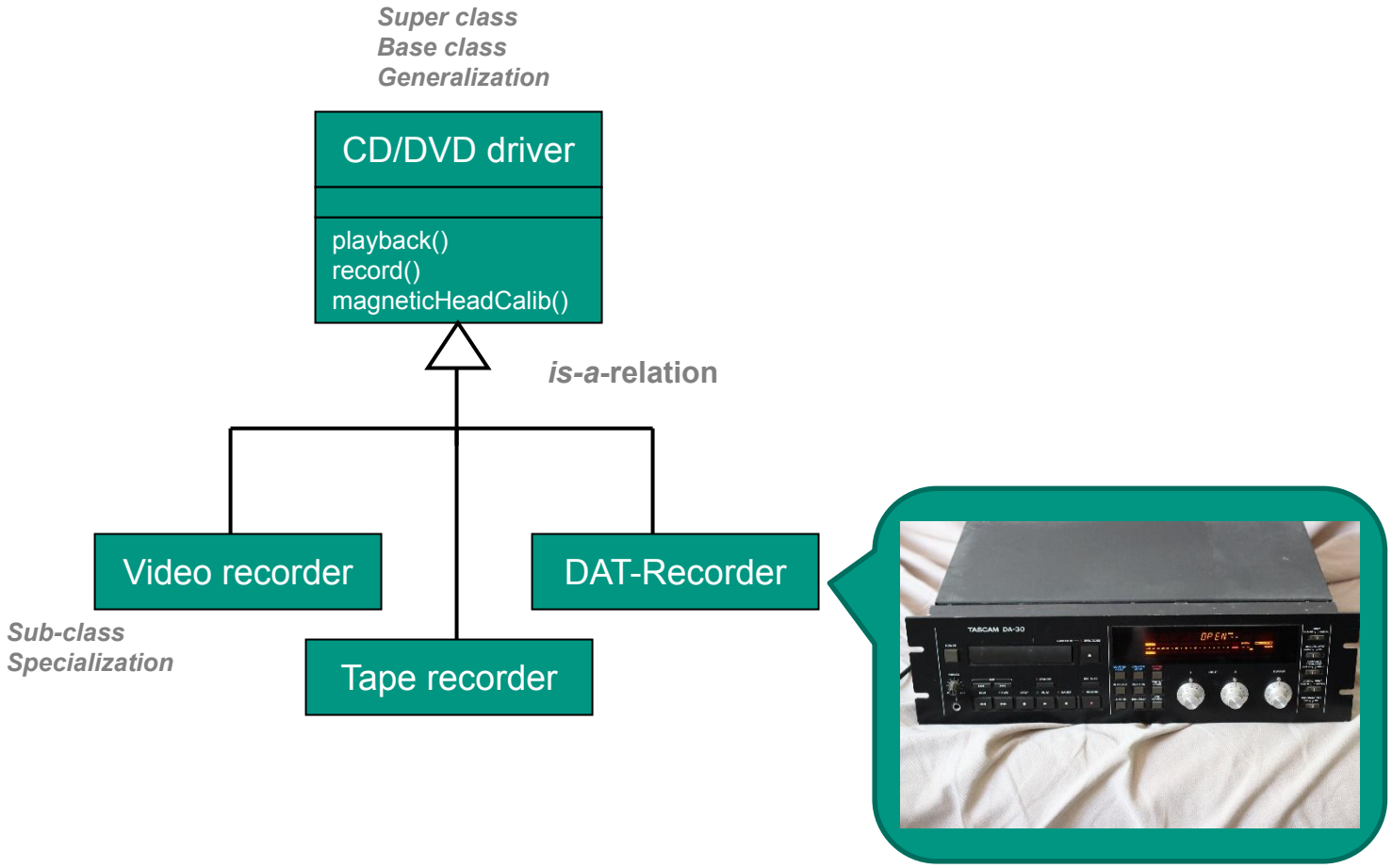


Class diagram – Example

- Describes the types of objects in the system
- Describes the static relationships among them

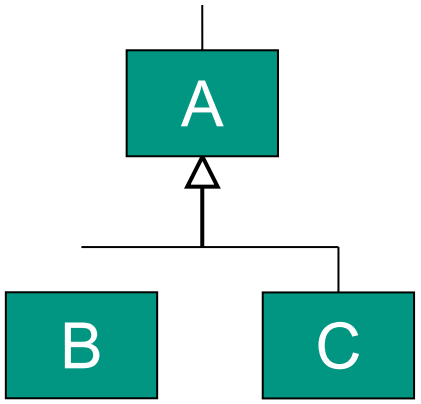


Inheritance

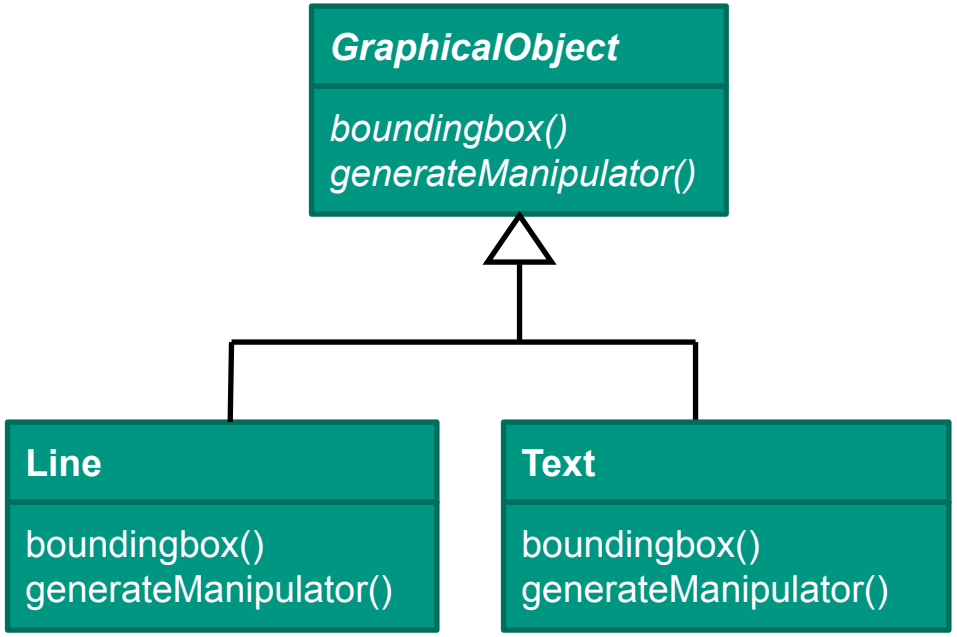


Inheritance – “is-a” relationship

- Let A and B be classes, and ΩA and ΩB the set of objects that make up classes A and B.
  - Then B is a subclass / specialization of A (or A is a superclass / generalization of B) if: ΩB ⊆ ΩA.
- It is also said that **B inherits from A**.
- Since each instance of B is also an instance of A, the relationship between A and B is called the "**is-a**" **relationship**.
- If A has several subclasses, these subclasses should usually be disjoint.



Inheritance – Example



Prototype-based inheritance (1)

- Javascript is different from traditional object-oriented languages in that it uses **prototype inheritance**.
- In a nutshell, prototype inheritance in Javascript works like this:

1. An object has a number of properties. This includes any attributes or functions (methods).
2. An object has a **special parent property**, this is also called the **prototype of the object** (**__proto__**). An object inherits all the properties of its parent.

Prototype-based inheritance (2)

3. An object can override a property of its parent by setting the property on itself.
4. A constructor creates objects. Each constructor has an associated prototype object, which is simply another object.
5. When an object is created, it's parent is set to the prototype object associated with the constructor that created it.
6. The prototype objects are used to implement *inheritance* with the mechanism of ***dynamic dispatch (delegation)***.

Static vs. dynamic dispatch

- Static dispatch: references are resolved at compile time
- Dynamic dispatch: resolves the references at runtime.
- Static dispatch in Java:
  - A class may have multiple methods with the same name but different parameter types
  - Method calls are dispatched to the method with the right number of parameters that has the most specific types that the actual parameters could match.

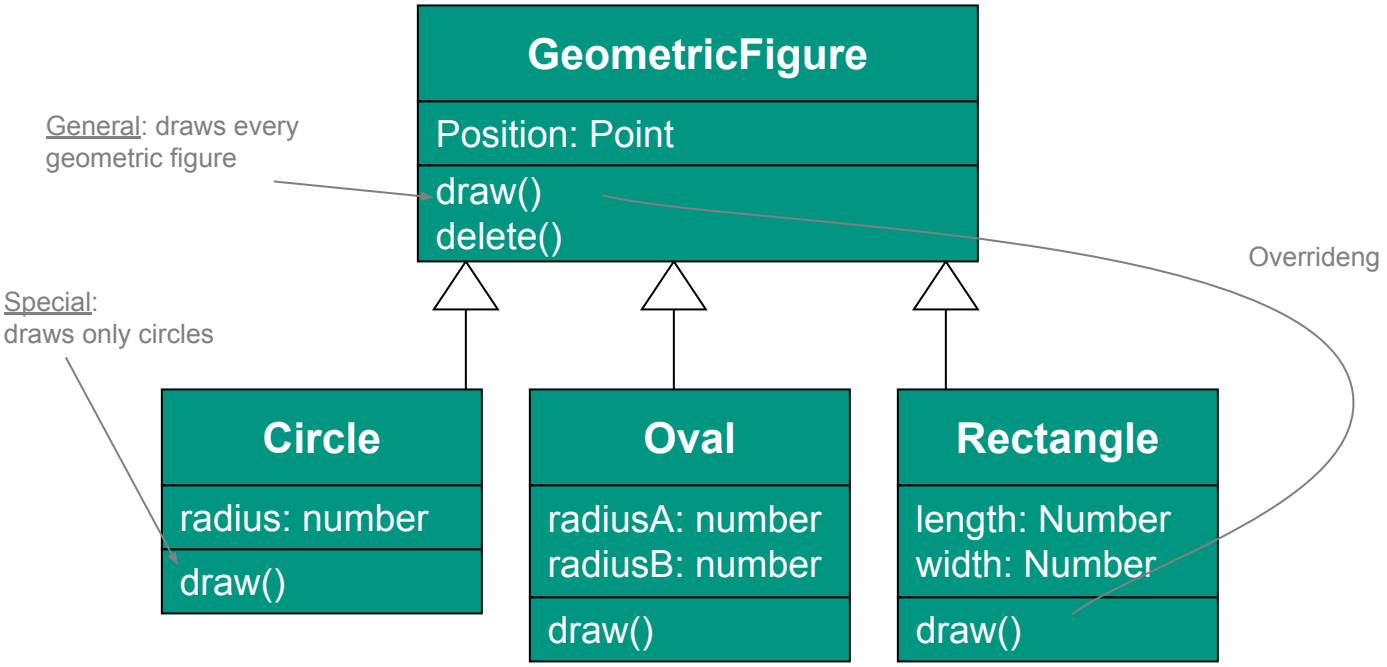
Static vs. dynamic dispatch (2)

- Dynamic (virtual method) dispatch in Java:
  - A subclass can override a method declared in a superclass.
  - At run-time, the JVM has to dispatch the method call to the version of the method that is appropriate to the run-time type of this.
- Double-dispatch is the combination of static and run-time (also called dynamic) dispatches.

Overloading (static dispatch) – Example

```
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    { … }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    { … }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    { … }


}
```

Overriding (Polymorphism) – Example



- Each of the three specializations must implement their own drawing method

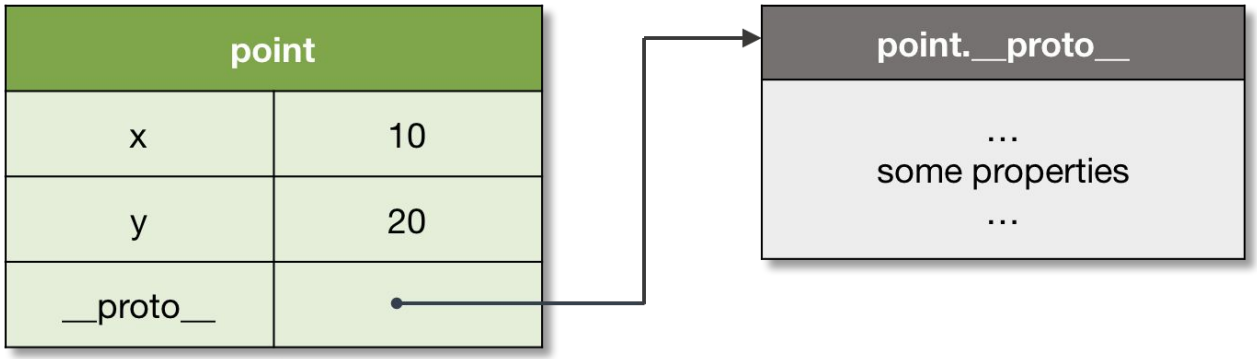
Overriding (Polymorphism) – Example

```
class A
{
    void m1()
    { System.out.println("Inside A's m1 method"); }
}
class B extends A
{
    // overriding m1()
    void m1()
    { System.out.println("Inside B's m1 method"); }
}
class C extends A
{
    // overriding m1()
    void m1()
    { System.out.println("Inside C's m1 method"); }
}
```

Prototype inheritance

- Object: An object is a collection of properties and has a single prototype object.
- A prototype of an object is referenced by the internal [[Prototype]] property, which to user-level code is exposed via the **__proto__** property.

```
1var point = {
2  x: 10,
3  y: 20,
4};
```



Source: http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/

- By default, objects receive Object.prototype as their inheritance object.

Prototype chain

- Any object can be used as a prototype of another object
- If a property is not found in the object itself, there is an attempt to *resolve* it in the prototype; in the prototype of the prototype, etc.
- The prototype can be set *explicitly* via either the **__proto__** property, or **Object.create** method

Dynamic dispatch or *delegation!*

```
1// Base object.
2let point = {
3  x: 10,
4  y: 20,
5};
6
7// Inherit from `point` object.
8let point3D = {
9  z: 30,
10  __proto__: point,
11};
12
13console.log(
14  point3D.x, // 10, inherited
15  point3D.y, // 20, inherited
16  point3D.z  // 30, own
17);
```

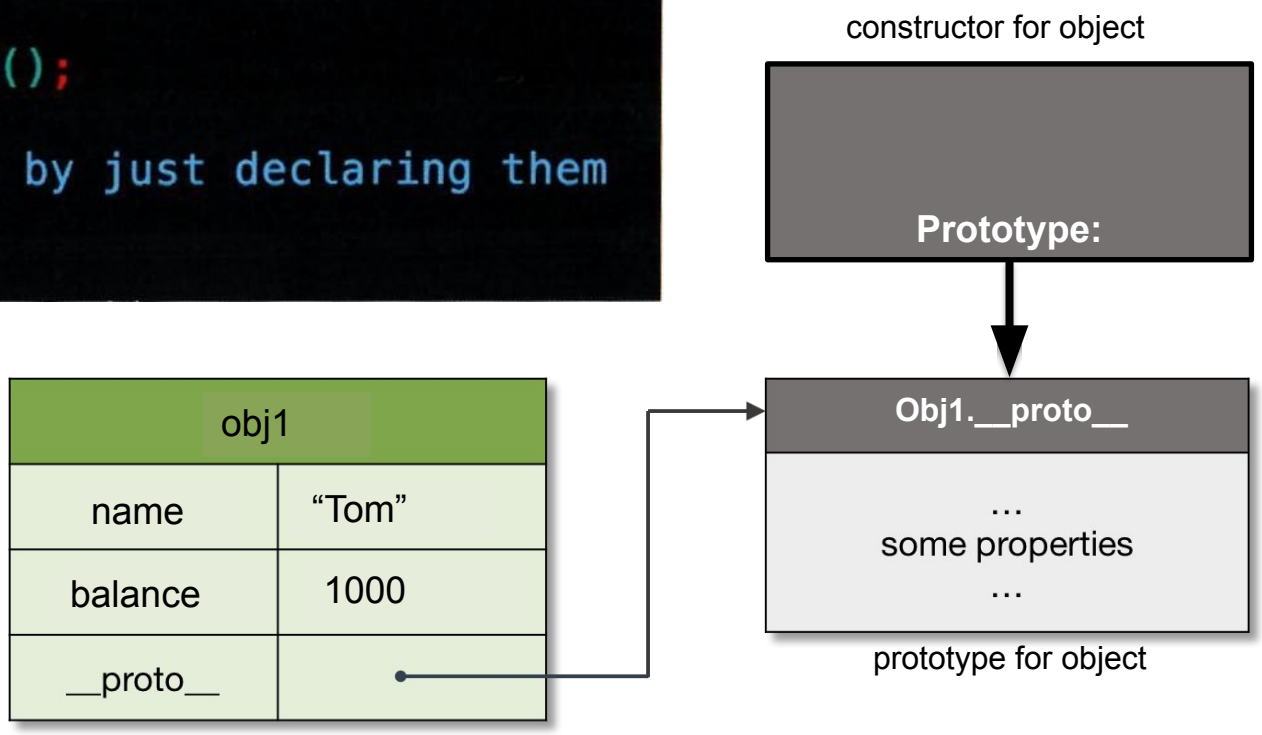
Prototype Inheritance (2)

- Adding Properties and Methods to Objects:
  - Sometimes you want to add new properties (or methods) to all **existing objects** of a given type.
  - Sometimes you want to add new properties (or methods) to **an object constructor**.

- The JavaScript **prototype property** allows you to add **new properties** to object constructors.
- The JavaScript **prototype property** also allows you to add **new methods** to objects constructors.

Prototype inheritance – Example (1)

```
// way one
var obj1 = new Object();

// can add attributes by just declaring them
obj1.name = "Tom";
obj1.balance = 1000;
```



- Every object, when is created, receives its ***prototype***.
- If the prototype is not set *explicitly*, objects receive *default prototype* as their *inheritance object*. (called as **Object.prototype)**

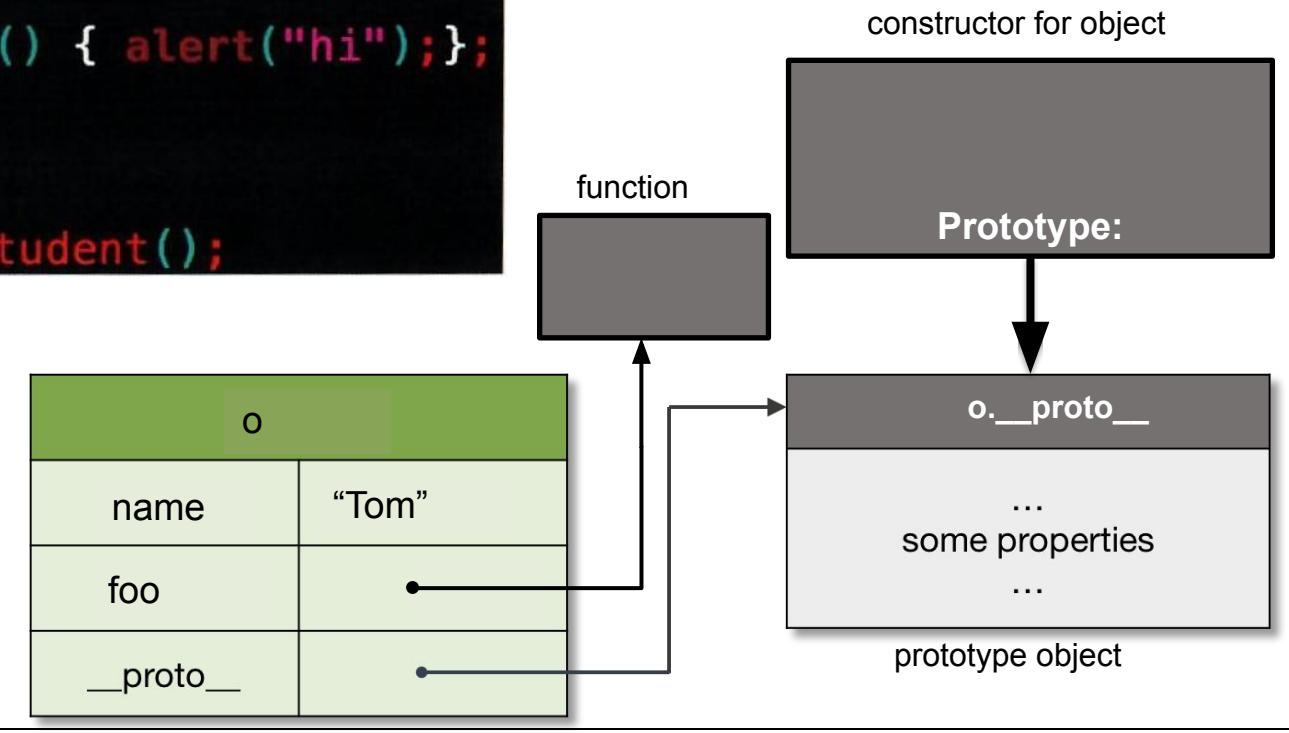
Prototype Inheritance (3)

- All JavaScript objects inherit properties and methods from a prototype:
  - Date objects inherit from **Date.prototype** (prototype of standard JavaScript objects)
  - Array objects inherit from **Array.prototype** (prototype of standard JavaScript objects)
  - Person objects inherit from **Person.prototype** (own prototype)
  - The **Object.prototype** is on the top of the prototype inheritance chain:
  - Date objects, Array objects, and Person objects inherit from Object.prototype.

Prototype inheritance – Example (2)

```
// -----------------
// Factory pattern
// -----------------
function createStudent() {
  var o = new Object();
  o.name = "Tom";
  o.foo = function() { alert("hi");};
  return o;
}

var obj3 = createStudent();
```

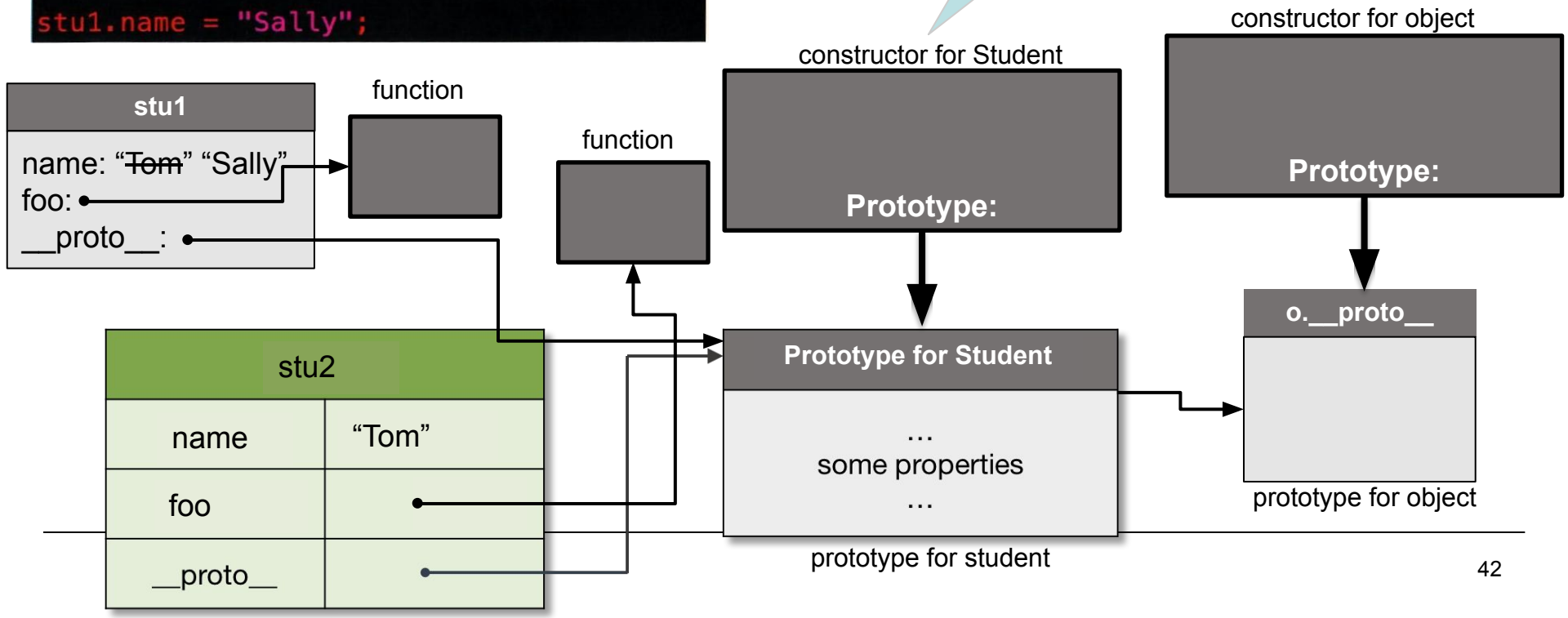
When a **function** is created, the JavaScript engine adds a **prototype** property to the **function**. This **prototype** property is an object (called as **prototype** object).



Prototype inheritance – Example (3)

```
// -----------------
// Constructor pattern
// -----------------

function Student () { // called a constr
  this.name = "Tom";
  this.foo = function() {alert("hi"); };
};

var stu1 = new Student(); // create a ne
var stu2 = new Student();

stu1.name = "Sally";
```

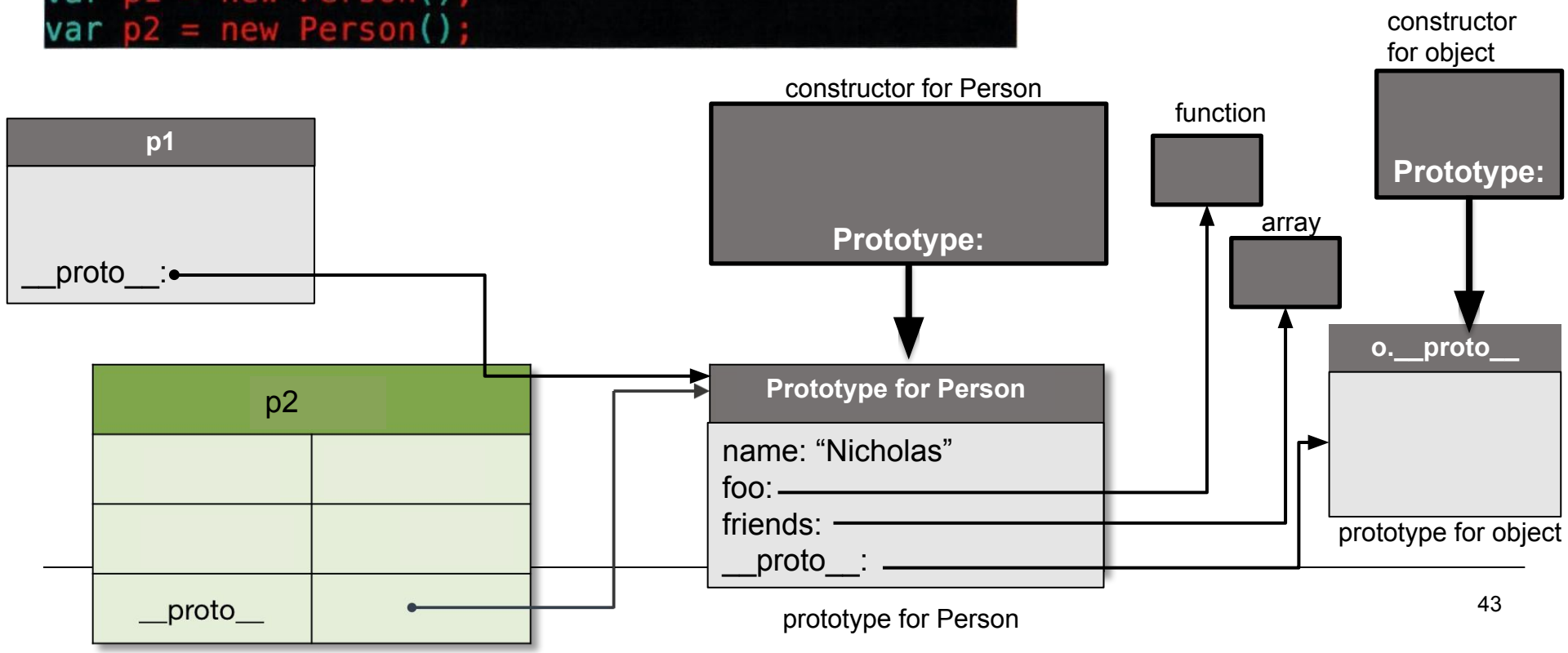
Prototype object of the constructor function is **shared** among all the objects created using the constructor function!



Prototype inheritance – Example (4)

```
// ----------------
// Prototype pattern
// ----------------
function Person() {};
Person.prototype.name = "Nicholas";
Person.prototype.foo = function() {alert("hi");};
Person.prototype.friends = ["Tom","Sally"];

var p1 = new Person();
var p2 = new Person();
```

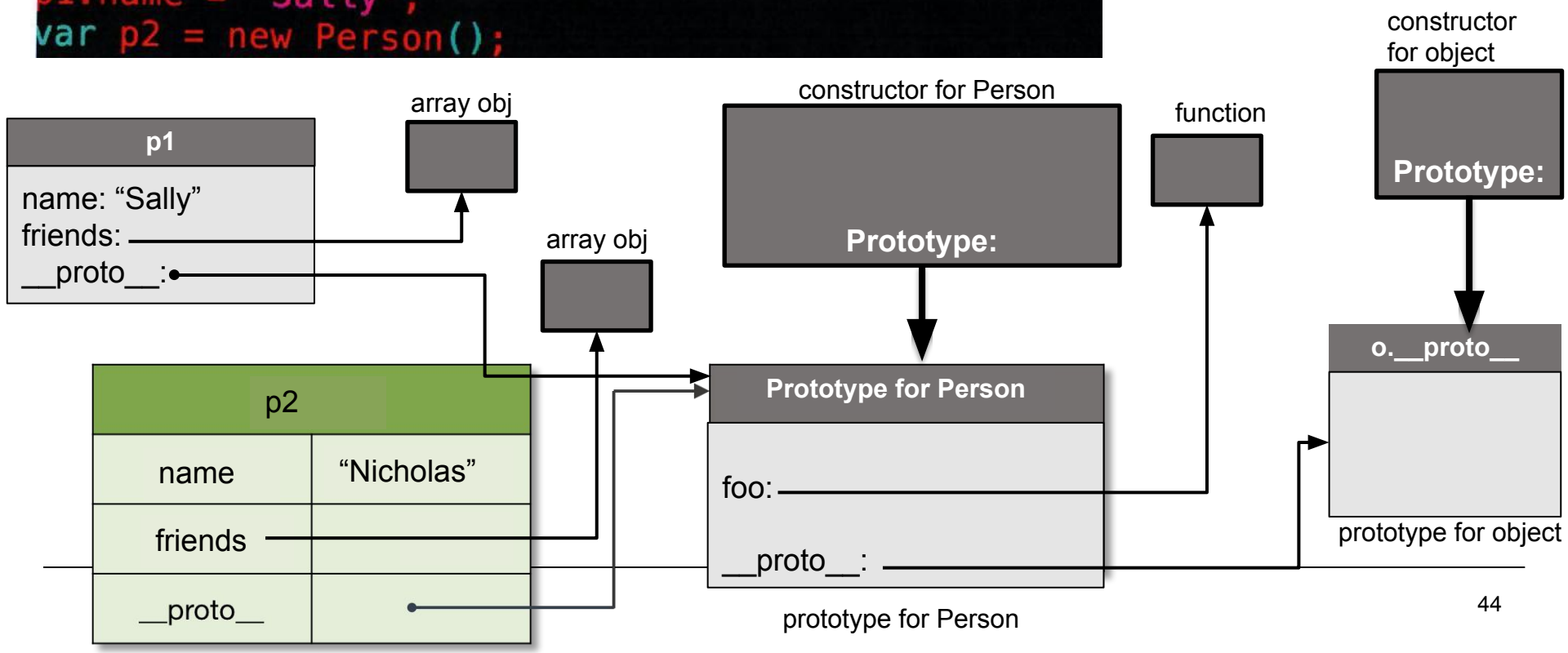
prototype property allows you to add new properties/methods to object constructors (to all existing objects of a given type)



Prototype inheritance – Example (5)

```
function Person() {
  this.name = "Nicholas";
  this.friends = ["Sam","Molly"];
}
Person.prototype.foo = function() {alert("hi");};
var p1 = new Person();
p1.name = "Sally";
var p2 = new Person();
```

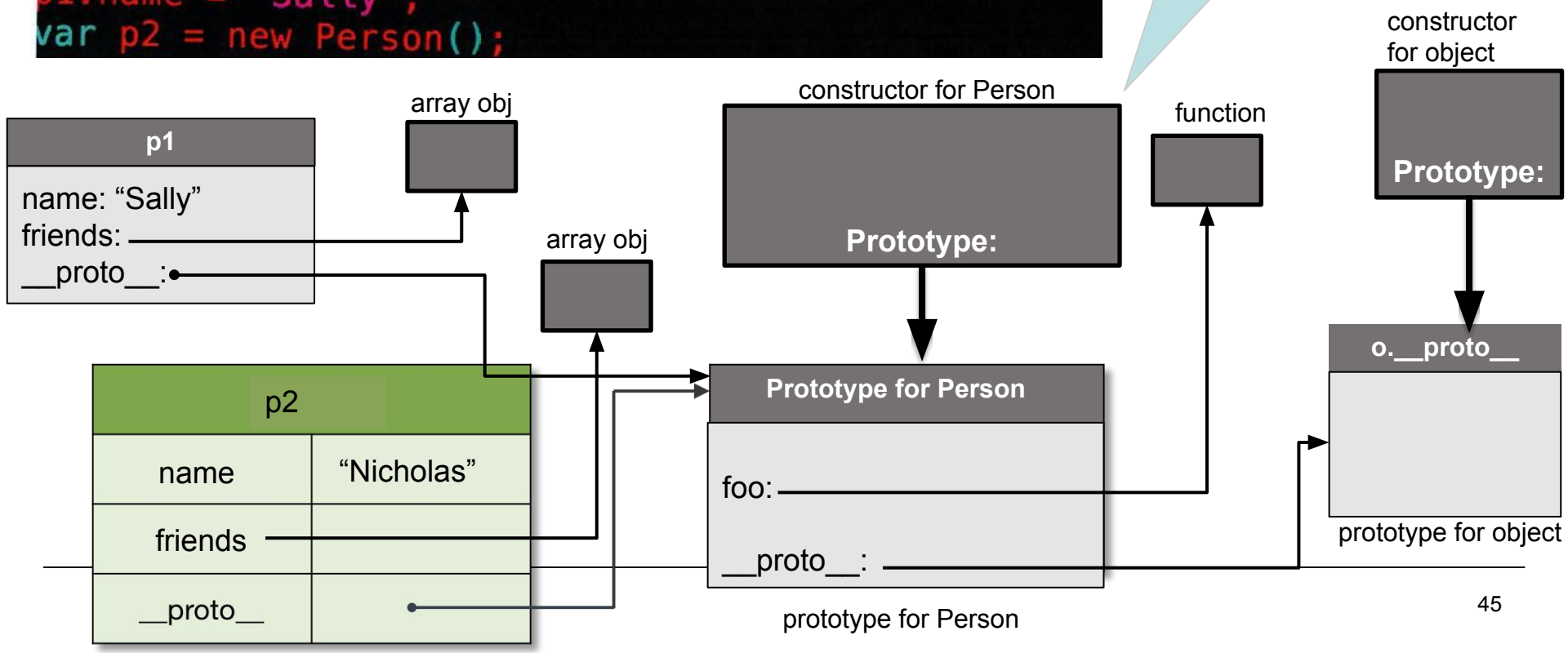
Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects!



Prototype inheritance – Example (5)

```
function Person() {
  this.name = "Nicholas";
  this.friends = ["Sam","Molly"];
}
Person.prototype.foo = function() {alert("hi");};
var p1 = new Person();
p1.name = "Sally";
var p2 = new Person();
```

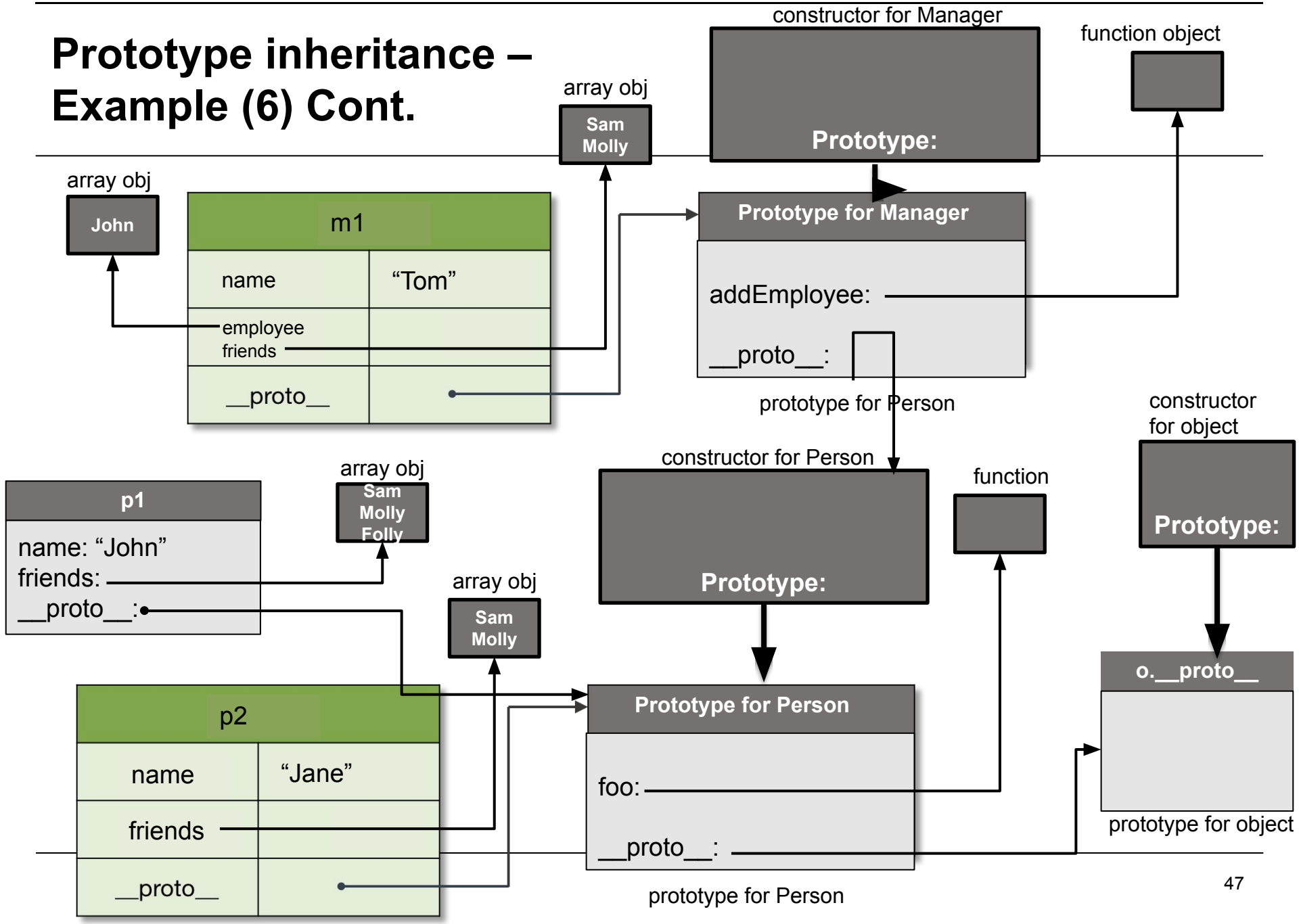
Prototype object of the constructor function is **shared** among all the objects created using the constructor function!



Prototype inheritance – Example (6)

```
class Person {
  constructor(s) {
    this._name = s;
    this._friends = ["Sam", "Molly"];
  }

  foo() {
    console.log("hi " + this._name);
    console.log(this._friends);
  }

}

let p1 = new Person("John");
let p2 = new Person("Jane");

p1._friends.push("Folly");

class Manager extends Person {

  constructor(s) {
    super(s);
    this._employee = [];
  }

  addEmployee(s) {
    this._employee.push(s);
  }


}

p1.foo();
p2.foo();

m1 = new Manager("Tom");
m1.addEmployee("John");
```

Prototype inheritance – Example (6) Cont.



Literature – JavaScript

- https://www.w3schools.com/
- JavaScript. The Core: 1st and 2nd Edition
  - http://dmitrysoshnikov.com/ecmascript/javascript-the-core-2nd-edition/
  - http://dmitrysoshnikov.com/ecmascript/javascript-the-core/