
Construction of User Interfaces (SE/ComS 319)

Ali Jannesari

Jinu Susan Kabala

Department of Computer Science

Iowa State University, Spring 2021

JAVASCRIPT SPOTLIGHTS

How to add js to html file

// how to include in html file

<script> your javascript code goes in here **</script>**

// can also include from a separate file

`<script src="./01_example.js"></script>`

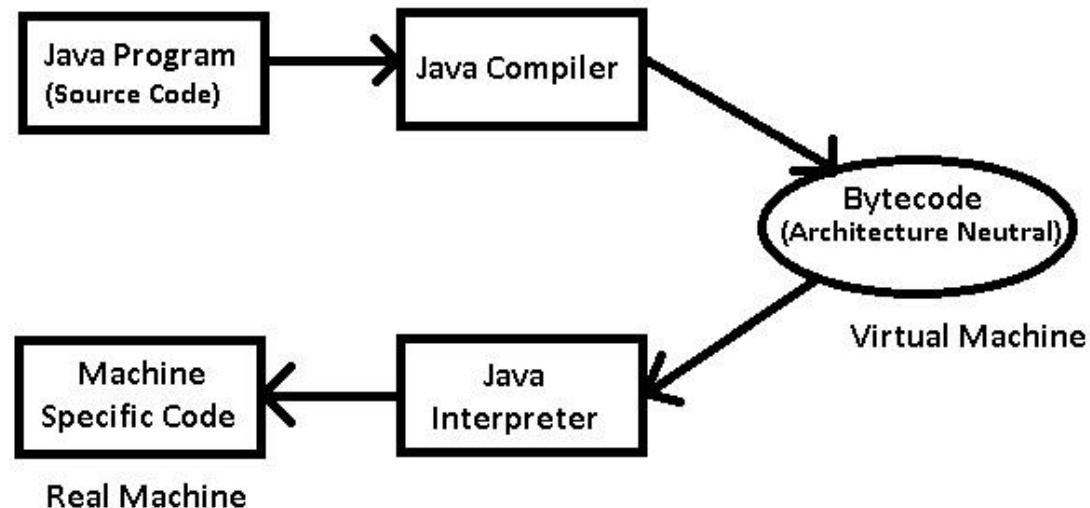
// can include from a remote site

`<script src="http://.../a.js"></script>`

JavaScript – Interpreter

Interpreter vs. compiler?

- JavaScript is **interpreted** at runtime by the client browser
 - Similar to Java Virtual Machine (JVM) that interprets **byte code**
- Java compiler
 - Provides an abstract machine that is programmed in Java
 - Is based on another abstract machine, i.e. the Java VM
 - Machine commands, bytecode is hidden
 - The masking is checked by the compiler



Interpreter vs. Compiler

Interpreter: (example: **JavaScript**, Python, Ruby).

- Translates program one statement at a time □ less amount of time to analyze the source code but the overall execution time is slower.
- No intermediate object code is generated □ memory efficient.
- Continues translating the program until the first error is met, in which case it stops □ debugging is easy.

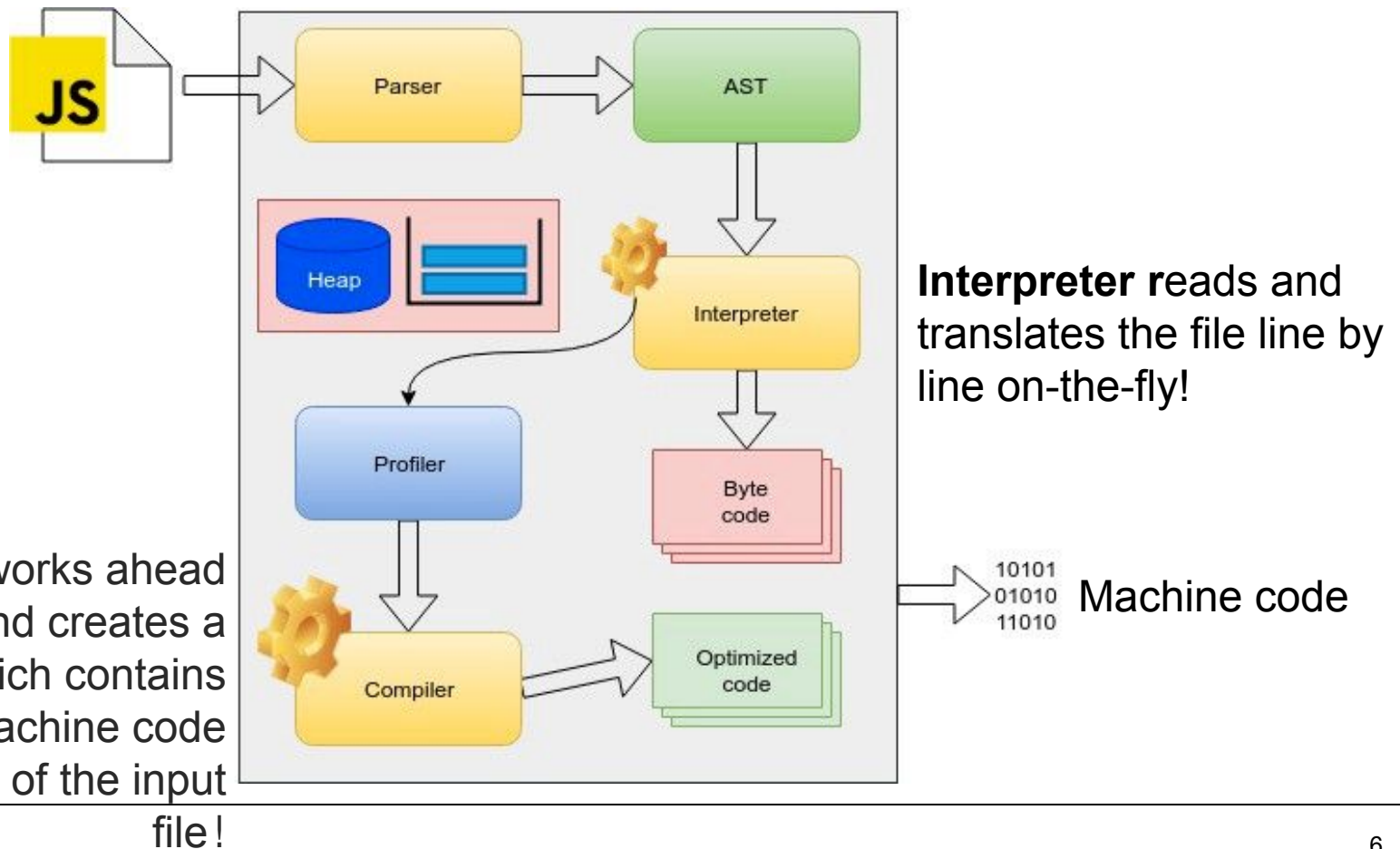
Interpreter vs. Compiler (2)

Compiler: (example: C/C++)

- Scans the entire program and translates it into machine code □ large amount of time to analyze the source code but the overall execution time is comparatively faster.
- Generates intermediate object code and requires linking □ more memory
- Error message after scanning the whole program □ Debugging hard

JavaScript – Interpreter (2)

- How does JavaScript work?



JavaScript syntax

The JavaScript syntax is similar to C# and Java

- Operators (+, *, =, !=, &&, ++, ...)
- Variables (**typeless**) □ JavaScript is typeless
 - Variables can store any sort of data
 - Doesn't prioritize type safety
- Conditional statements (if, else)
- Loops (for, while)
- Arrays (my_array[])
- Associative arrays (my_array['abc'])
- Functions

JavaScript basics

Declaring variables: `var` keyword, typeless variables

Basic I/O: `document.writeln();`
 `window.alert();`
 `window.prompt();`

Type conversion: `parseInt();`

Data types

JavaScript data types:

- Numbers (integer, floating-point)
- Boolean (true / false)

String type – string of characters

```
var myName = "You can use both single or double  
quotes for strings";
```

Arrays

```
var my_array = [1, 5.3, "aaa"];
```

Associative arrays (hash tables)

```
var my_hash = {a:2, b:3, c:"text"};
```

Data types (2)

- Every variable can be considered as **object**
 - Arrays are objects
- Objects use **names** to access its "members"
- Example
 - **person.firstName** returns **John**:

```
var person = {firstName:"John", lastName:"Doe", age:46};
```

Static and dynamic binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution
- A binding is *dynamic* if it first occurs during execution or can change during execution of the program
- **JavaScript: Dynamic Type Binding!**

Dynamic type binding in JavaScript

- Specified through an assignment statement

```
list = [2, 4.33, 6, 8];
```

```
list = 17.3;
```

- Advantage:
 - Flexibility (generic program units)
- Disadvantages:
 - High cost (dynamic type checking, interpretation, and dynamic storage allocation)
 - Type error detection by the compiler is difficult (less reliable)
 - Usually implemented using pure interpretation

String operations

The + operator joins strings

```
string1 = "fat ";  
string2 = "cats";  
alert(string1 + string2); // fat cats
```

What is "9" + 9?

```
alert("9" + 9); // 99
```

Converting string to number:

```
alert(parseInt("9") + 9); // 18
```

Arrays operations and properties

Declaring new empty array:

```
var arr = new Array();
```

Declaring an array holding few elements:

```
var arr = [1, 2, 3, 4, 5];
```

Appending an element / getting the last element:

```
arr.push(3); // Add items to the end of an array  
var element = arr.pop(); // Remove an item from the end of an array
```

Reading the number of elements (array length):

```
arr.length;
```

Variable declaration: var vs. let

- **let** : declare variables that are limited in scope to the block, statement, or expression on which it is used.

```
let x = 2;
}
```

// x can NOT be used here

- **var** : defines a variable globally, or locally to an entire function regardless of block scope.

```
var x = 2;
}
```

// x CAN be used here

Example:

```
var x = 10;
// Here x is 10
{
  var x = 2;
  // Here x is 2
}
// Here x is ???
```

Everything is object!

Every variable can be considered as object

- For example strings and arrays have member functions:

```
var test = "some string";  
alert(test.charAt(5)); // shows letter 's'  
alert("test".charAt(1)); //shows letter 'e'  
alert("test".substring(1,3)); //shows 'es'
```

```
var arr = [1,3,4];  
alert (arr.length); // shows 3  
arr.push(7); // appends 7 to end of array  
alert (arr[3]); // shows 7
```


Sum of numbers – Example

sum-of-numbers.html

```
<html>

<head>
  <title>JavaScript Demo</title>
  <script type="text/javascript">
    function calcSum() {
      value1 =
        parseInt(document.mainForm.textBox1.value);
      value2 =
        parseInt(document.mainForm.textBox2.value);
      sum = value1 + value2;
      document.mainForm.textBoxSum.value = sum;
    }
  </script>
</head>
```

Switch statement

The switch statement works like in C# / Java:

```
switch (variable) {  
    case 1:  
        // do something  
        break;  
    case 'a':  
        // do something else  
        break;  
    case 3.14:  
        // another code  
        break;  
    default:  
        // something completely different  
}
```

Loops

Like in C# / Java / C++

- for loop
- while loop
- do ... while loop

```
var counter;  
for (counter=0; counter<4; counter++) {  
    alert(counter);  
}  
while (counter < 5) {  
    alert(++counter);  
}
```

Operators: pre-increment vs. post-increment

- **++x** (pre-increment): increment the variable; the value of the expression is the final value
- **x++** (post-increment): remember the original value, then increment the variable; the value of the expression is the original value.
 - **Example:**

```
x = 0;  
y = array[x++];  
// This will get array[0]
```

```
x = 0;  
y = array[++x];  
// This will get array[1]
```

Functions

```
function average(a, b, c)
{
    var total;
    total = a+b+c;
    return total/3;
}
```

Parameters come in here.

Declaring variables is optional. Type is never declared.

Value returned here.

Function arguments and return value

- Functions are not required to return a value
- When calling function it is not obligatory to specify all of its arguments
 - The function has access to all the arguments passed via arguments array

```
function sum() {  
    var sum = 0;  
    for (var i = 0; i < arguments.length; i ++)  
        sum += parseInt(arguments[i]);  
    return sum;  
}  
alert(sum(1, 2, 4));
```

Standard popup boxes

- Alert box with text and [OK] button
 - Just a message shown in a dialog box:

```
alert("Some text here");
```

- Confirmation box
 - Contains text, [OK] button and [Cancel] button:

```
confirm("Are you sure?");
```

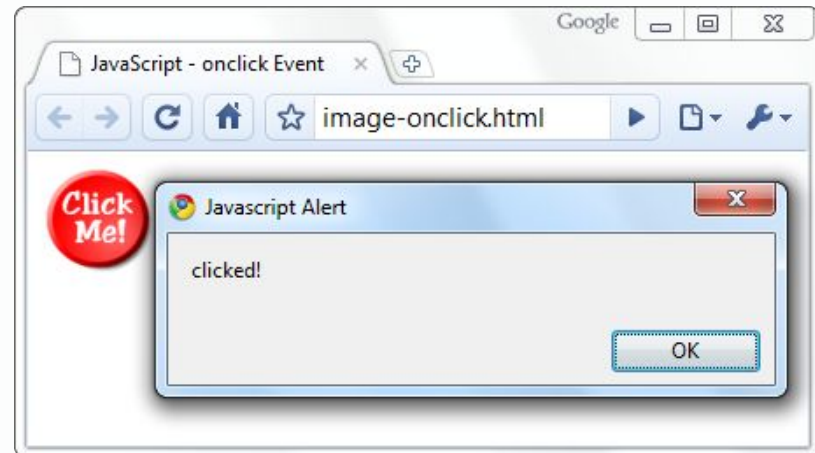
- Prompt box
 - Contains text, input field with default value:

```
prompt ("enter amount", 10);
```

Calling a JavaScript function from Event Handler – Example

```
<html>
<head>
<script type="text/javascript">
    function test (message) {
        alert(message);
    }
</script>
</head>

<body>
    
</body>
</html>
```



While loops

```
while( expression )  
    statement;
```

Executes a statement until expression becomes false

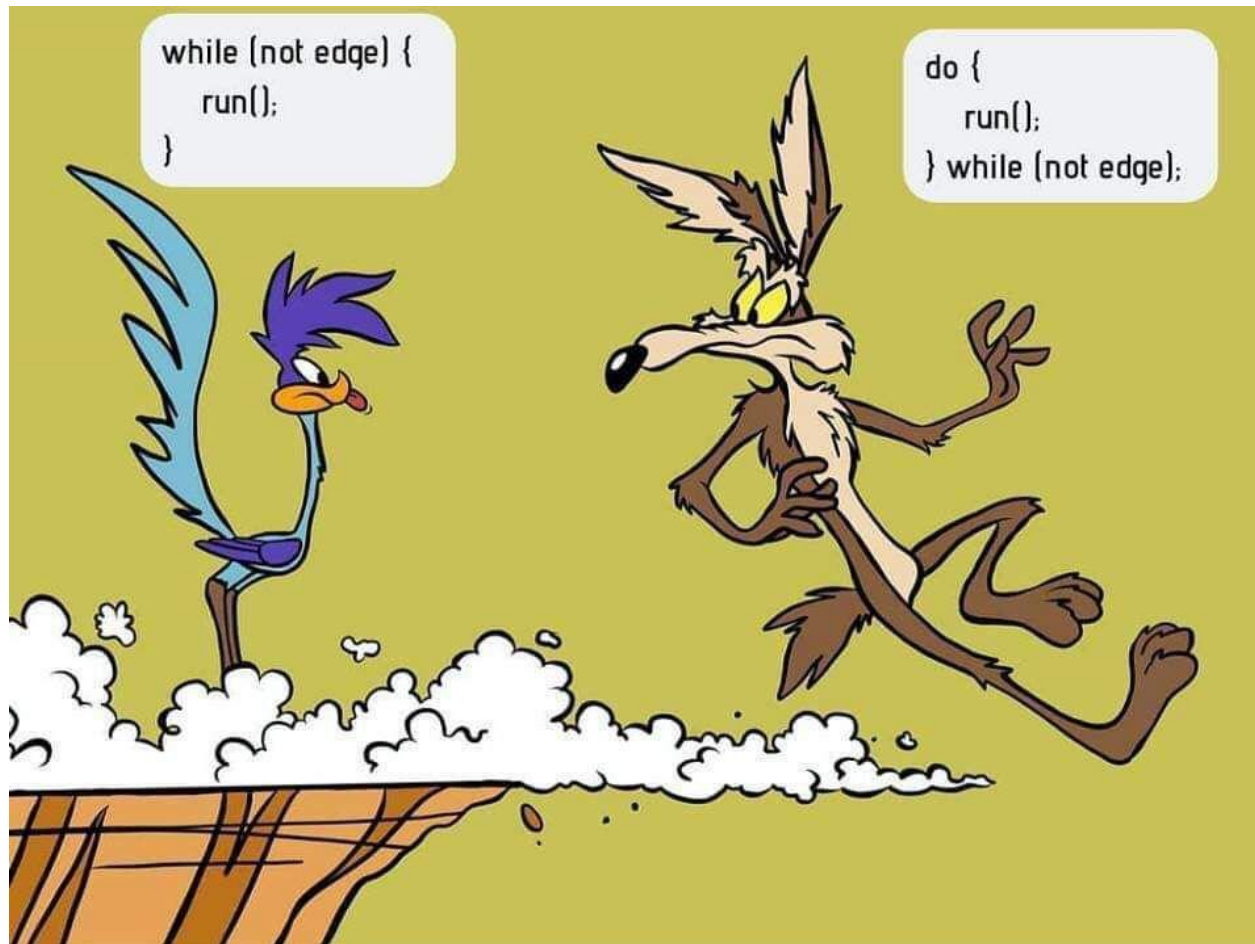
Evaluates expression before first iteration

```
do  
    statement;  
while( expression );
```

Evaluates expression after first iteration

Executes statement at least once

While loops – “while” vs. “do... while”



While-loop example

Print the numbers 0 to 99 to the screen

```
int i = 0;
while ( i < 100 )
{
    alert(i);
    i++;
}
```

For loops (1)

General format:

```
for( expr1; expr2; expr3 )  
    statement;
```

expr1 is executed at the beginning of the loop

expr2 is executed at the beginning of every iteration

- If it is false, the loop ends

expr3 is executed at the end of every iteration

For loops (2)

General format:

```
for( expr1; expr2; expr3 )  
    statement;
```

It is possible to omit any of the expressions

- The semicolon must stay

If `expr2` is omitted, the condition is always true

- it becomes an infinite loop

```
for( ;; ) //infinite loop
```

For loops (3)

Usual use case:

```
int i;  
for( i=0; i < 100; i++ )  
    alert(i);
```

Break statement

`break;`

Terminates the innermost loop or switch statement

Execution resumes after the loop or switch statement

```
while ( 1 )  
{  
    n++;  
    if ( n > 5 ) break;  
}
```

Continue statement

`continue;`

Terminates the current iteration of the innermost loop

Execution resumes at the beginning of the next iteration

```
for (i=0; i<100; i++)  
{  
    if ( i == 57 ) continue;  
    alert( i );  
}
```

Print the numbers 0 to 99, but not 57

Accessing DOM

- GET the DOM element by ID or CLASS attributes

`<p id="xyz" class="abc"> </p>`

`document.getElementById("xyz")`

`document.getElementsByClassName("abc")`

`someDOMelement.value // this is value of the element`

Accessing DOM – Example

```
<html>
<body>
<h2>JavaScript Arrays</h2>

<p>JavaScript array elements are accessed using
numeric indexes (starting from 0).</p>

<p id="demo"></p>

<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML =
cars[0];
</script> </body> </html>
```

How to print

`document.write()` // write to DOM

`document.write("test")`

`console.log()` // write to console

`alert()` // popup

`<p id="xyz" class="abc"> </p>`

`document.getElementById("xyz").innerHTML= "hi"`

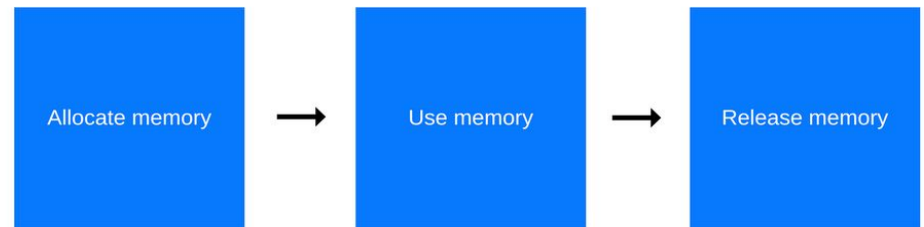
Demonstration

- Demonstration
 - JavaScript example in Browser
- Good resource for JavaScript:
 - <https://www.w3schools.com/>

JAVASCRIPT MEMORY MANAGEMENT

Memory management in JavaScript

- 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
```

Static allocation	Dynamic allocation
<ul style="list-style-type: none">• Size must be known at compile time• Performed at compile time• Assigned to the stack• FILO (first-in, last-out)	<ul style="list-style-type: none">• Size may be unknown at compile time• Performed at run time• Assigned to the heap• No particular order of assignment

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)

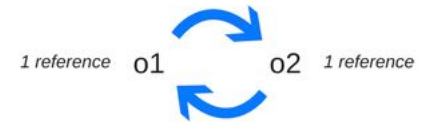
- **Limitation** of Reference-counting garbage collection

- 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();`



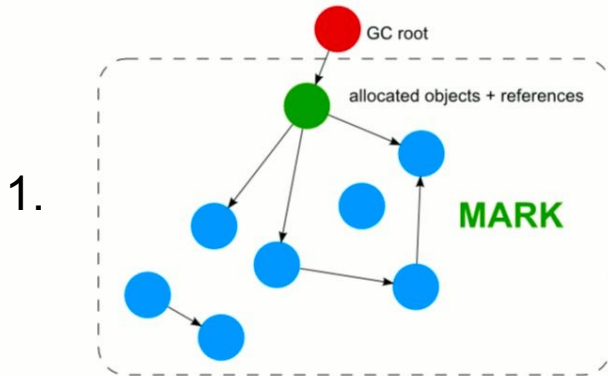
- Internet Explorer 6 and 7 are known to have reference-counting garbage collectors

Mark-and-sweep algorithm (1)

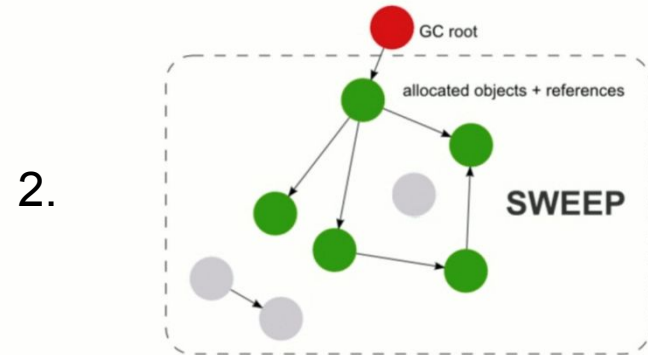
- Mark-and-sweep algorithm
 - an object is unreachable □ Garbage
 - knowledge of a set of objects called roots
 - In JavaScript, the root is the global object
 - Periodically, the garbage-collector will start from these roots
 - Finds all objects that are referenced from these roots
 - The garbage collector 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)

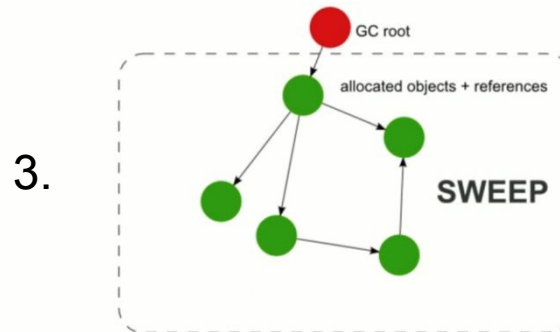
Mark and sweep (MARK)



Mark and sweep (SWEEP)



Mark and sweep (SWEEP)

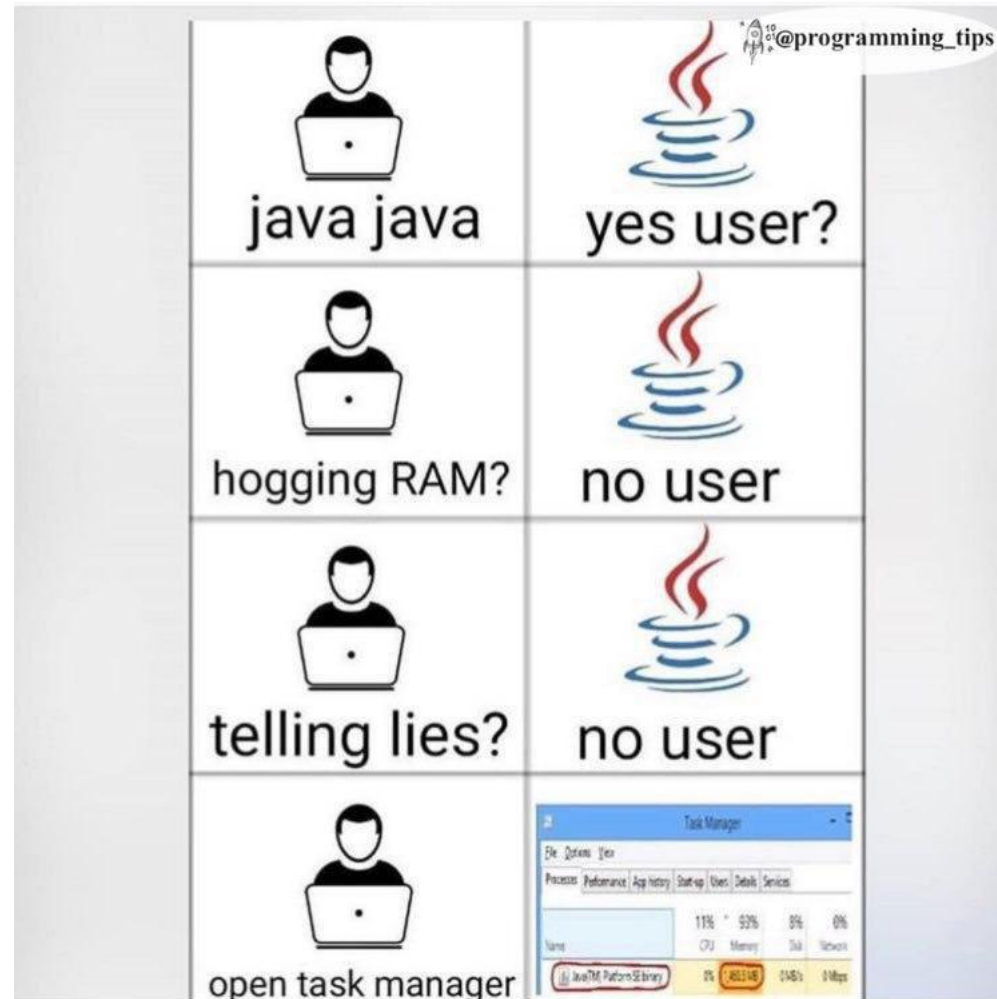


- All modern browsers ship a mark-and-sweep garbage-collector

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

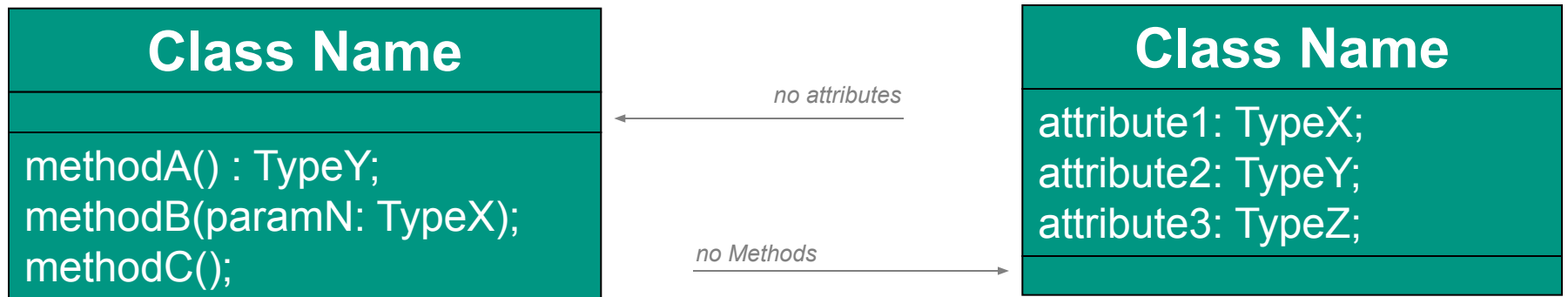
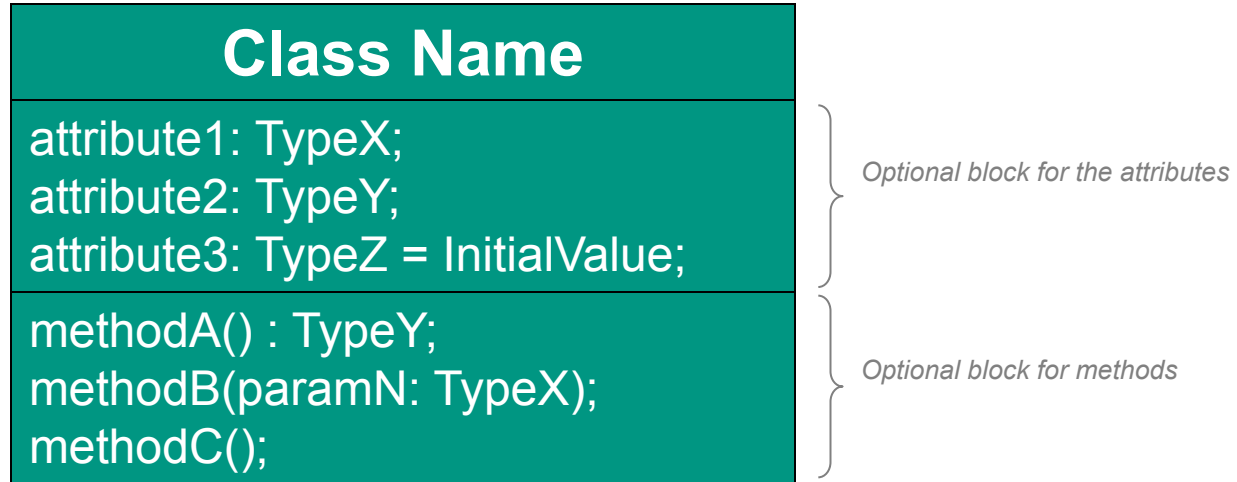
Garbage collection...

- 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!



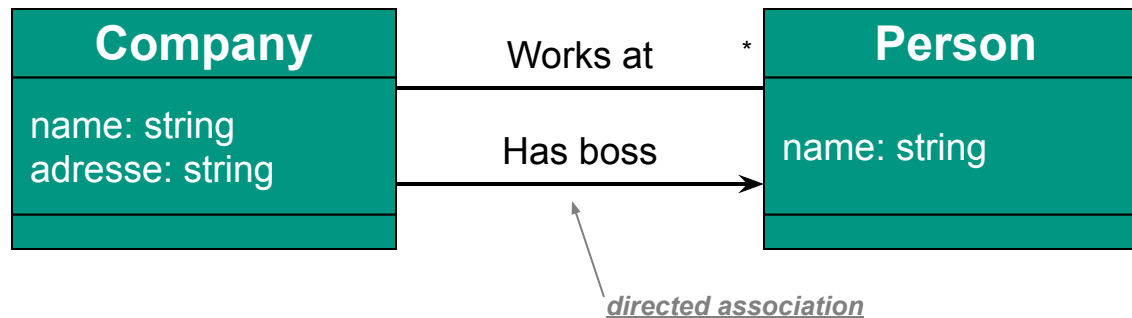
JAVASCRIPT PROTOTYPE-BASED INHERITANCE

Classes – UML class diagram

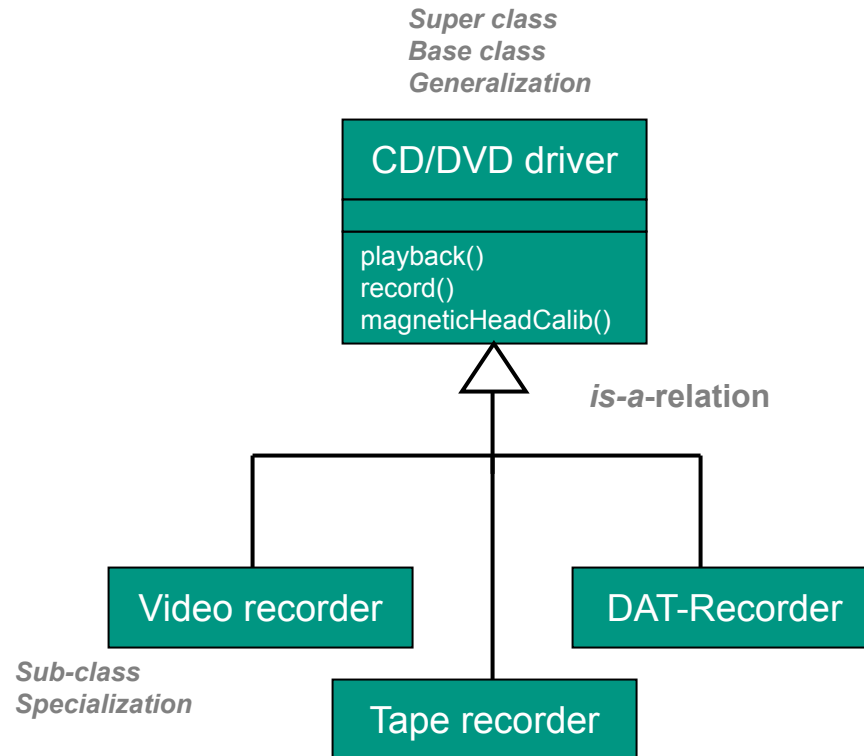


Class diagram – Example

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

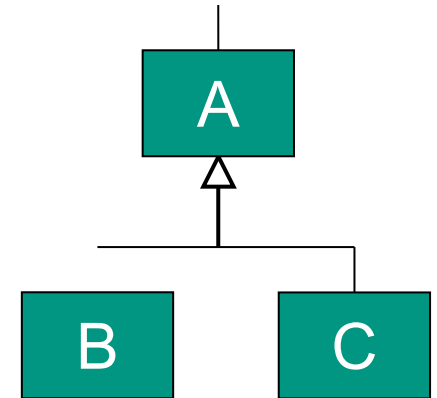


Inheritance

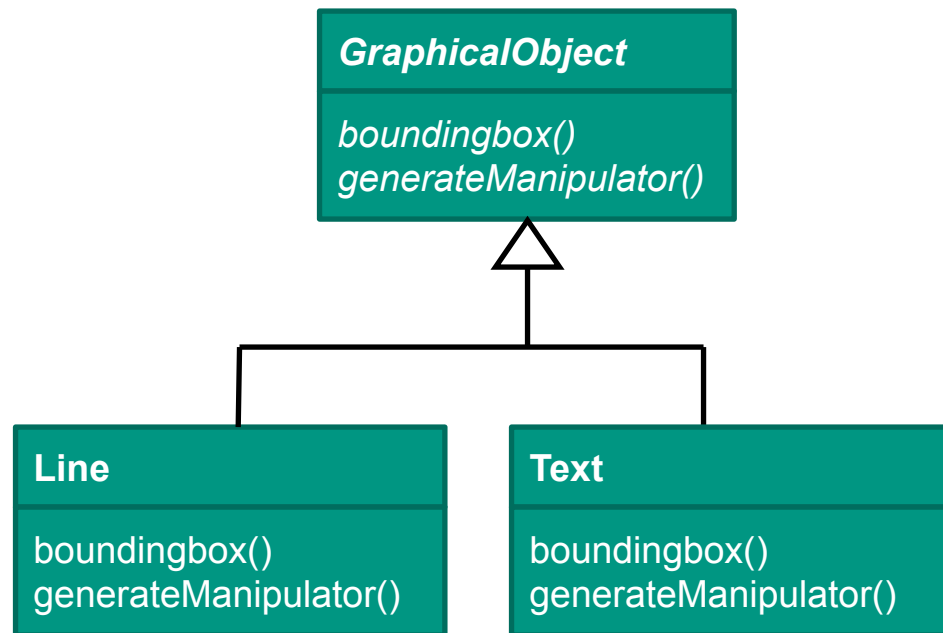


Inheritance – “is-a” relationship

- 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: $\Omega B \subseteq \Omega A$.
- 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, its 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.

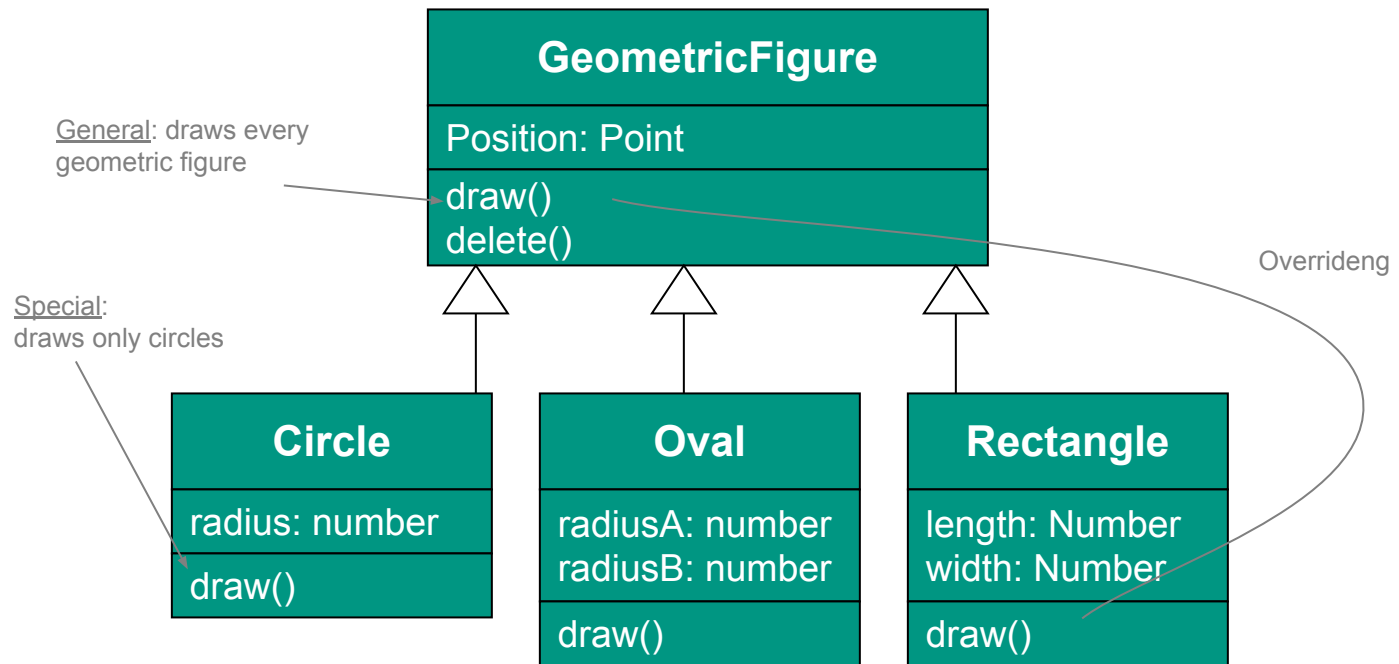
Static vs. dynamic dispatch (2)

- 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

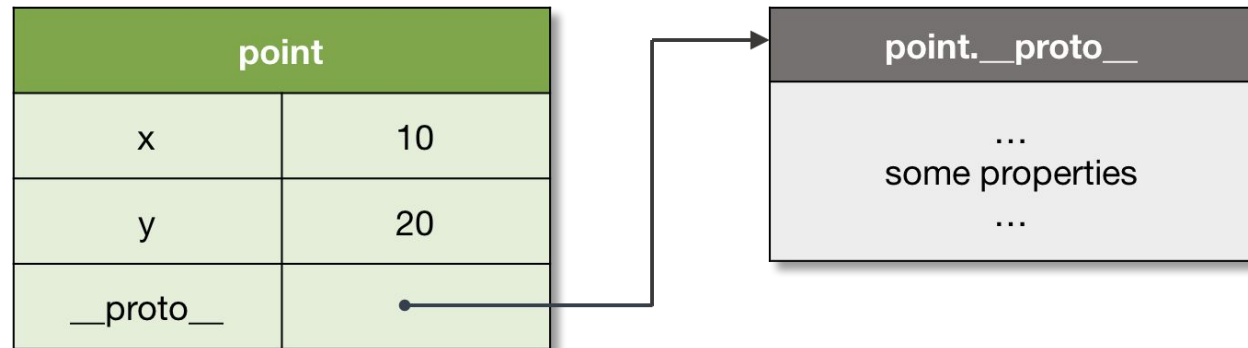
Overriding (Polymorphism) – Example

```
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.

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



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

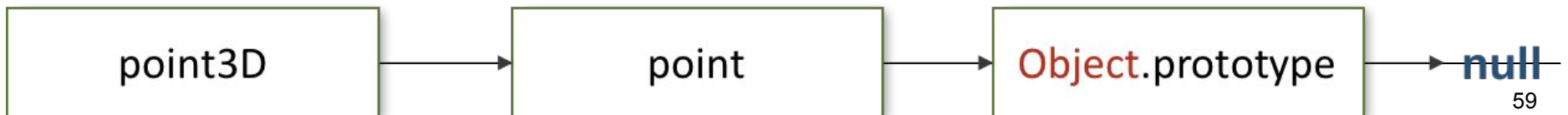
- By default objects receive `Object.prototype` as their inheritance object.

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);
```



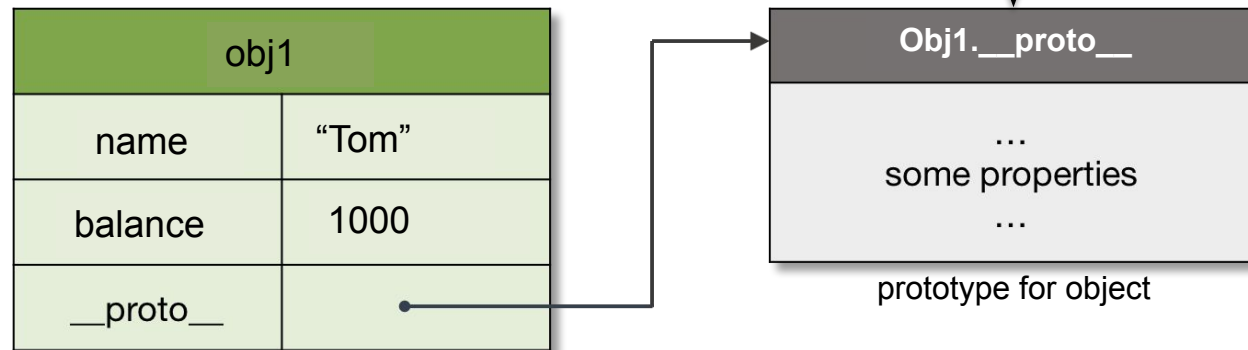
Prototype Inheritance (2)

- 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**)

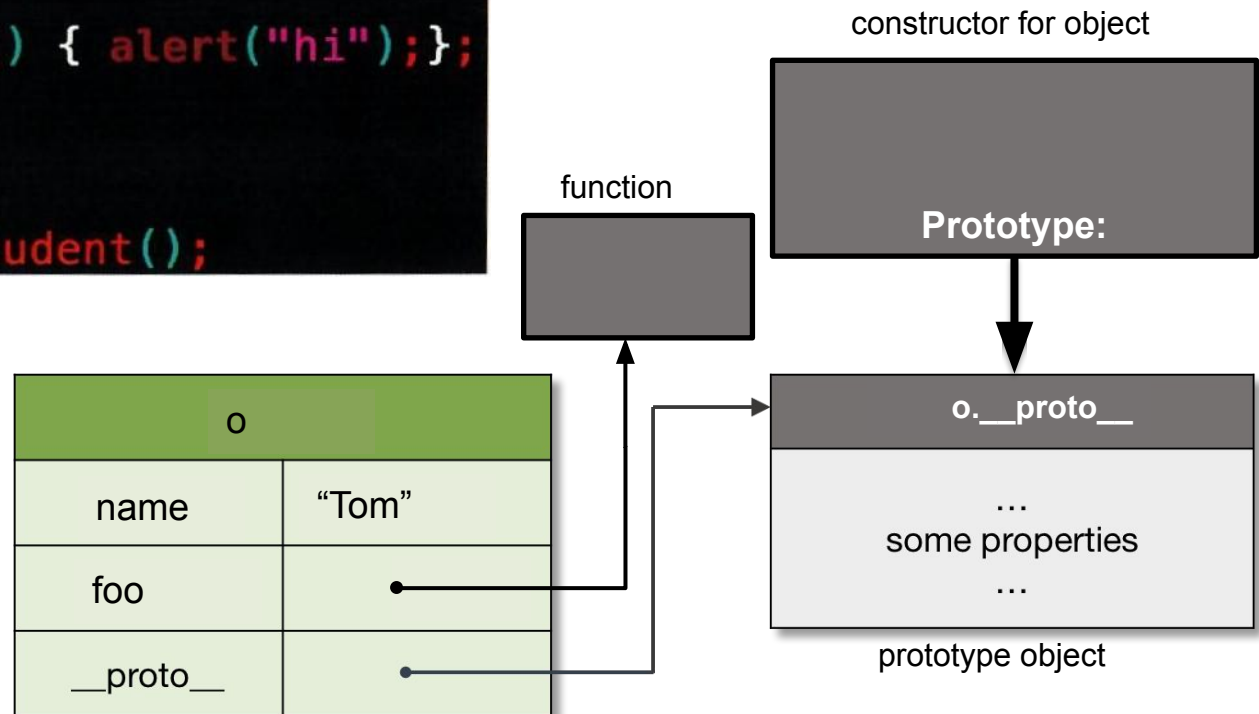
Prototype Inheritance (3)

- 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();
```

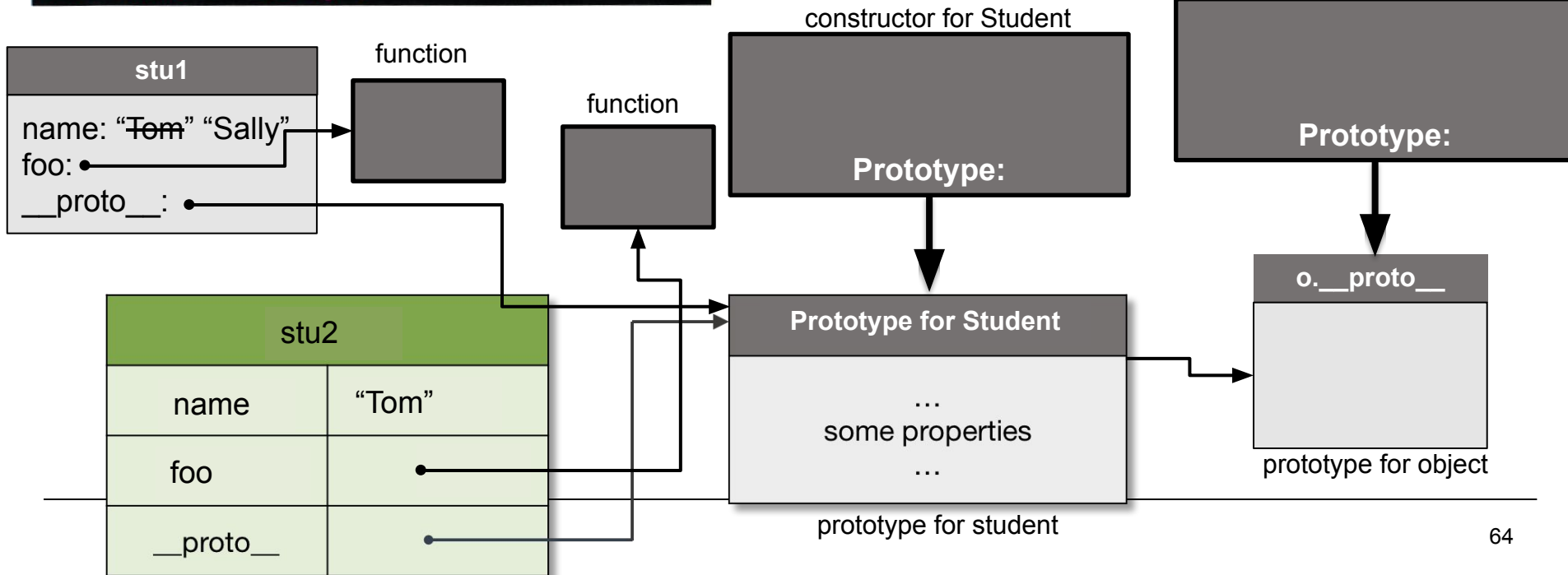
When a **function** is created, the JavaScript engine adds a **prototype** property to the **function**. This **prototype** property is an object (called as **prototype object**).



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";
```

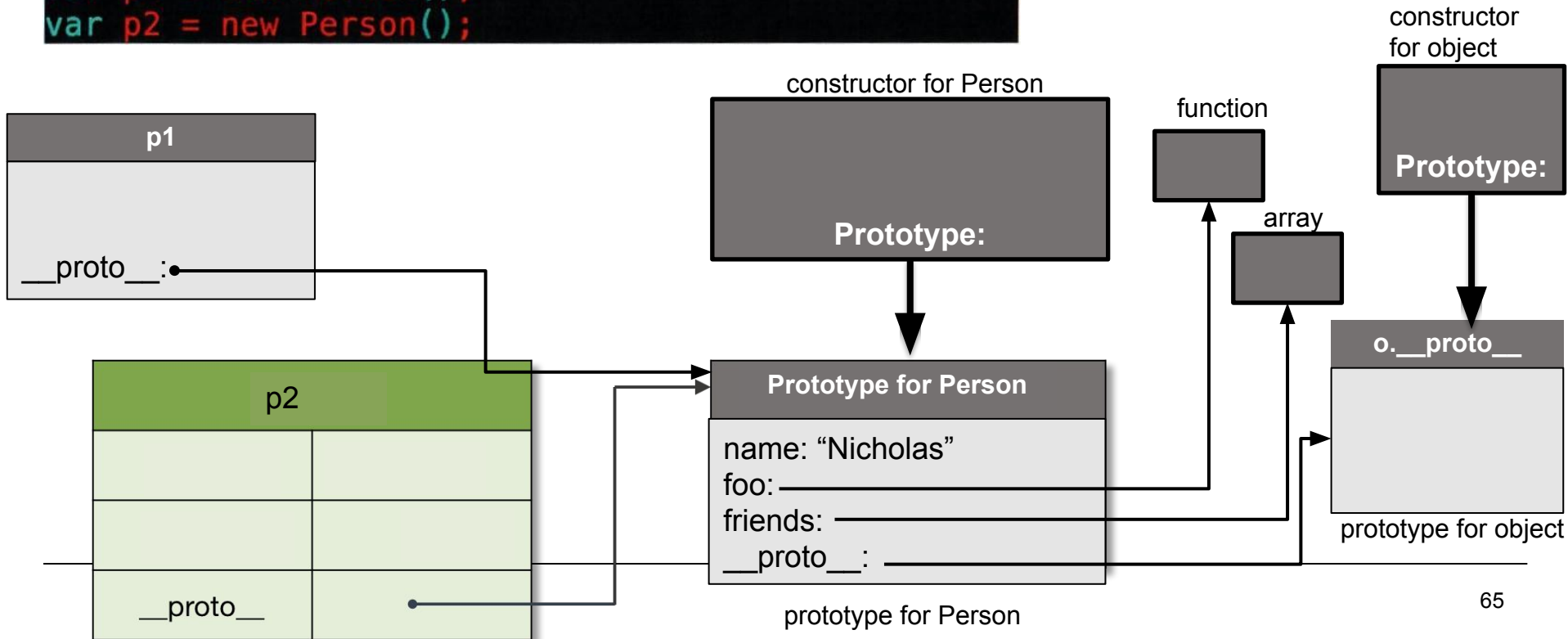
Prototype object of the constructor function is **shared** among all the objects created using the constructor function!



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();
```

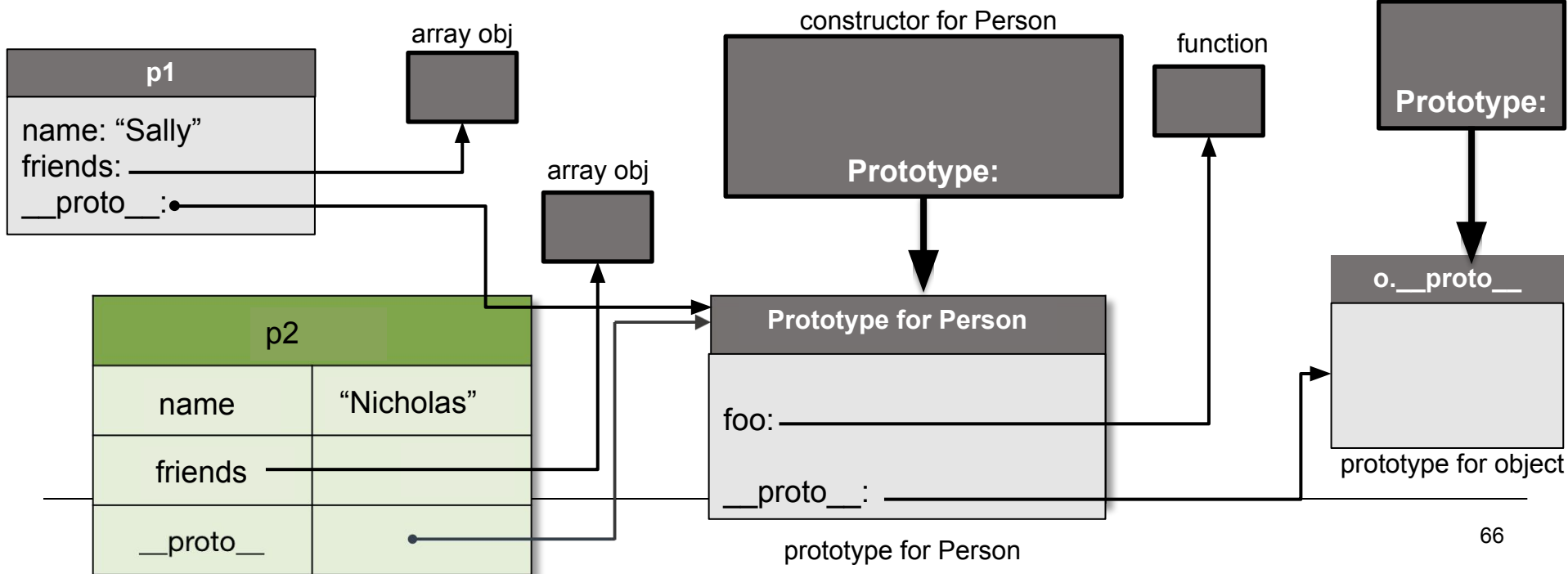
prototype property allows you to add new properties/methods to object constructors (to all existing objects of a given type)



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();
```

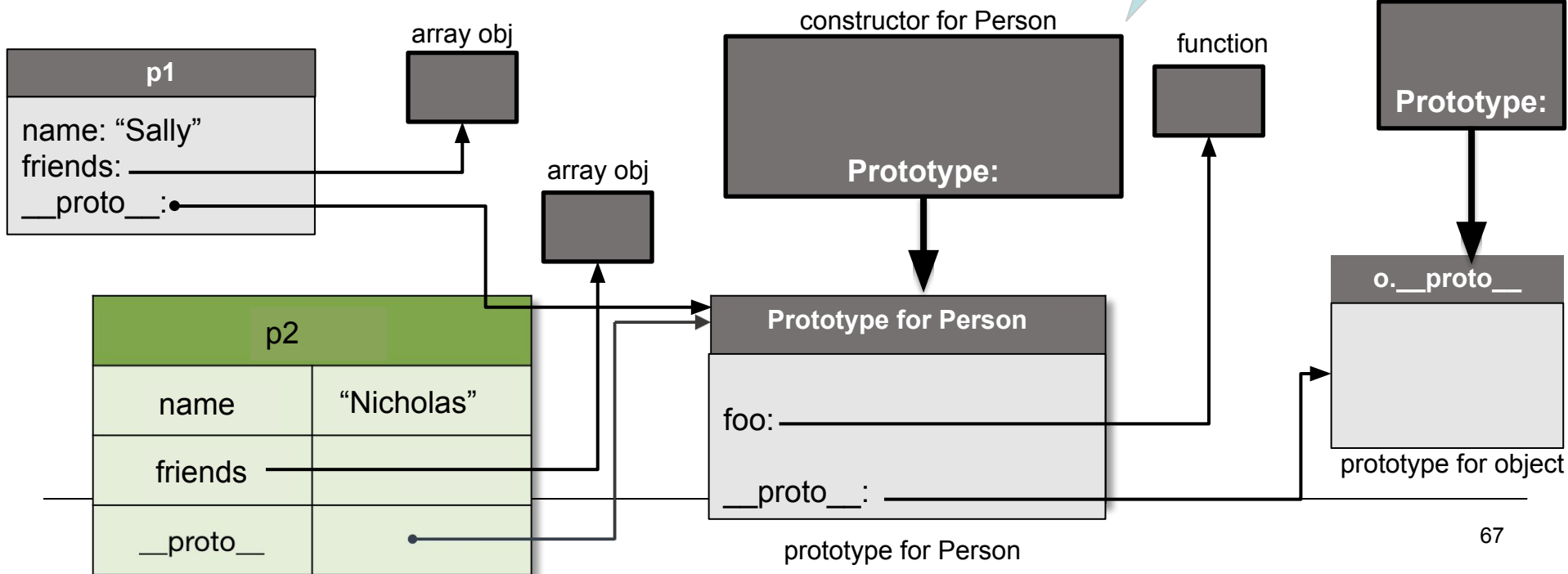
Only modify
your **own** prototypes.
Never modify the
prototypes of standard
JavaScript objects!



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");

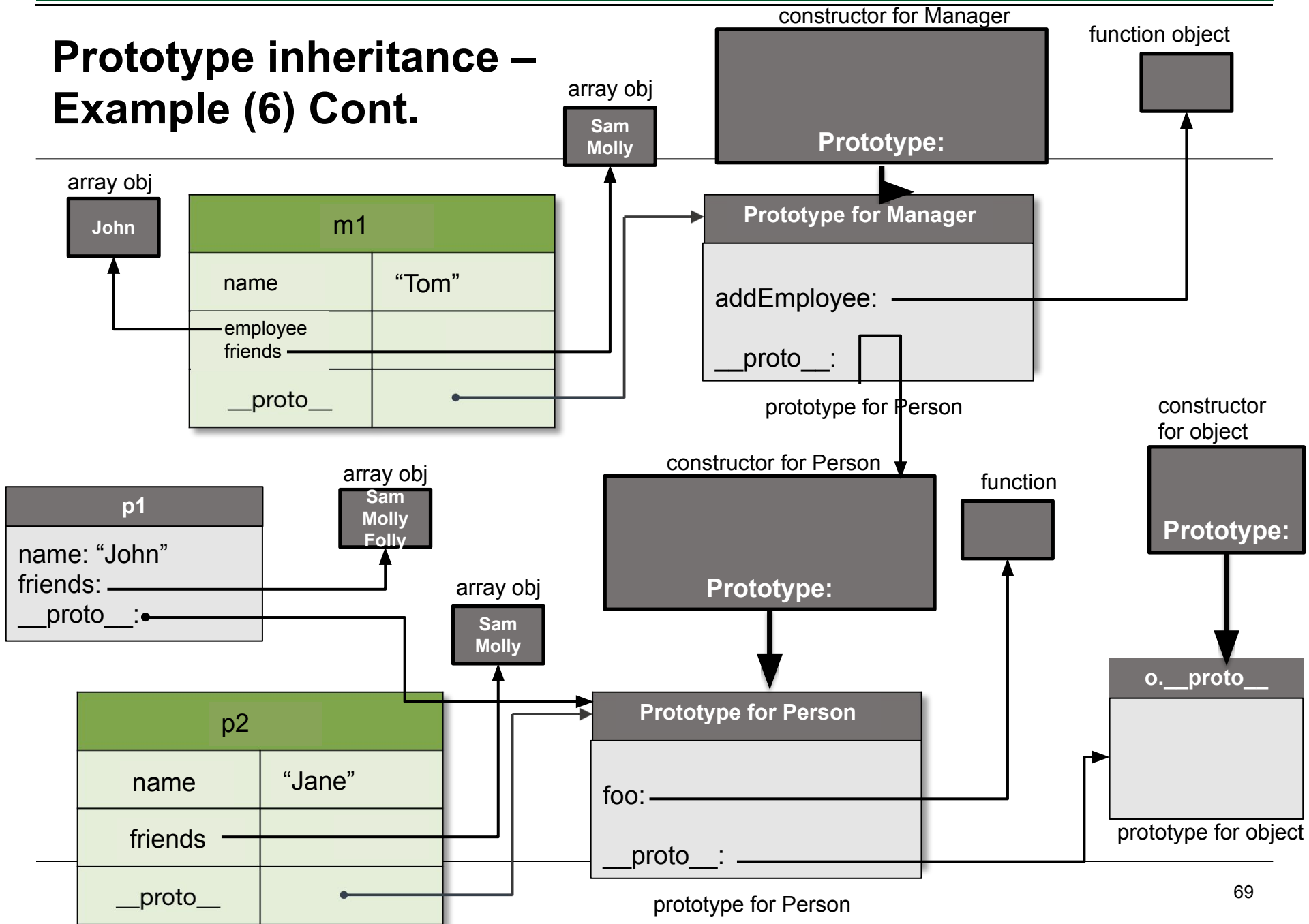
class Manager extends Person {
  constructor(s) {
    super(s);
    this._employee = [];
  }

  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/>