Zaheer ul Hussain Sani

JavaScript

Classes

MOBILE APPLICATION DEVELOPMENT

JavaScript is Prototype-Based

- **Class-based** object-oriented languages, such as Java, C++, C#, are founded on the concept of two distinct entities: **classes and instances**
- **JavaScript** is an **object-based language** based on **prototypes**, rather than being **class-based**
- A **prototype-based language**, such as **JavaScript**, does not make this distinction: it simply has **objects**

Defining a Class

- In **class-based** languages, you define a class in a separate *class definition.*
  - Definition contains Constructors, Instance variables, methods/members
  - Create an object using **new** operator

- **JavaScript** follows a similar model, but does not have a class definition separate from the constructor
  - Instead, you define a constructor function to create objects with a particular initial set of properties and values
  - Any JavaScript function can be used as a constructor
  - You use the **new** operator with a constructor function to create a new object.

Defining a Class

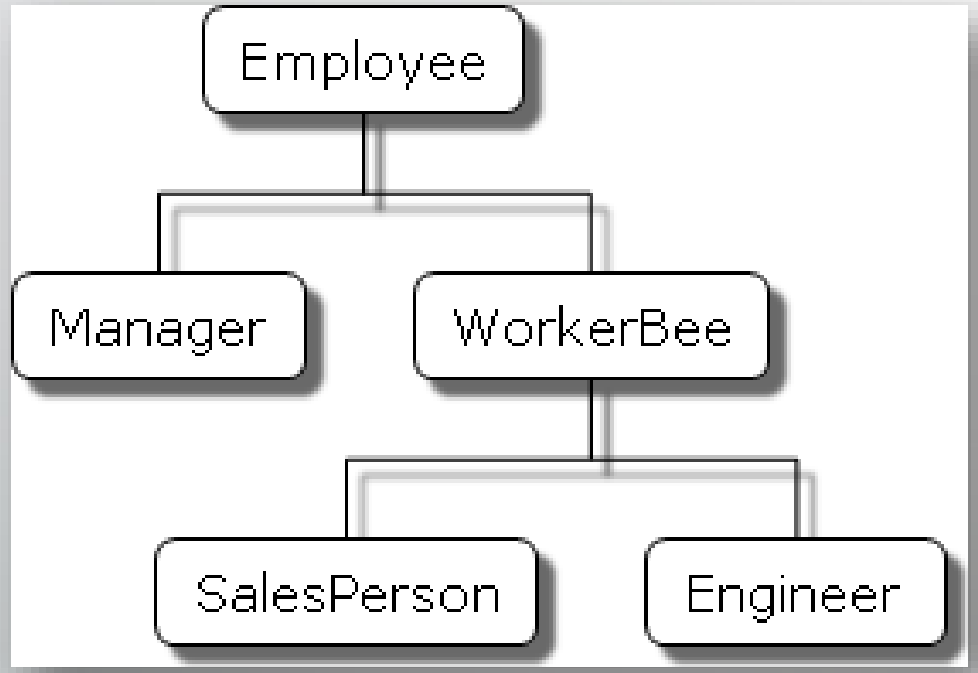
Class-Based (Java/C#)

```
public class Employee {
    public String name = "";
    public String dept = "general";
}
Employee eObj = new Employee();
Console.WriteLine(eObj.dept);
```

Prototype-Based (JavaScript)

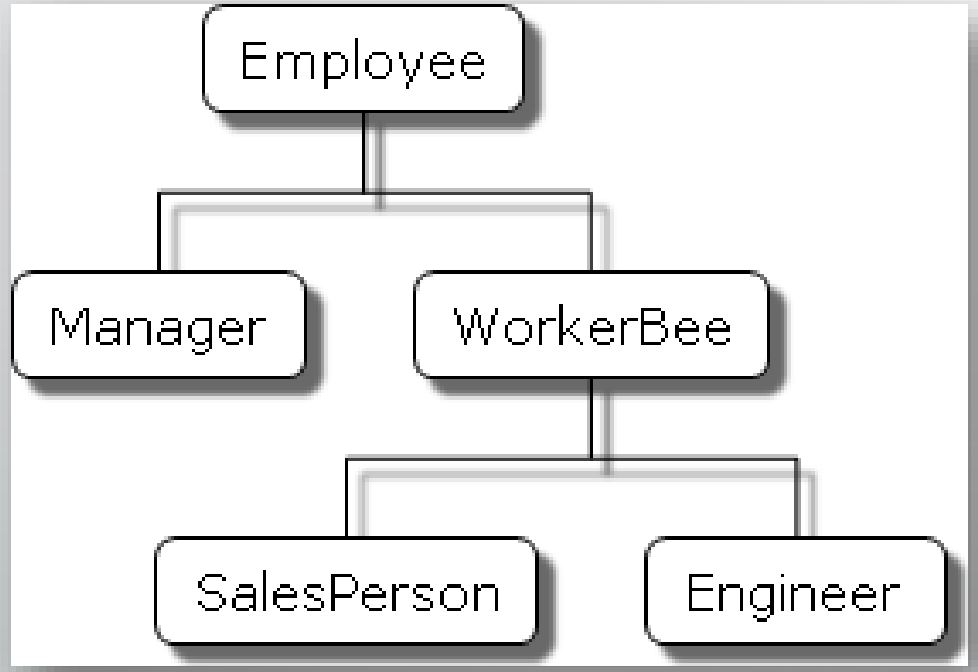
```
function Employee() {
    this.name = '';
    this.dept = 'general';
}
var eObj = new Employee();
console.log(eObj.dept);
```

Inheritance



- There are several ways to define appropriate constructor functions to implement the Employee hierarchy. How you choose to define them depends largely on what you want to be able to do in your application.
- In a real application, you would probably define constructors that allow you to provide property values at object creation time

Inheritance




```
function Employee() {
    this.name = '';
    this.dept = 'general';
}
function Manager() {
    Employee.call(this);
    this.tasks = "task1, task2";
}
Manager.prototype = Employee;

var mObj = new Manager();
console.log(mObj.dept); // general
var eObj = new Employee();
console.log(eObj.dept); // general
```

Object Properties

- `contact` property is associated with only `mObj` of type `Manager`.
- `contact` property is not associated with `Manager` Constructor

```
function Employee() {
    this.name = '';
    this.dept = 'general';
}
function Manager() {
    Employee.call(this);
    this.tasks = "task1, task2";
}
Manager.prototype = Employee;

var mObj = new Manager();
mObj.contact = "+923424324242";
console.log(mObj.dept); // general
console.log(mObj.contact);
```

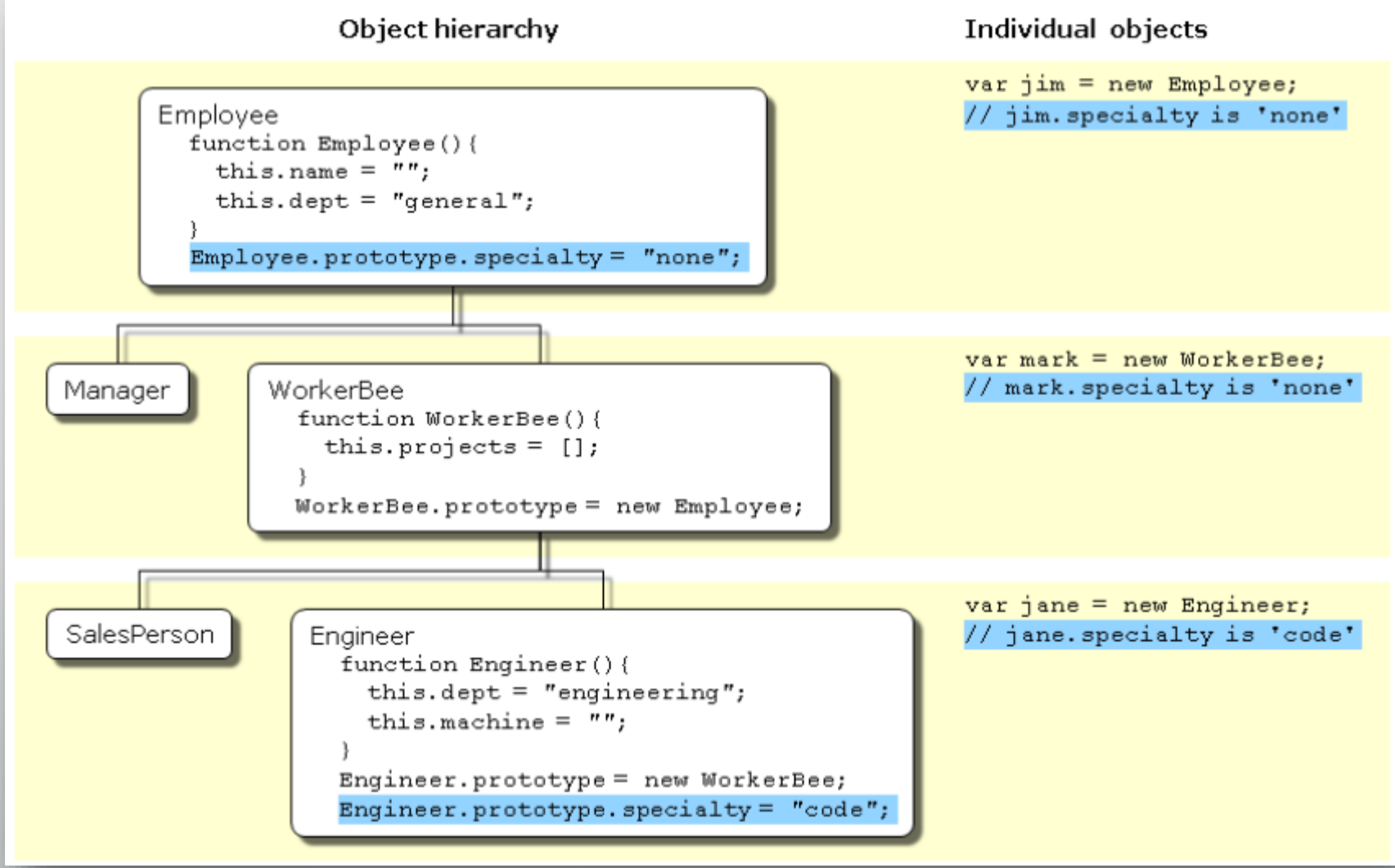
Object Properties

- With the help of prototype, we can define a class level property at runtime

```
Manager.prototype = Employee;
Manager.prototype.contact = "+92000000000";
var mObj = new Manager;
```

```
mObj.
conso   abc ★ prototype
conso   contact
var e   tasks
```

Object Properties

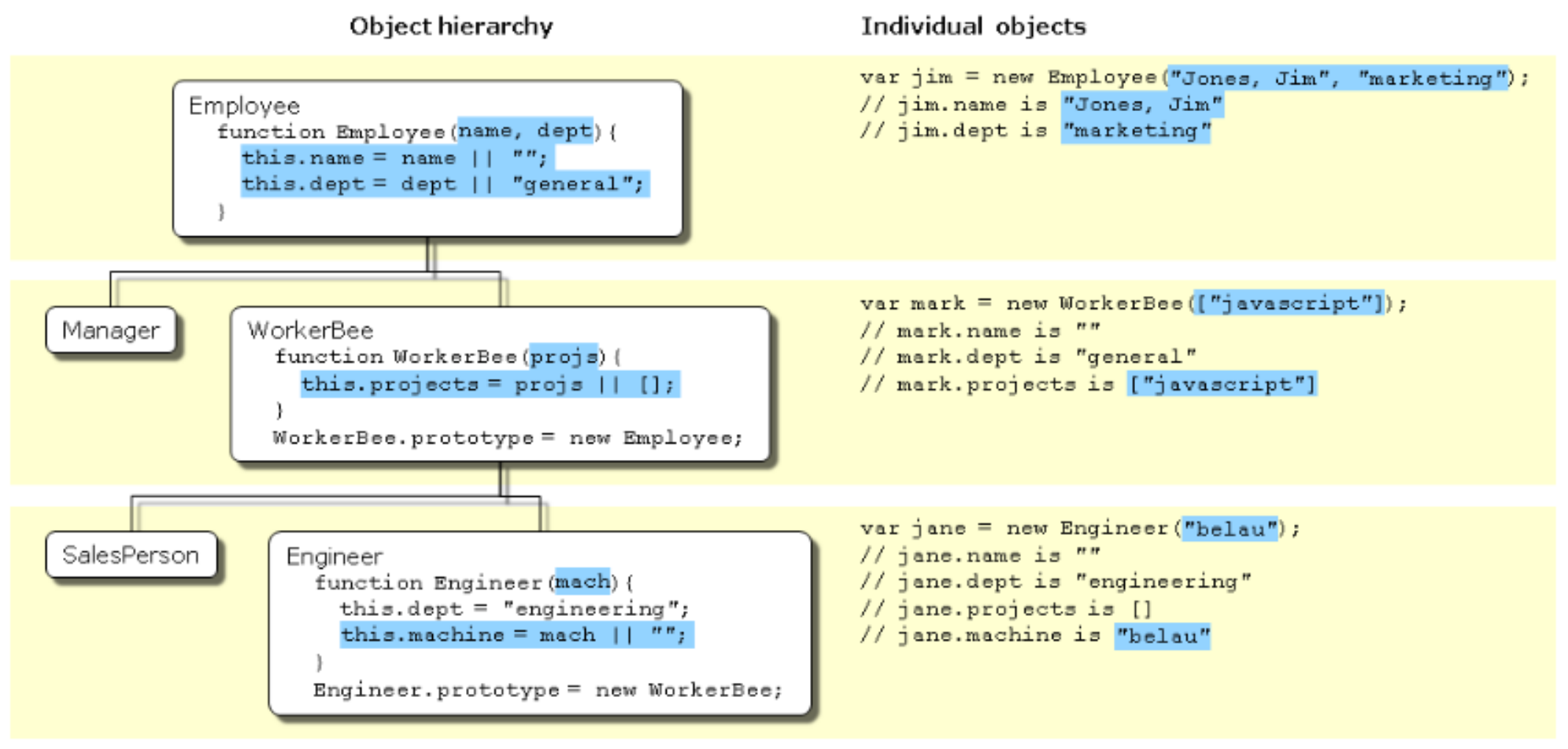


Flexible Constructor

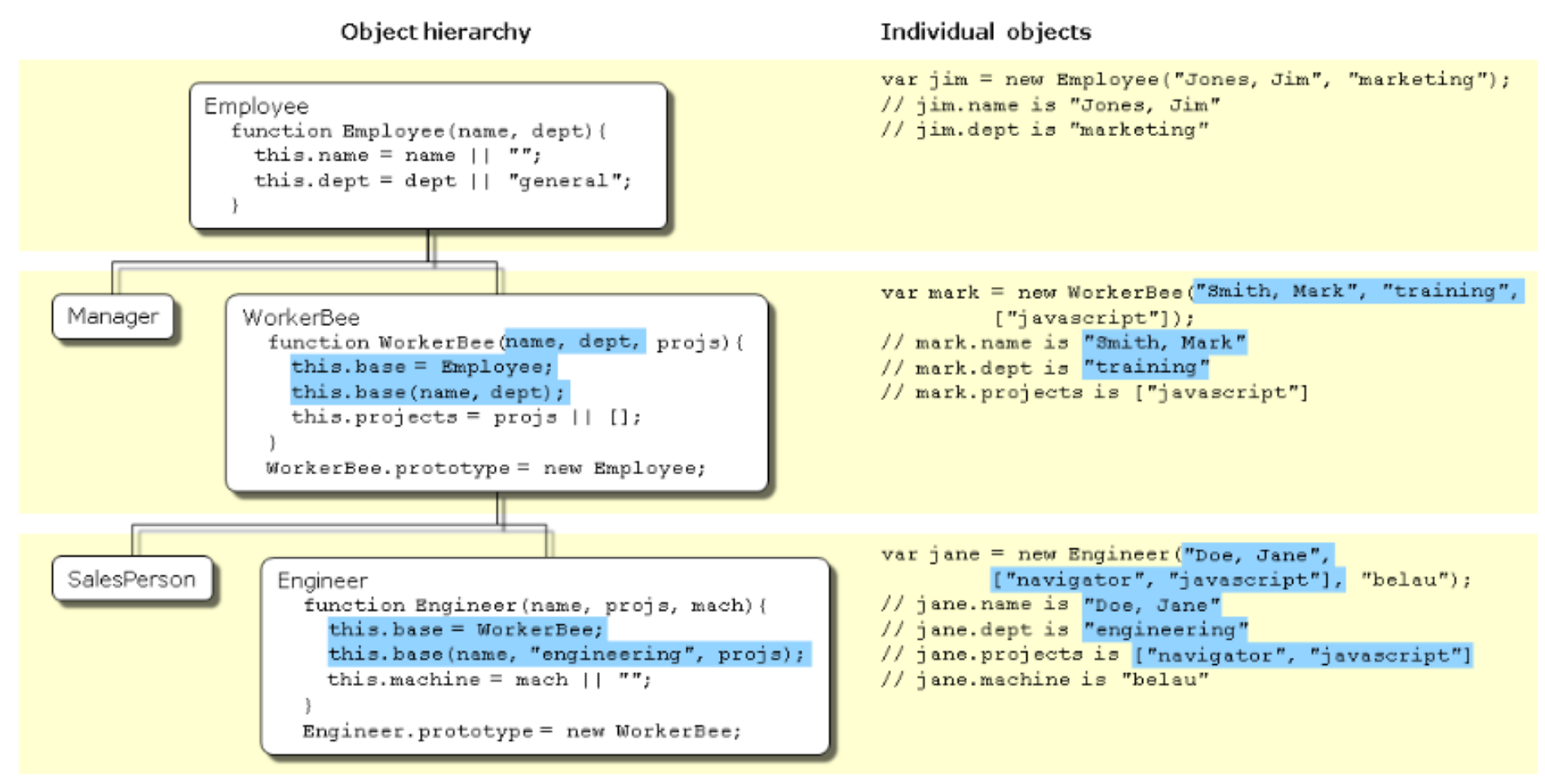
```
function Employee(Name, Department) {
    this.name = Name;
    this.dept = Department;
}
var emp = new Employee("Ahmed Faraz");
console.log(emp.name); // Ahmed Faraz
console.log(emp.dept); // undefined
```

```
function Employee(Name, Department) {
    this.name = Name || '';
    this.dept = Department || 'general';
}
var emp = new Employee("Ahmed Faraz");
console.log(emp.name); // Ahmed Faraz
console.log(emp.dept); // general
```

Flexible Constructor



Initialize Inherited Property – base



Initialize Inherited Property - call

```
function Employee(Name, Department) {
    this.name = Name || 'no name';
    this.dept = Department || 'general';
}
function Manager(Name, Department, Tasks) {
    Employee.call(this);
    this.tasks = Tasks || '';
}
Manager.prototype = Employee;
var manager = new Manager("Zafar Ali", "Services", "Task1, Task2");
console.log(manager.dept); // general
```

Initialize Inherited Property - call

```
function Employee(Name, Department) {
    this.fullName = Name || 'no name';
    this.dept = Department || 'general';
}
function Manager(Name, Department, Tasks) {
    Employee.call(this, Name, Department);
    this.tasks = Tasks || '';
}
Manager.prototype = Employee;
var manager = new Manager("Zafar Ali", "Services", "Task1, Task2");
console.log(manager.fullName); // Zafar Ali
console.log(manager.dept); // Services
```

ES6 Classes

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance
- The class syntax *does not* introduce a new object-oriented inheritance model to JavaScript
- Classes are in fact "special functions", and just as you can define function expressions and function declarations
- Class syntax has two components:
  - class expressions
  - class declarations

Class Declarations

- One way to define a class is using a class declaration. To declare a class, you use the **class** keyword

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

Hoisting (1/4)

- Hoisting is JavaScript's default behavior of moving declarations to the top.
- In JavaScript, a variable can be declared after it has been used.
- In other words; a variable can be used before it has been declared.

```
x = 5;
var x;
console.log(x); // 5
```

```
var x;
x = 5;
console.log(x); // 5
```

```
fName = "Ali"
lName = "Zafar"
var fName, lName
console.log(fName, lName); // Ali Zafar
```

```
var fName, lName
fName = "Ali"
lName = "Zafar"
console.log(fName, lName); // Ali Zafar
```

Hoisting (2/4)

- Variables and constants declared with **`let`** or **`const`** are not hoisted!

```
x = 5
let x
console.log(x)
// ReferenceError: Cannot access 'x' before initialization
```

```
x = 5
const x
console.log(x)
// SyntaxError: Missing initializer in const declaration
```

Hoisting (3/4)

- JavaScript only hoists declarations, not initializations.
- only the declaration (var y), not the initialization (=7) is hoisted to the top.

```
var x = 5; // Initialize x
var y = 7; // Initialize y
console.log(x + y) // 12
```

```
var x = 5; // Initialize x
console.log(x + y) // NaN
var y = 7; // Initialize y
```

```
var x = 5;  // Initialize x
var y;      // Declare y
console.log(x + y); // NaN
y = 7;      // Initialize y
```

```
var x = 5;  // Initialize x
y = 7;      // Initialize y
console.log(x + y); // 12
var y;      // Declare y
```

Hoisting (4/4)

- An important difference between function declarations and class declarations is that **function declarations are hoisted** and class declarations are not

```
var rect = new Rectangle();
class Rectangle { }
// ReferenceError
```

```
var sqr = new Square();
function Square() { }
```

Class Expression

- A **class expression** is another way to define a class. Class expressions can be named or unnamed

```
// unnamed
let Rectangle = class {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};
console.log(Rectangle.name);
// output: "Rectangle"

// named
let Rectangle = class Rectangle2 {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};
console.log(Rectangle.name);
// output: "Rectangle2"
```

Constructor

- The constructor method is a special method for creating and initializing an object created with a class
- There can only be one special method with the name "**`constructor`**" in a class
- A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.
- A constructor can use the **`super`** keyword to call the constructor of the super class.

Getters, Methods

- Get keyword is used for creating Getter
  - Getter is called as a property or field name instead of function
  - E.g. **`square.area`** and not ~~**`square.area()`**~~
- Whereas, Method are declared without function keyword and can be called as a function
  - E.g. **square.calcArea()**

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    // Getter
    get area() {
        return this.calcArea();
    }
    // Method
    calcArea() {
        return this.height * this.width;
    }
}

const square = new Rectangle(10, 10);

console.log(square.area); // 100
```

Getters, Methods – Another Example

- Height is a getter and used as rect.Height
- setHeight is a setter and used as rect.Height(20)

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    get Height() {
        return this.height;
    }
    setHeight(height) {
        this.height = height;
    }
}

var rect = new Rectangle(10, 15);
console.log(rect.Height); // 10

rect.setHeight(20);

console.log(rect.Height); // 20
```

Static Method

- The static keyword defines a static method for a class
- Static methods are called without instantiating their class and cannot be called through a class instance
- Static methods are often used to create **utility functions** for an application.

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.hypot(dx, dy);
    }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
p1.distance; //undefined
p2.distance; //undefined

console.log(Point.distance(p1, p2));
// 7.0710678118654755
```

Instance Properties vs Static

- Instance properties must be defined inside of class methods:

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

- Static (class-side) data properties and prototype data properties must be defined outside of the Class Body declaration:

```
// Access with Class Name
Rectangle.staticWidth = 20;
// Access through class object
Rectangle.prototype.prototypeWidth = 25;

console.log(Rectangle.staticWidth); // 20
var rect = new Rectangle();
console.log(rect.prototypeWidth); // 25
```

Public Field Declarations

```
class Rectangle {
    height = 0;
    width;
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    get Height() {
        return this.height;
    }
    setHeight(height) {
        this.height = height;
    }
}

var rect = new Rectangle(10, 15);
console.log(`Height of Rectangle is ${rect.height} cm`)
// Height of Rectangle is 10 cm
console.log(rect.Height); // 10
rect.setHeight(20);
console.log(rect.Height); // 20
```

Private Field Declarations

```
class Rectangle {
    #height = 0;
    #width;
    constructor(height, width) {
        this.#height = height;
        this.#width = width;
    }
    get Height() {
        return this.#height;
    }
    setHeight(height) {
        this.#height = height;
    }
}

var rect = new Rectangle(10, 15);
console.log(`Height of Rectangle is ${rect.height} cm`)
// rect.height is undefined
console.log(rect.Height); // 10
rect.setHeight(20);
console.log(rect.Height); // 20
```

Inheritance

- The **extends** keyword is used in class declarations or class expressions to create a class as a child of another class.
- If there is a constructor present in the subclass, it needs to first call super() before using "this".

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Dog extends Animal {
    constructor(name) {
        super(name);  // call the super class constru
ctor and pass in the name parameter
    }
    speak() {
        console.log(`${this.name} barks.`);
    }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

Browser compatibility

Update compatibility data on GitHub

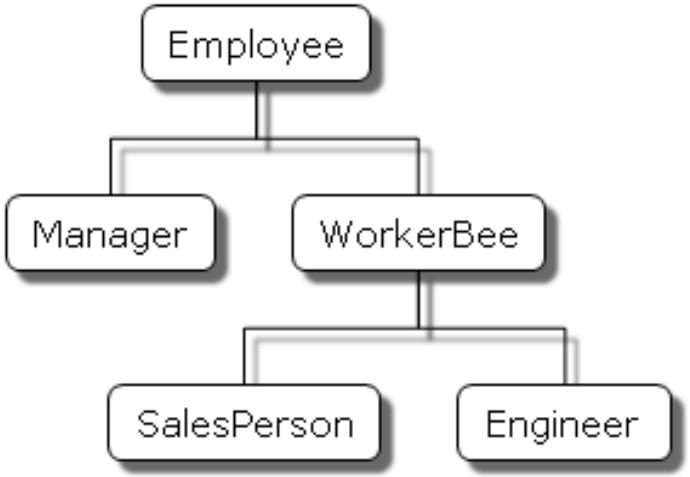
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
`classes`	49 *	13	45	No	36 *	9	49 *	49 *	45	36 *	9	5.0 *	6.0.0
`constructor`	49 *	13	45	No	36 *	9	49 *	49 *	45	36 *	9	Yes	6.0.0
`extends`	49 *	13	45	No	36 *	9	49 *	49 *	45	36 *	9	Yes	6.0.0
Private class fields	74	No	No	No	62	No	74	74	No	53	No	No	12.0.0
Public class fields	72	No	69	No	60	No	72	72	No	51	No	No	12.0.0
`static`	49 *	13	45	No	36 *	9	49 *	49 *	45	36 *	9	Yes	6.0.0
Static class fields	72	No	No *	No	60	No	72	72	No	51	No	No	12.0.0

What are we missing?

- Full support
- No support
- * See implementation notes.
- User must explicitly enable this feature.

Exercise: Implement following scenario in ES6 Classes

A simple object hierarchy with the following objects:



- `Employee` has the properties `name` (whose value defaults to the empty string) and `dept` (whose value defaults to "general").
- `Manager` is based on `Employee`. It adds the `reports` property (whose value defaults to an empty array, intended to have an array of `Employee` objects as its value).
- `WorkerBee` is also based on `Employee`. It adds the `projects` property (whose value defaults to an empty array, intended to have an array of strings as its value).
- `SalesPerson` is based on `WorkerBee`. It adds the `quota` property (whose value defaults to 100). It also overrides the `dept` property with the value "sales", indicating that all salespersons are in the same department.
- `Engineer` is based on `WorkerBee`. It adds the `machine` property (whose value defaults to the empty string) and also overrides the `dept` property with the value "engineering".

References

- MDN Web Docs: Details of the object model (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model)
- MDN Web Docs: Classes (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)
- W3Schools React ES6 (https://www.w3schools.com/react/react_es6.asp)