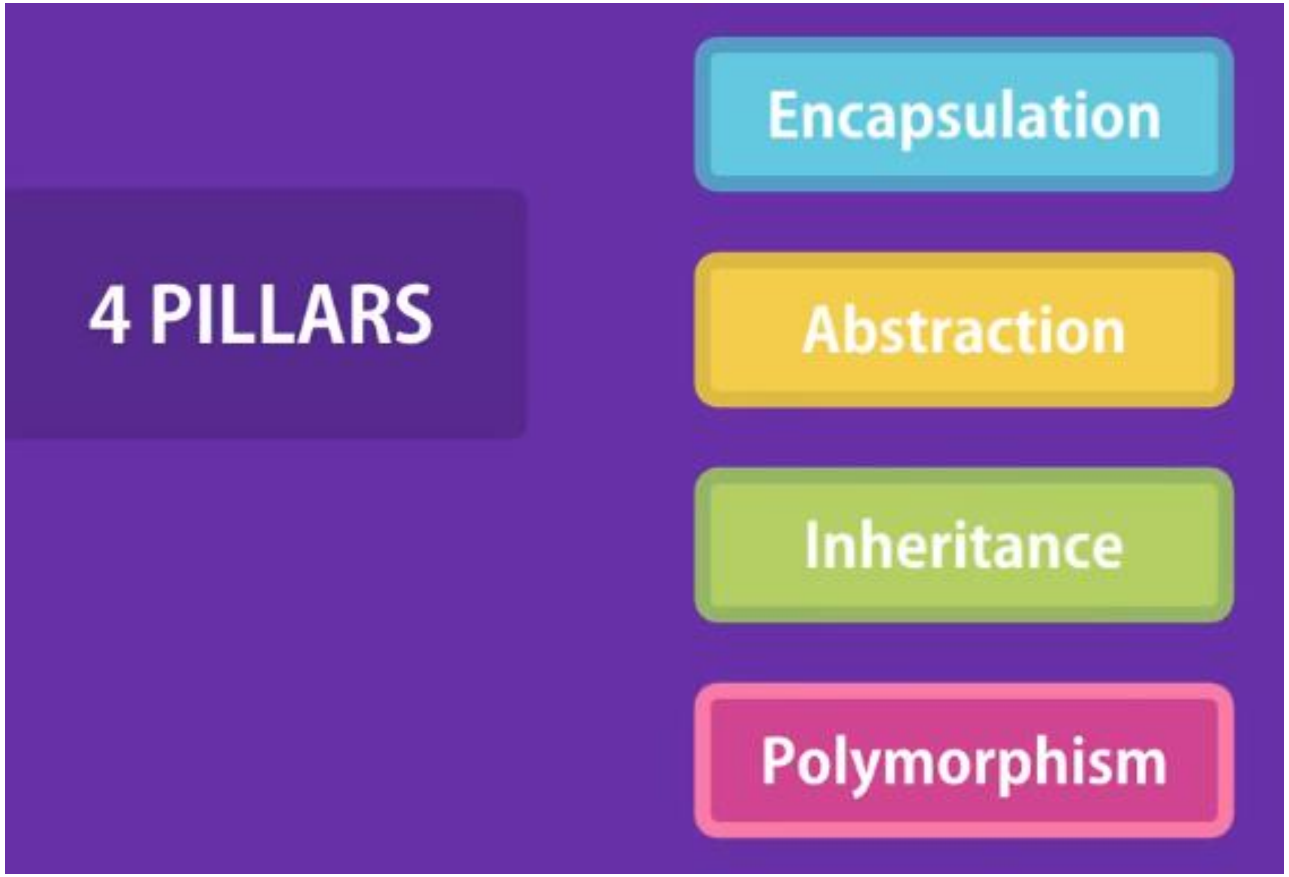
Node

JS OOP

4 Pillars of OOP



4 PILLARS

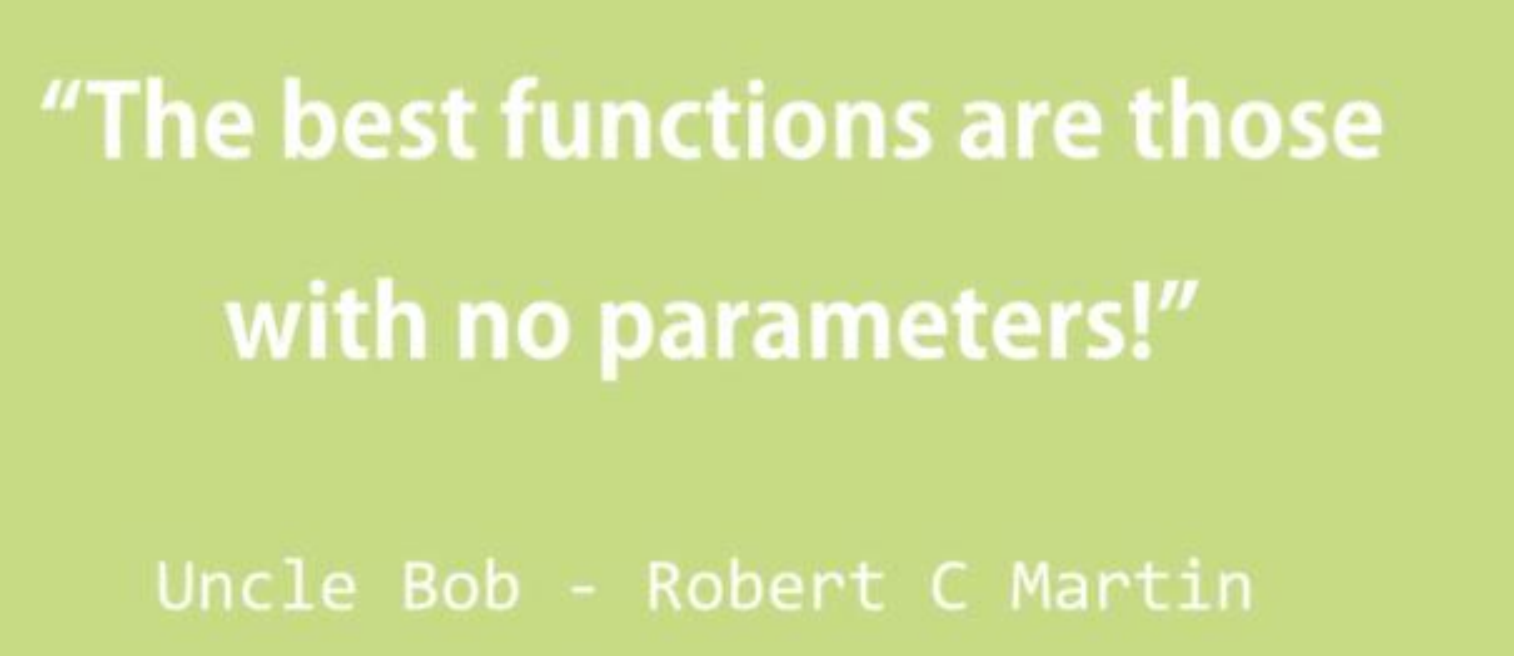
Encapsulation

Abstraction

Inheritance

Polymorphism

Encapsulation



“The best functions are those with no parameters!”

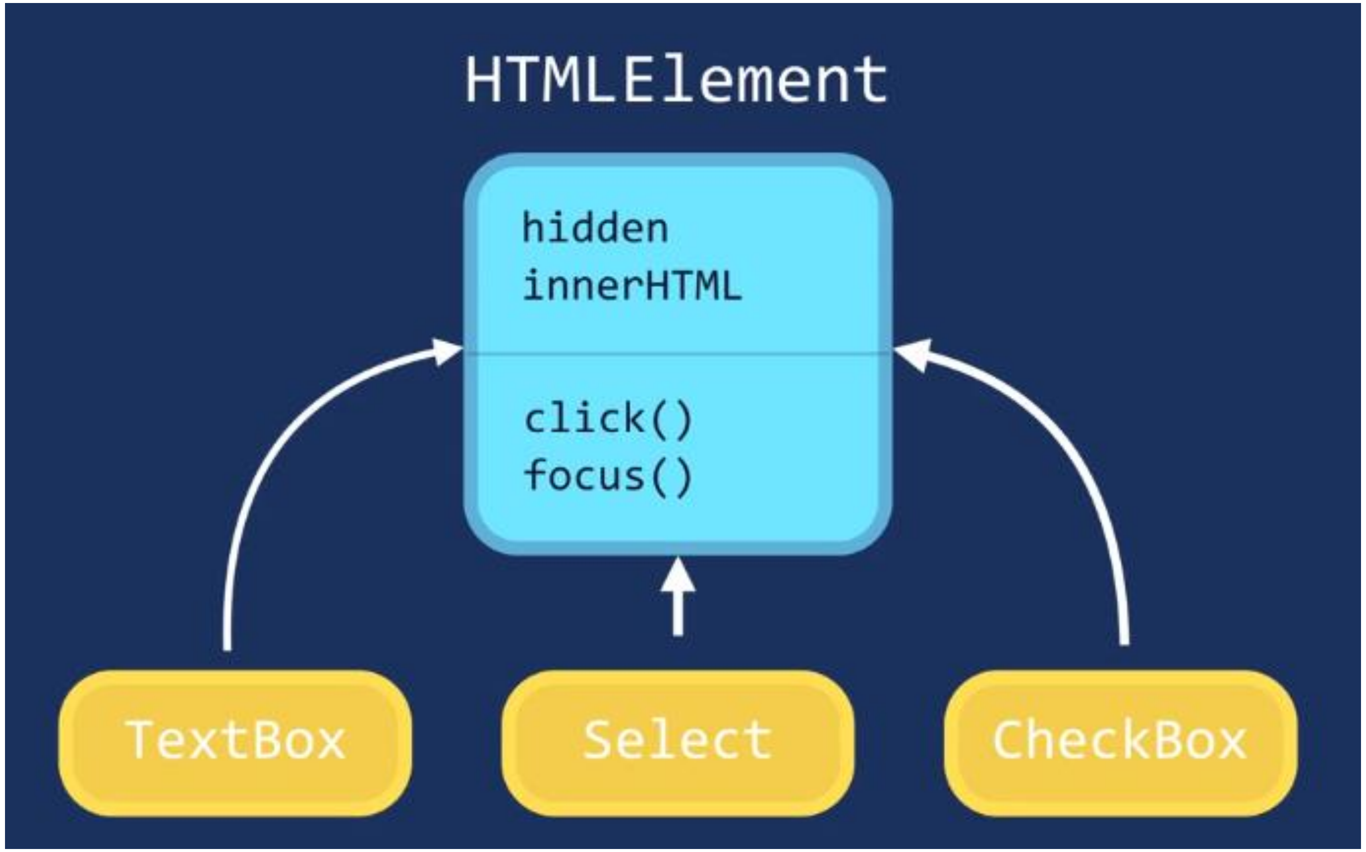
Uncle Bob - Robert C Martin

Abstraction

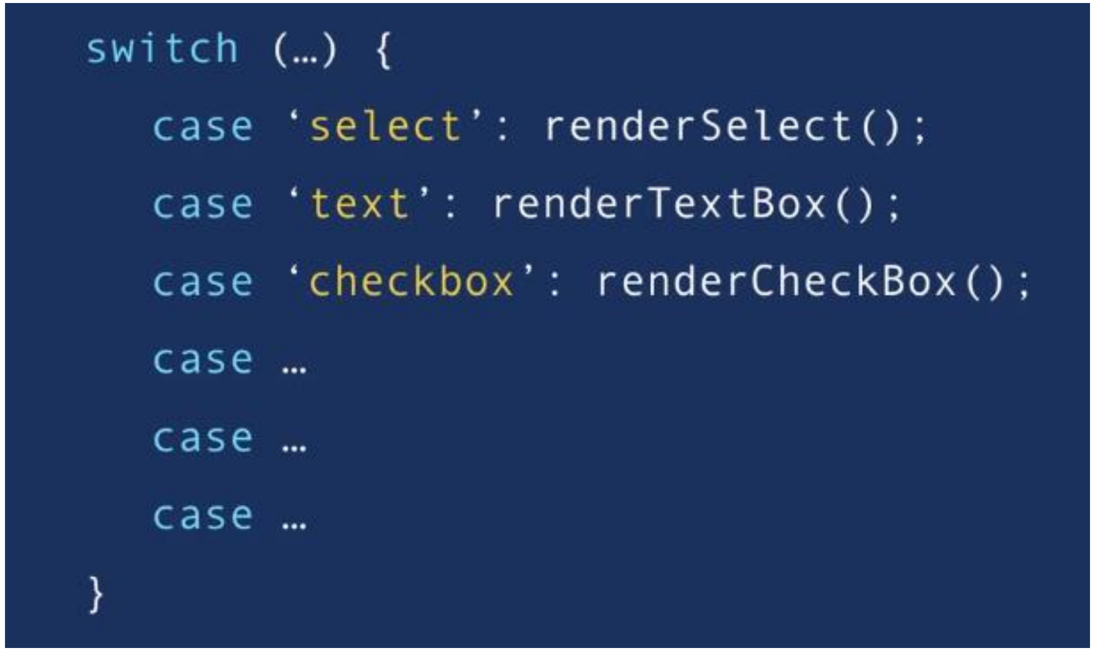
Hide Complex Implementation Details

- Clean your interface

Inheritence

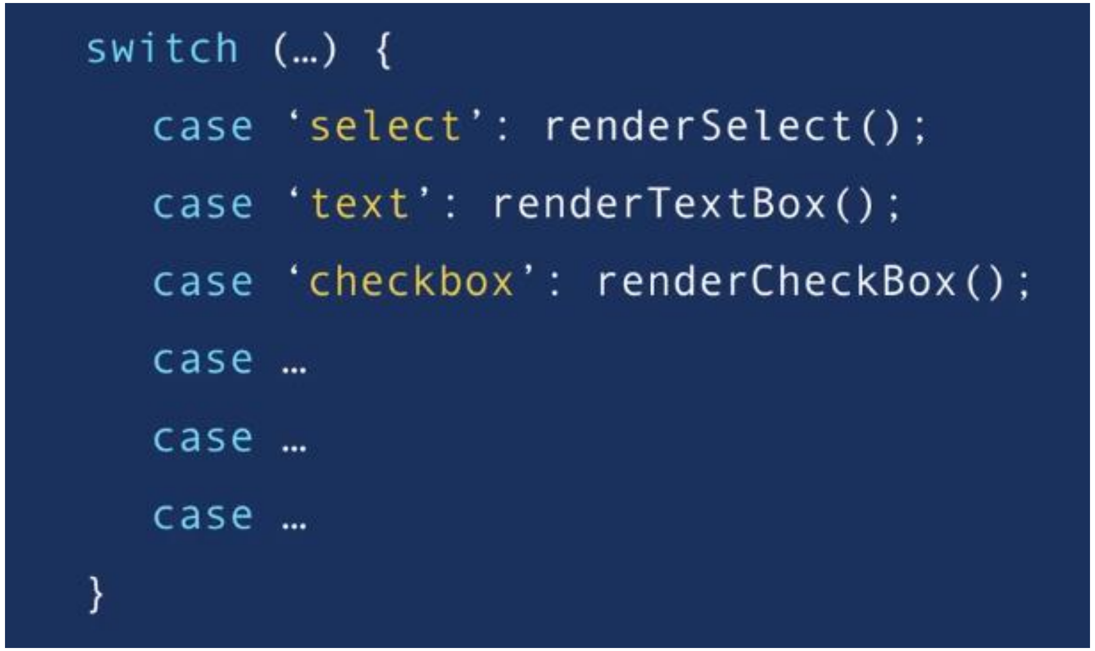


Poly Morphism

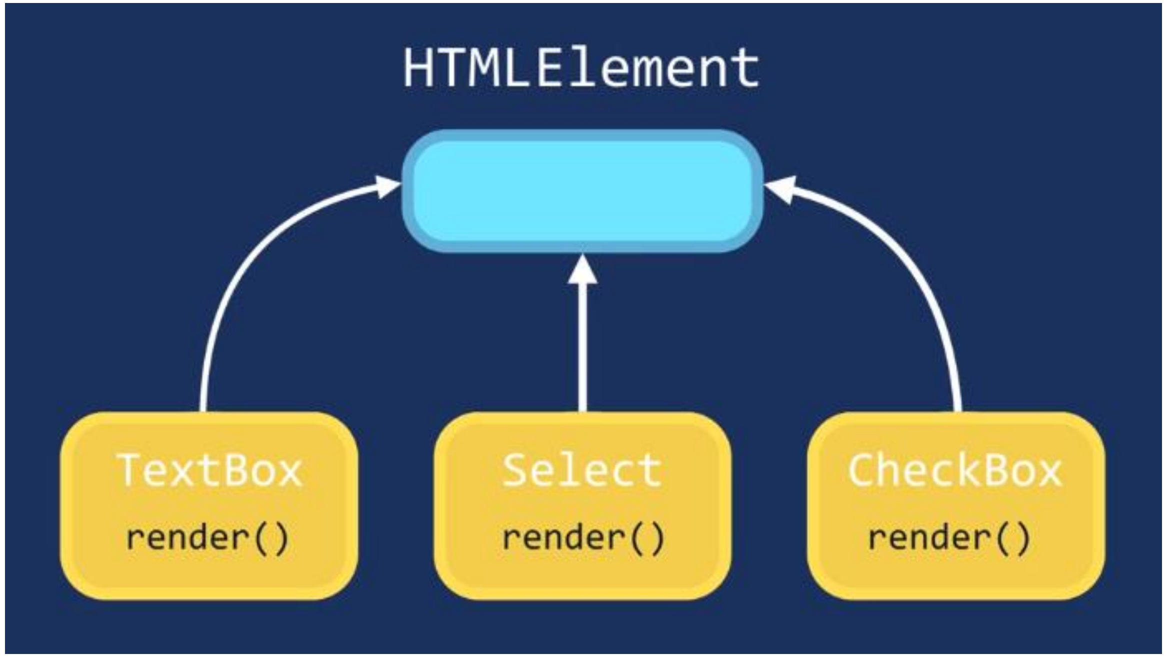


```
switch (…) {
   case ‘select’: renderSelect();
   case ‘text’: renderTextBox();
   case ‘checkbox’: renderCheckBox();
   case …
   case …
   case …
}
```

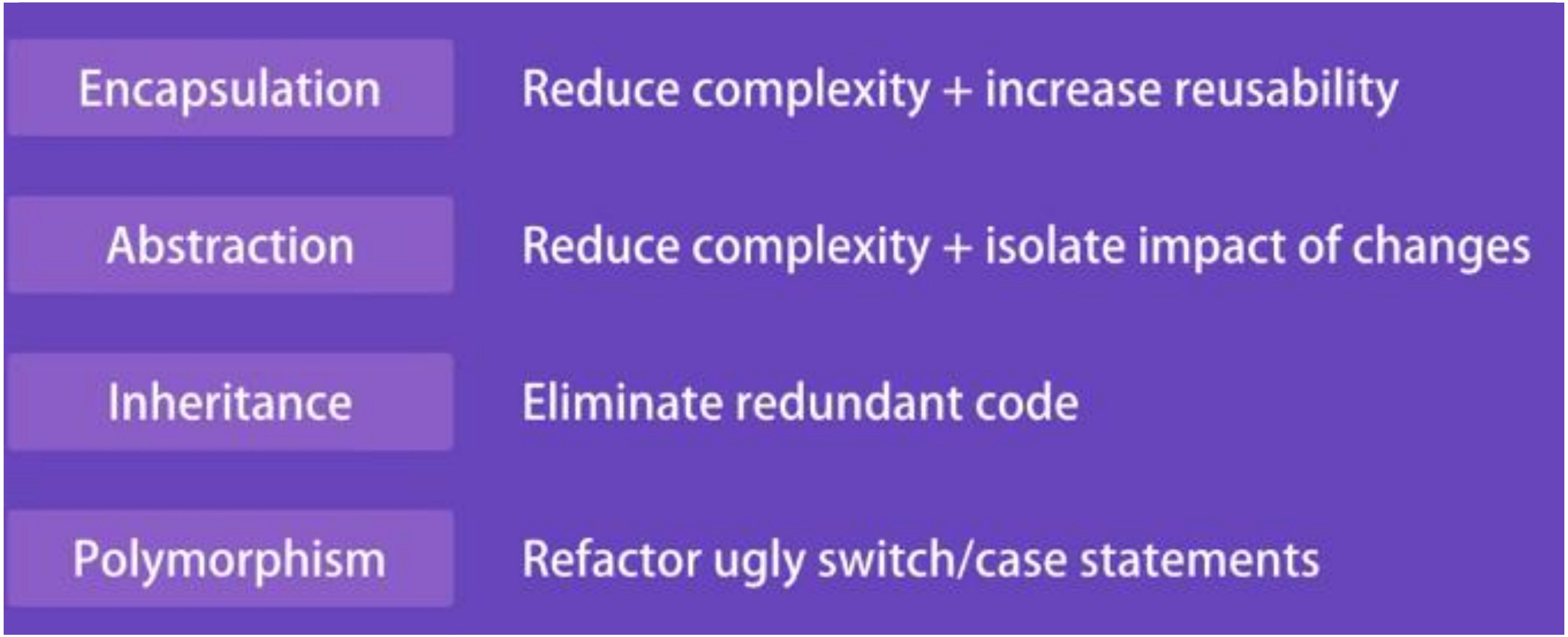
Poly Morphism



```
switch (…) {
   case ‘select’: renderSelect();
   case ‘text’: renderTextBox();
   case ‘checkbox’: renderCheckBox();
   case …
   case …
   case …
}
```

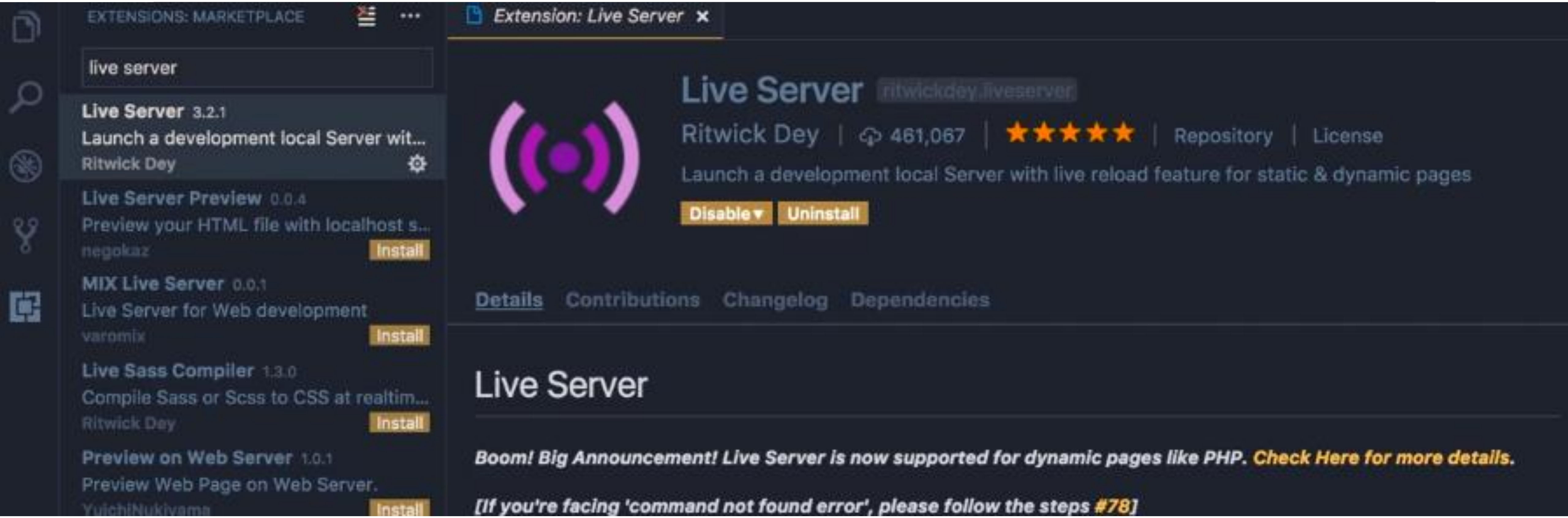


Why OOP



Encapsulation	Reduce complexity + increase reusability
Abstraction	Reduce complexity + isolate impact of changes
Inheritance	Eliminate redundant code
Polymorphism	Refactor ugly switch/case statements

Development Environment



Live Server 3.2.1
Launch a development local Server wit...
Ritwick Dey

Live Server Preview 0.0.4
Preview your HTML file with localhost s...
negokaz

MIX Live Server 0.0.1
Live Server for Web development
varomix

Live Sass Compiler 1.3.0
Compile Sass or Scss to CSS at realtim...
Ritwick Dey

Preview on Web Server 1.0.1
Preview Web Page on Web Server.
YuichiNukiyama

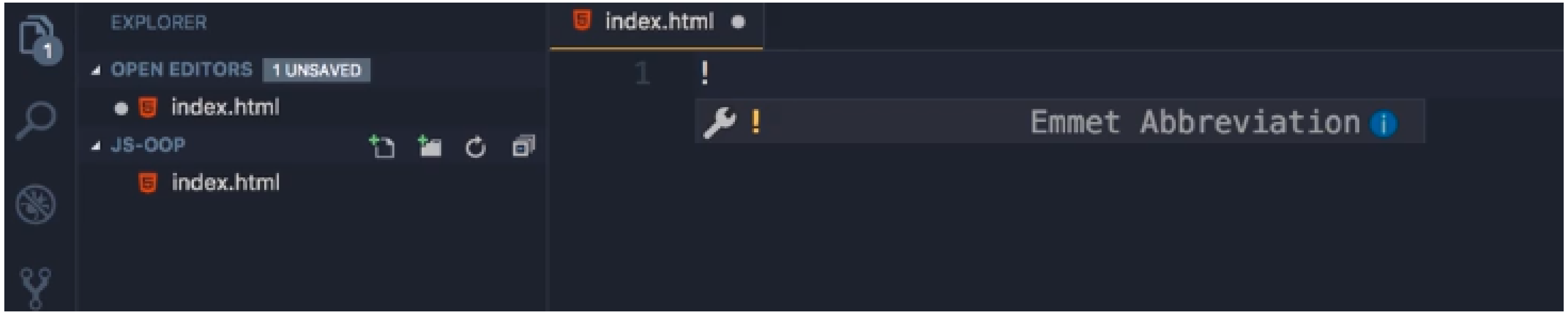
Live Server ritwickdey.liveserver
Ritwick Dey | 461,067 | ★★★★★ | Repository | License
Launch a development local Server with live reload feature for static & dynamic pages

Live Server

Boom! Big Announcement! Live Server is now supported for dynamic pages like PHP. Check Here for more details.

[If you're facing 'command not found error', please follow the steps #78]

Live Templates in CS Code (!) press tab



```
!
```

! Emmet Abbreviation

Object Literal

```
let circle = {
radius: 1,
border: 2,
}
```

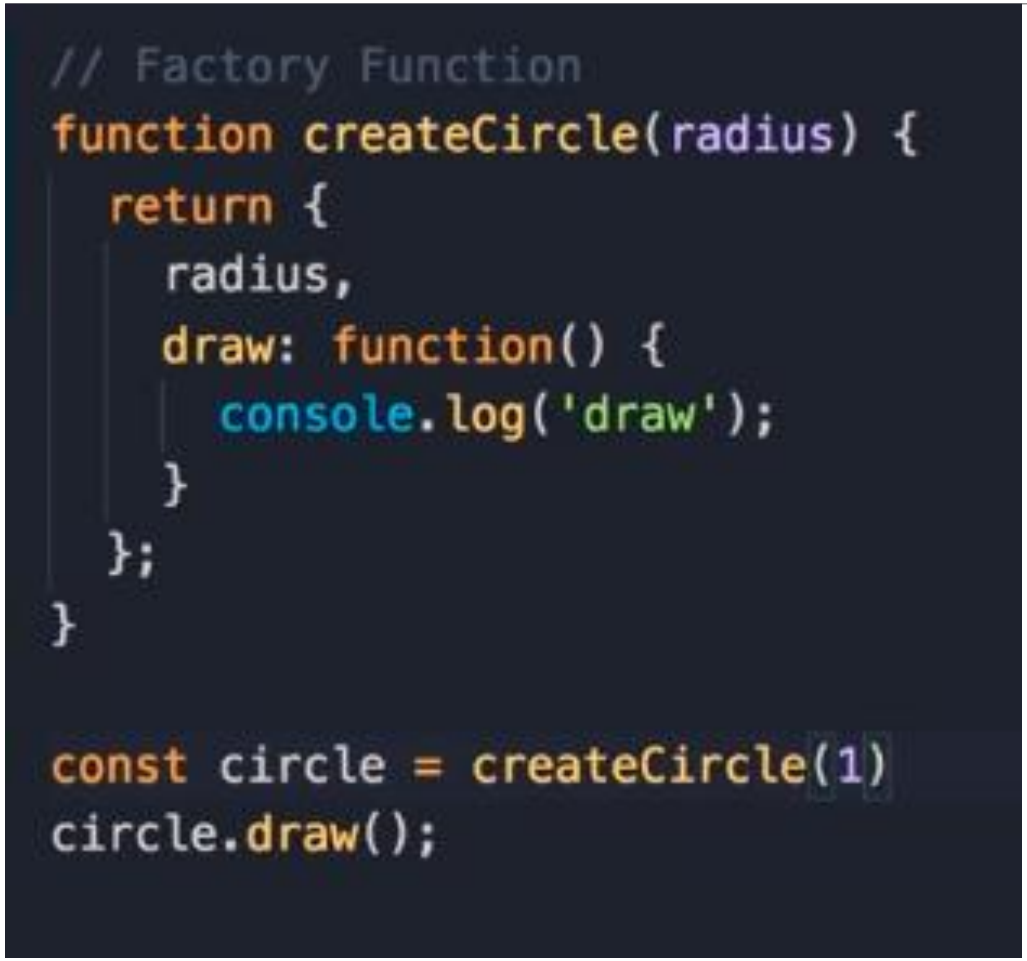
Object Literal

```
let circle = {
radius: 1,
border: 2,
location: {
  x: 45,
  y: 35
 }
}
```

Object Literal

```
let circle = {
radius: 1,
draw: function () {
console.log('draw');
}
}
circle.draw();
```

Factory Function



```
// Factory Function
function createCircle(radius) {
  return {
    radius,
    draw: function() {
      console.log('draw');
    }
  };
}

const circle = createCircle(1)
circle.draw();
```

Constructor Function

```
function Circle(radius) {
 this.radius = radius;
 this.draw = function () {
  console.log("Draw: r=" + radius);
 }
}
const c = new Circle(5); //new Object
c.draw();
```

Don’t Miss It

this

Referes to the object calling current function

Constructor property

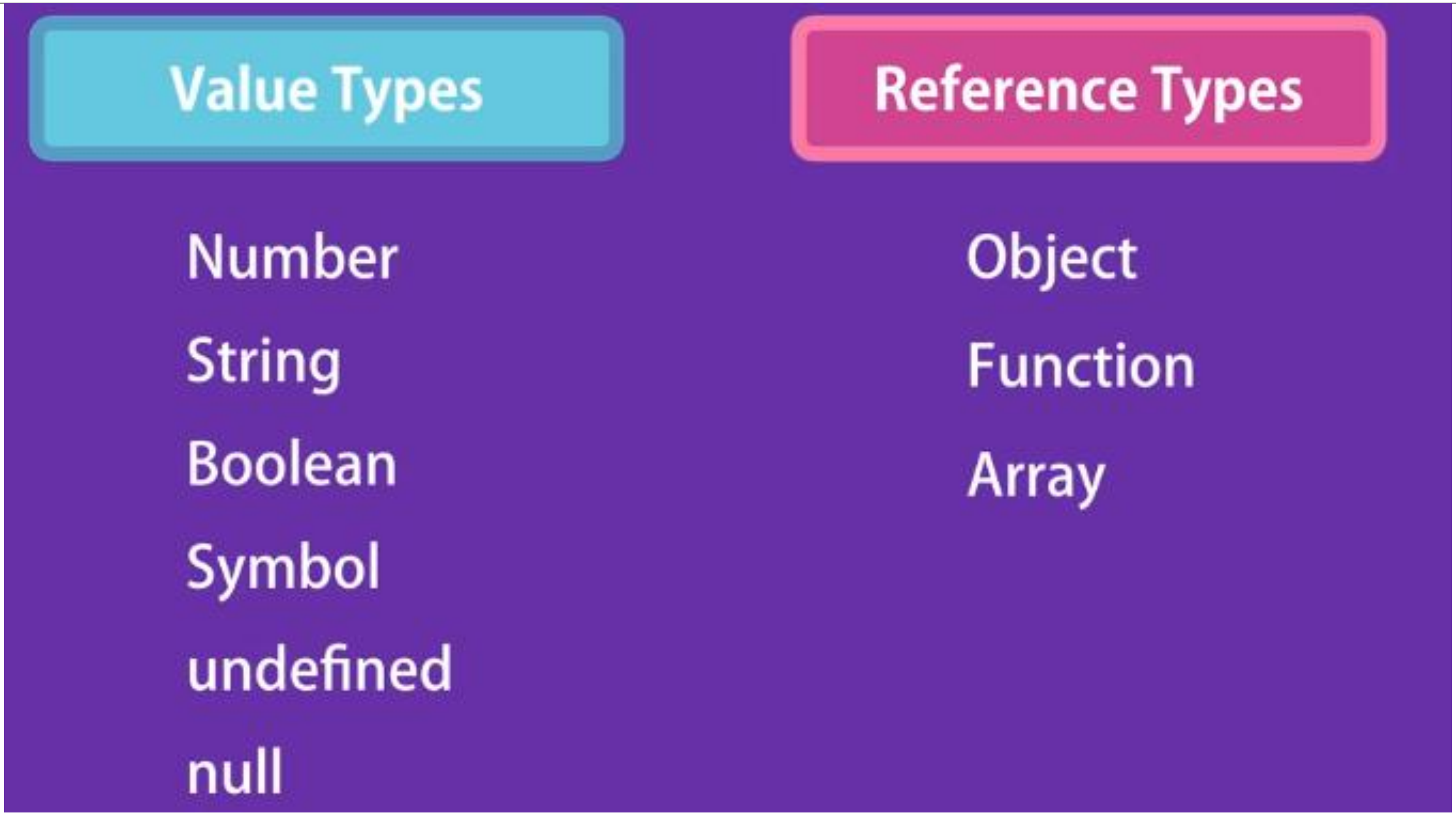
```
let x = {}
// let x= new Object()
```

//factory functions use default constructor

//check from browser by

object.constructor

Value vs Reference Types

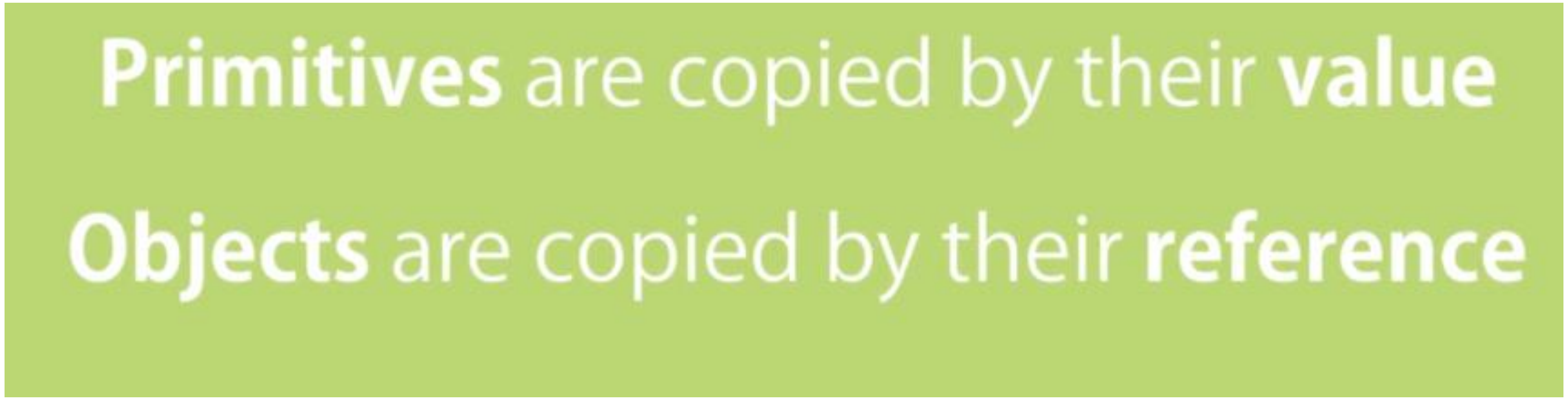


Value Types	Reference Types
Number	Object
String	Function
Boolean	Array
Symbol	
undefined	
null	

Value vs Reference Types

```
let x = 10;

let y = x;

x = 20;

//y will have 10
```

```
let x = {value:10}

let y = x;

x.value = 20;

//y.value will have 20
```



Primitives are copied by their **value**

Objects are copied by their **reference**

What will be the output

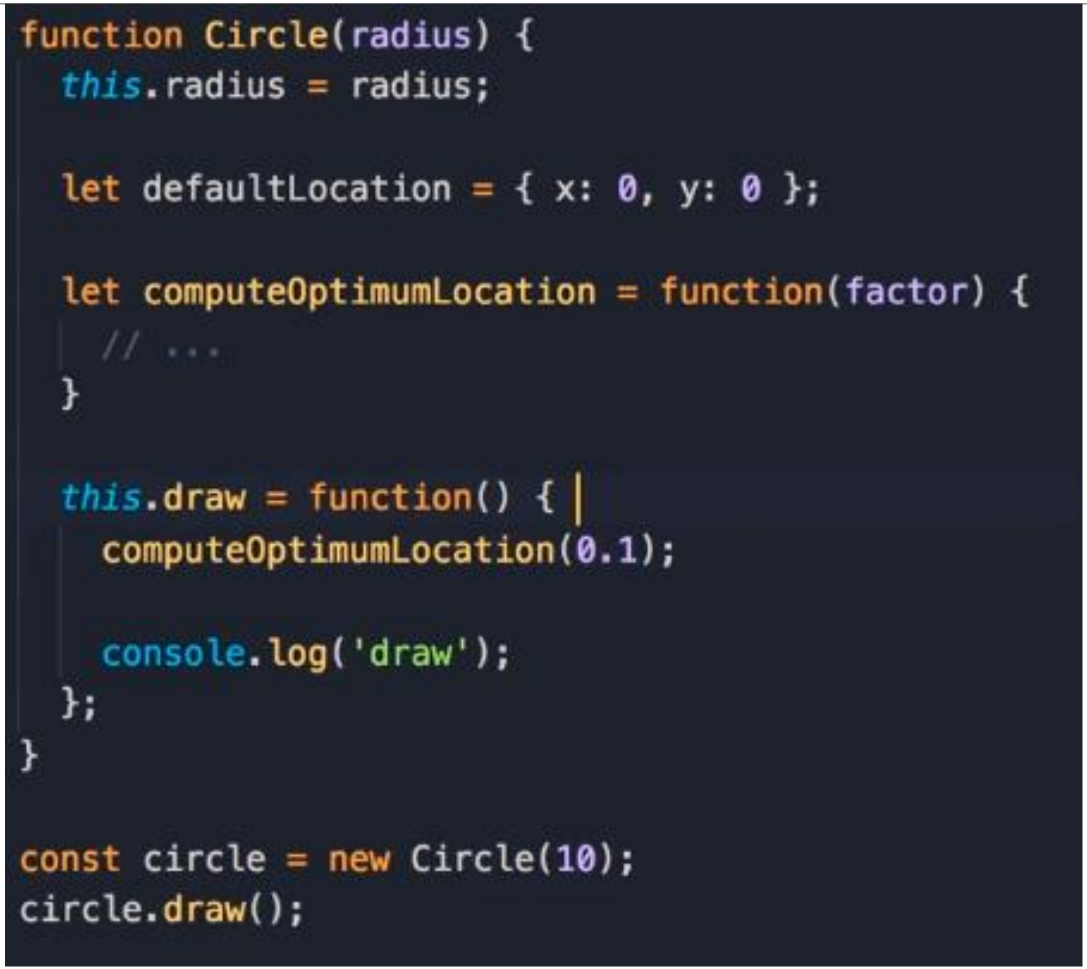
```
let x = 10;
function increase(x) {
x++;
}
console.log(x);
//10
```

```
let y = { value: 10 };
function increaseObj(y) {
y.value++;
}
increaseObj(y);
console.log(y.value);
//11
```

Loop Through keys

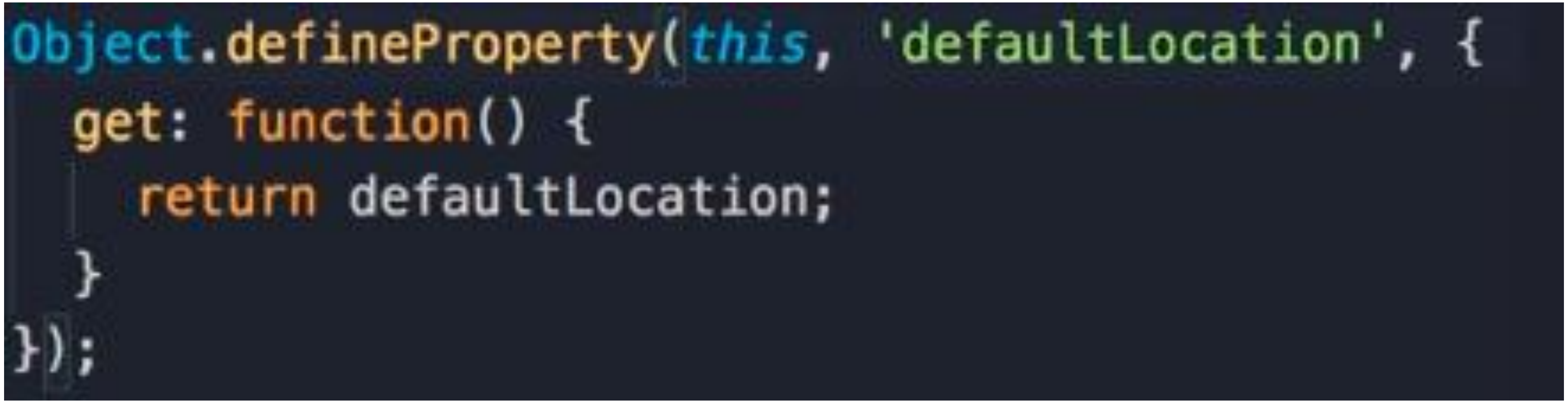
```
function Circle(radius) {
this.radius = radius;
this.draw = function () {
console.log("Draw: r=" + radius);
}
}
const c = new Circle(5);
for (let key in c) {
console.log(key, c[key]);
}
```

Private Properties And Methods



```
function Circle(radius) {
  this.radius = radius;

  let defaultLocation = { x: 0, y: 0 };

  let computeOptimumLocation = function(factor) {
    // ...
  }

  this.draw = function() {
    computeOptimumLocation(0.1);

    console.log('draw');
  };
}

const circle = new Circle(10);
circle.draw();
```

Getter (Computed Properties)

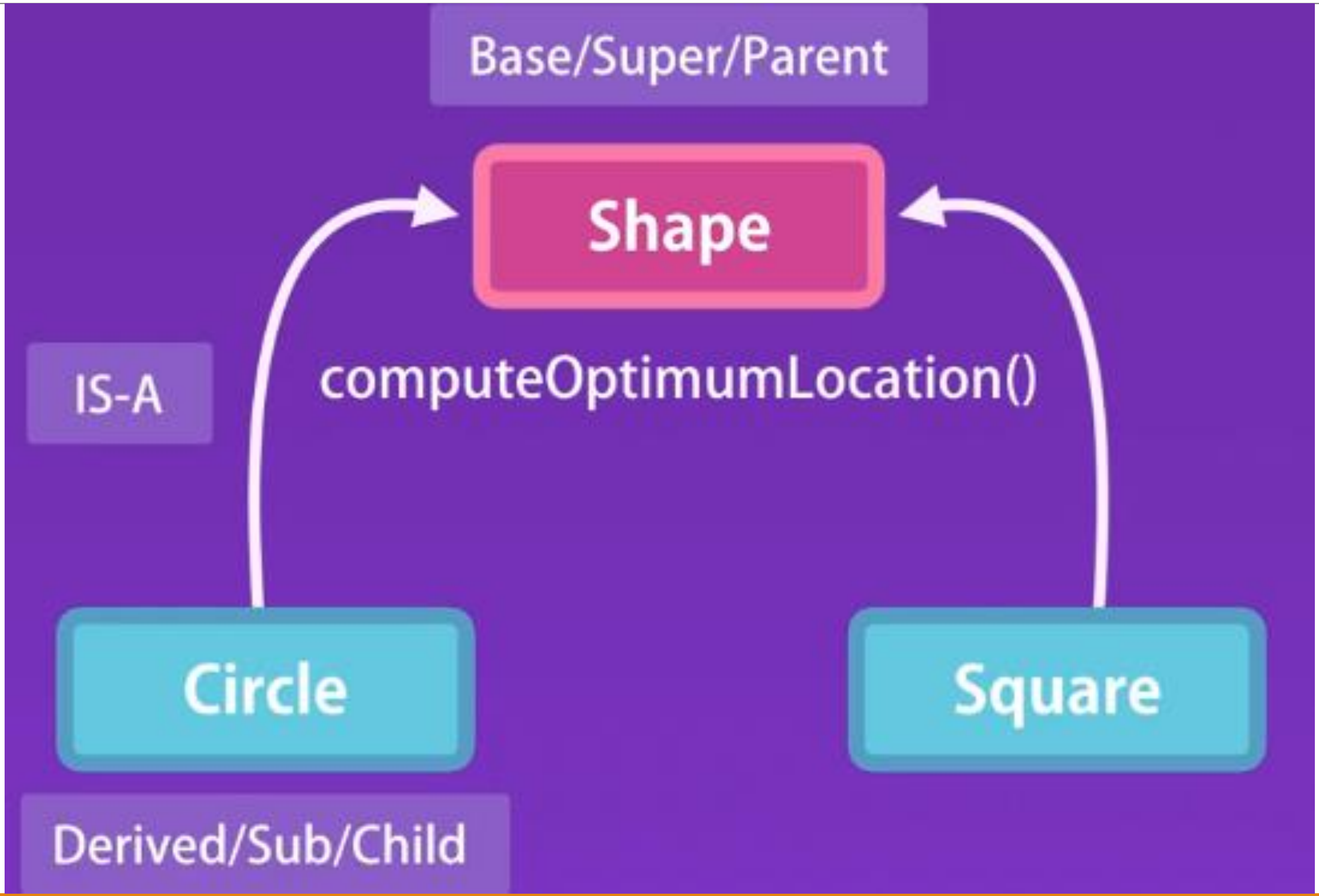


```
Object.defineProperty(this, 'defaultLocation', {
  get: function() {
    return defaultLocation;
  }
});
```

Cheat Sheet

https://1drv.ms/u/s!AtGKdbMmNBGdhQqT7nVD8sP5MlW2

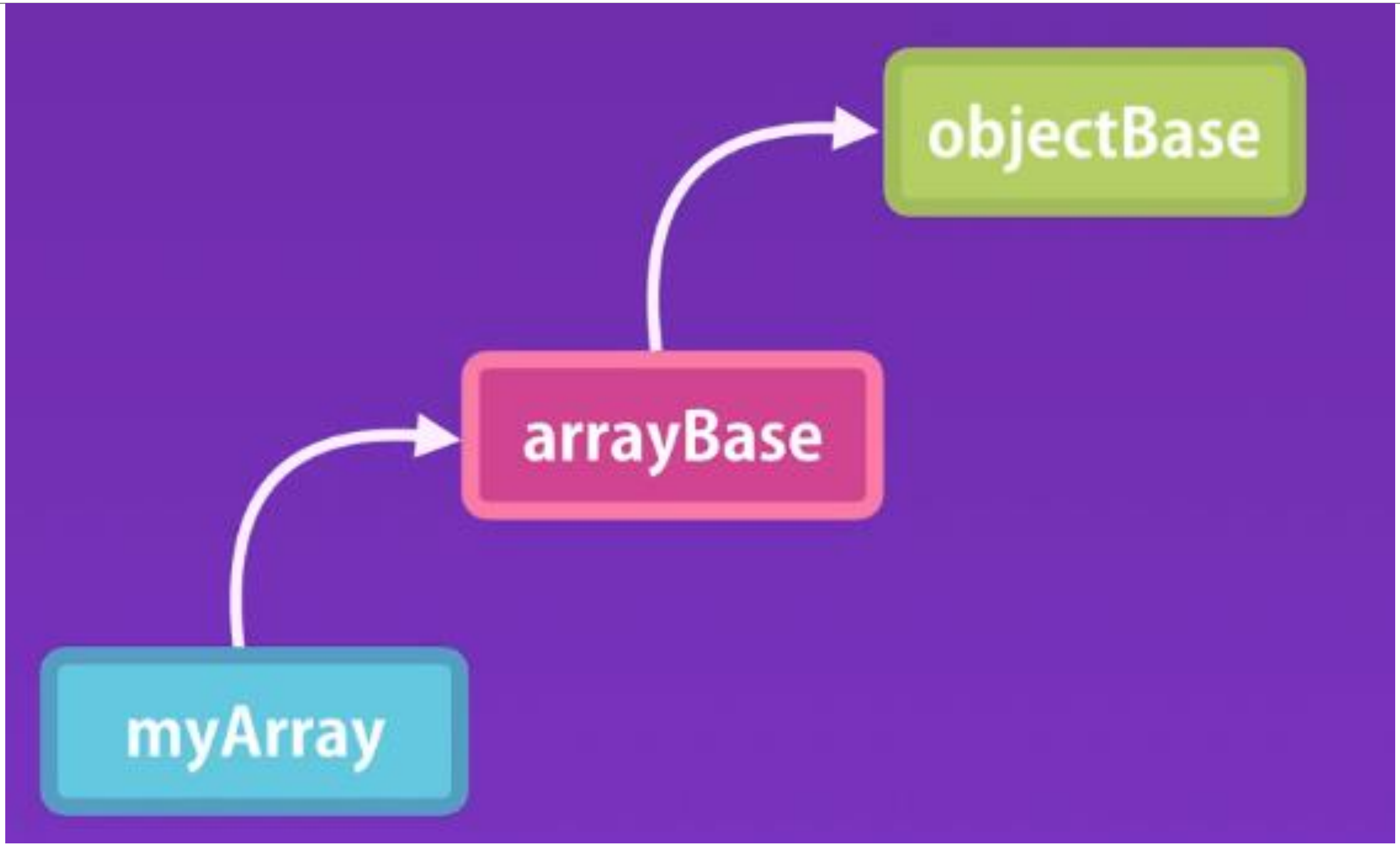
Inheritence



Prototypical Inheritence

```
// Every object (except the root object) has a prototype
(parent).

// To get the prototype of an object:

Object.getPrototypeOf(obj);

// In Chrome, you can inspect "__proto__" property. But you
should

// not use that in the code.

// x.__proto__ === y.__proto__
```

Multi level Inheritence



Property Descriptor

```
// To get the attributes of a property:
Object.getOwnPropertyDescriptor(obj,
toString');
configurable: true, // can be deleted
writable: true,
enumerable: false
```

"prototype" property

```
 // Constructors have a "prototype"
property. It returns the object

 // that will be used as the prototype for
objects created by the constructor.

 Object.prototype ===
Object.getPrototypeOf({})

 Array.prototype ===
Object.getPrototypeOf([])
```

Same Constructor Same Prototype

```
// All objects created with the same constructor will have the same prototype.
// A single instance of this prototype will be stored in the memory.
const x = {};
const y = {};
Object.getPrototypeOf(x) === Object.getPrototypeOf(y); // returns true
```

Best Practice

```
// When dealing with large number of objects, it's better to put their
// methods on their prototype. This way, a single instance of the methods
// will be in the memory.
Circle.prototype.draw = function() {}
```

Prototypical Inheritence

```
 function Shape() {}
 function Circle() {}

// Prototypical inheritance
 Circle.prototype =
Object.create(Shape.prototype);
 Circle.prototype.constructor = Circle;
```

Call Super

```
function Rectangle(color) {
// To call the super constructor
Shape.call(this, color);
}
```

Method Overriding

```
 // Method overriding
 Shape.prototype.draw = function() {}
 Circle.prototype.draw = function() {
 // Call the base implementation
 Shape.prototype.draw.call(this);

// Do additional stuff here
 }
```

Dos & Donts

```
 // Don't create large inheritance hierarchies.
 // One level of inheritance is fine.

// Use mixins to combine multiple objects
 // and implement composition in JavaScript.
```

Mixin

```
const canEat = {
eat: function() {}
};
const canWalk = {
walk: function() {}
};
function mixin(target,
...sources) {
// Copies all the
properties from all the
source objects
// to the target object.
Object.assign(target,
...sources);
}

function Person() {}

mixin(Person.prototype,
canEat, canWalk);
```

Resources

https://1drv.ms/f/s!AtGKdbMmNBGdhQmUmPL4RQRrfM1Y

ES6 Classes

Syntactical Sugar to Prototypical Inheritence

Class

```
class Circle {
 constructor(radius) {
  this.radius = radius;
 }
 // These methods will be added to the prototype.
 draw() {
 }
}
```

Static Methods

```
 // This will be available on the Circle class (Circle.parse())
 static parse(str) {
 }
```

Private Symbol

```
 // Using symbols to implement private
properties and methods
 const _size = Symbol();
 const _draw = Symbol();
```

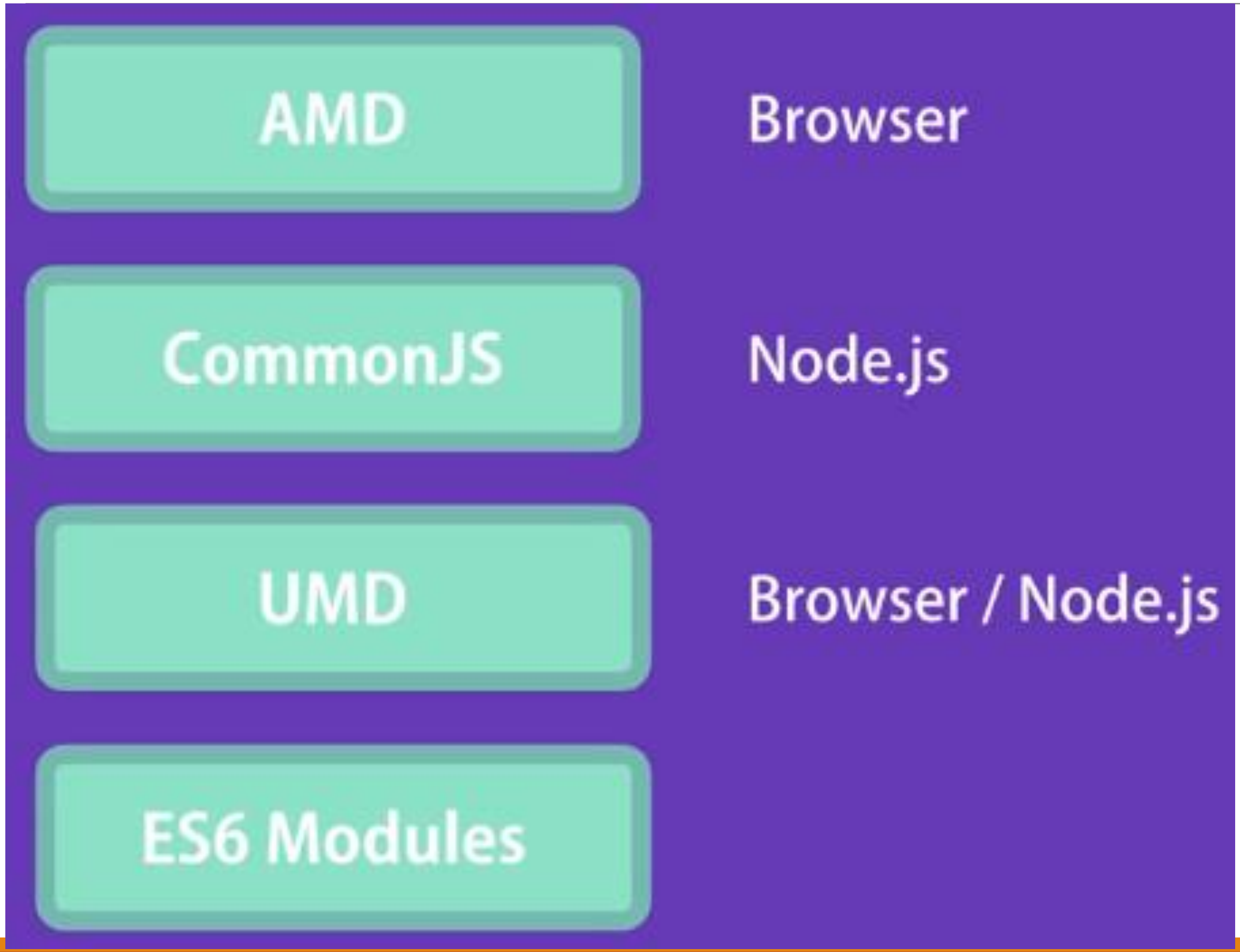
Inheritence

```
// Inheritance
class Triangle extends Shape {
constructor(color) {
// To call the base constructor
super(color);
}
```

```
draw() {
// Call the base method
super.draw();

// Do some other stuff here
}
}
```

Go Pro- ES6 Tooling

Module Formats



AMD	Browser
CommonJS	Node.js
UMD	Browser / Node.js
ES6 Modules	

Asynchronous Module Definition

Universal Module Definition

Common JS

```
// CommonJS (Used in Node)
// Exporting
module.exports.Cirlce = Circle;
// Importing
const Circle = require('./circle');
```

ES6

```
// ES6 Modules (Used in Browser)
// Exporting
export class Square {}
// Importing
import {Square} from './square';
```

Babel

```
// We use Babel to transpile our modern JavaScript code
// into code that browsers can understand (typically ES5).
```

Web Pack

```
 // We use Webpack to combine our JavaScript files into a
 // bundle.
```