

# JavaScript Object Orientation

## Goals

- Review how objects work in JavaScript
- Define classes in JavaScript
- Use classes to create instances that share functionality
- Describe constructor functions and use them to create instances
- Describe inheritance
- Define commonly used OOP (object-orientated programming) terms

## JS Objects Review

“Plain Old JavaScript Object” (POJO):

```
let o1 = {};  
  
let o2 = new Object(); // same thing  
  
o1.name = "Whiskey";  
o1["name"] = "Whiskey"; // same thing
```

Can add functions as keys:

```
o1.sayHi = function() { return "Hi!" };  
  
o1.sayHi(); // Hi!
```

Can get arrays of keys, values, or [key, val] arrays:

```
Object.keys(o1); // ["name", "sayHi"]  
  
Object.values(o1); // ["Whiskey", function () {...} ]  
  
Object.entries(o1); // [{"name", "Whiskey"},  
// ["sayHi", function () { ... } ]
```

## Details You Should Know

- Properties that do not exist in the object register as **undefined**.

```
o1.elie // undefined
```

(This causes issues when you attempt to invoke `()` or `.` access them)

- All keys get “stringified”:

```
o1[1] = "hello";  
o1["1"] = "goodbye";
```

- What is `o1[1]`?

```
o1[1];    // "goodbye"
```

(This gets even more confusing when using things like nested arrays as keys)

## Mixing Data And Functionality

### Functions and Data

Imagine some useful functions:

*demo/triangles.js*

```
/* area of right triangle */  
  
function getTriangleArea(a, b) {  
  return (a * b) / 2;  
}  
  
/* hypotenuse of right triangle */  
  
function getTriangleHypotenuse(a, b) {  
  return Math.sqrt(a * a + b * b);  
}
```

```
getTriangleArea(3, 4)    // 6  
getTriangleHypotenuse(3, 4) // 5
```

This gets a bit messy, though — all those functions to keep track of!

### Using a POJO

*demo/triangle-pojo.js*

```
let triangle = {  
  a: 3,  
  b: 4,  
  getArea: function() {  
    return (this.a * this.b) / 2;  
  },  
}
```

```
getHypotenuse: function() {  
    return Math.sqrt(this.a ** 2 + this.b ** 2);  
}  
};
```

```
triangle.getArea()           // 6  
triangle.getHypotenuse()     // 5
```

For now:

```
let triangle = {  
    a: 3,  
    b: 4,  
    getArea: function() {  
        return (this.a + this.b) / 2;  
    }  
};
```

**this** references to “this object”

So, we can helpfully mix data & functionality!

- This is tidy: related functionality lives together
- Annoying when we want more than one triangle

## Classes

Classes are a “blueprint” of functionality:

*demo/triangle-oo.js*

```
class Triangle {  
  
    getArea() {  
        return (this.a * this.b) / 2;  
    }  
  
    getHypotenuse() {  
        return Math.sqrt(this.a ** 2 + this.b ** 2);  
    }  
}
```

```
let myTri = new Triangle();    // "instantiation" of triangle  
myTri.a = 3;  
myTri.b = 4;  
myTri.getArea();              // 6  
myTri.getHypotenuse();        // 5
```

*demo/triangle-oo.js*

```
class Triangle {
  getArea() {
    return (this.a * this.b) / 2;
  }

  getHypotenuse() {
    return Math.sqrt(this.a ** 2 + this.b ** 2);
  }
}
```

- Defines the **methods** each instance of ***Triangle*** will have
- Make a new triangle with `new Triangle()`
- Can still add/look at arbitrary keys (“properties”)
- **this** is “the actual triangle in question”

Class names should be **UpperCamelCase**

Reduces confusion between ***triangle*** (an *actual, individual* triangle) and ***Triangle*** (the *class* of triangles)

A triangle is still an object:

```
typeof myTri;    // 'object'
```

But JS knows it’s an “instance of” the ***Triangle*** class:

```
myTri instanceof Triangle;    // true
```

## Constructors

Consider how we made an instance of our ***Triangle*** class:

```
let myTri = new Triangle();    // "instantiation" of triangle
myTri.a = 3;
myTri.b = 4;
```

*demo/triangle-constructor.js*

```
class Triangle {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  getArea() {
    return (this.a * this.b) / 2;
  }
}
```

```
getHypotenuse() {
  return Math.sqrt(this.a ** 2 + this.b ** 2);
}
```

The method with the special name **constructor** is called when you make a new instance.

```
let myTri2 = new Triangle(3, 4);
myTri2.getArea(); // 6
```

## What Can You Do in the Constructor?

- Whatever you want!
- Common things:
  - Validate data
  - Assign properties

```
constructor(a, b) {
  if (!Number.isFinite(a) || a <= 0)
    throw new Error("Invalid a: " + a);

  if (!Number.isFinite(b) || b <= 0)
    throw new Error("Invalid b: " + b);

  this.a = a;
  this.b = b;
}
```

(Note you don't return anything from constructor function).

## Methods

```
getArea() {
  return (this.a * this.b) / 2;
}
```

Functions placed in a class are “methods” (formally: “**instance methods**”).

They have access to properties of object with **this**.

They can take arguments/return data like any other function.

A method can call another method:

```
class Triangle {
  getArea() {
    return (this.a * this.b) / 2;
  }

  /* Is this a big triangle? */

  isBig() {
```

```

    return this.getArea() > 50;
  }
}

```

Note: to call a method, you need to call it on **this**

Without **this**, calling **getArea** throws a ReferenceError - it is not in scope!

## Inheritance & Super

demo/triangle-duplicate.js

```

class Triangle {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  getArea() {
    return (this.a * this.b) / 2;
  }

  getHypotenuse() {
    return Math.sqrt(
      this.a ** 2 + this.b ** 2);
  }

  describe() {
    return `Area is ${this.getArea()}.`;
  }
}

```

demo/triangle-duplicate.js

```

class ColorTriangle {
  constructor(a, b, color) {
    this.a = a;
    this.b = a;
    this.color = color;
  }

  getArea() {
    return (this.a * this.b) / 2;
  }

  getHypotenuse() {
    return Math.sqrt(
      this.a ** 2 + this.b ** 2);
  }

  describe() {
    return `Area is ${this.getArea()}.` +
      ` Color is ${this.color}!`;
  }
}

```

demo/triangle-extends.js

```

class Triangle {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  getArea() {
    return (this.a * this.b) / 2;
  }

  getHypotenuse() {
    return Math.sqrt(
      this.a ** 2 + this.b ** 2);
  }

  describe() {
    return `Area is ${this.getArea()}.`;
  }
}

```

demo/triangle-extends.js

```

class ColorTriangle extends Triangle {
  constructor(a, b, color) {
    // call parent constructor with (a, b)
    super(a, b);
    this.color = color;
  }

  // will "inherit" getArea, getHypotenuse

  // "override" describe() w/new version

  describe() {
    return super.describe() +
      ` Color is ${this.color}!`;
  }
}

```

## Multi-Level Inheritance

demo/triangle-extends.js

```

class ColorTriangle extends Triangle {
  constructor(a, b, color) {
    // call parent constructor with (a, b)
    super(a, b);
    this.color = color;
  }

  // will "inherit" getArea, getHypotenuse

  // "override" describe() w/new version

  describe() {
    return super.describe() +
      ` Color is ${this.color}!`;
  }
}

```

demo/triangle-extends.js

```

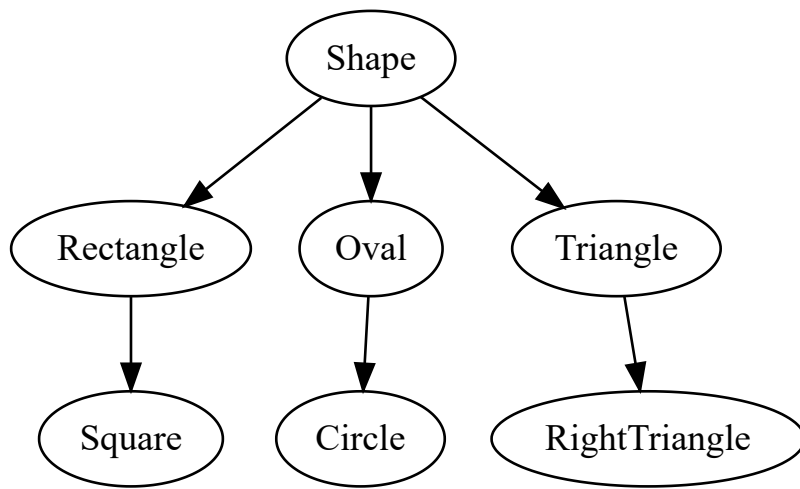
class InvisTriangle extends ColorTriangle {
  constructor(a, b) {
    // call parent constructor
    super(a, b, "invisible");
  }

  // still inherit getArea, getHypotenuse

  describe() {
    return "You can't see me!";
  }
}

```

Often end up with "class hierarchy":



## Terminology

- Instance
  - an individual instance; an array is “instance” of **Array**
- Class
  - blueprint for making instances
- Property
  - piece of data on an instance (e.g. `myTriangle.a`)
  - most languages call this idea an “instance attribute”
- Method
  - function defined by a class, can call on instance
  - most accurate to call these “instance methods”
- Parent / Superclass
  - More general class you inherit from
  - **Rectangle** might be parent of **Square**
- Child / Subclass
  - More specific class (a **Square** is a special kind of **Rectangle**)
- Inherit
  - Ability to call methods/get properties defined on ancestors
- Object Oriented Programming
  - Using classes & instances to manage data & functionality together
  - Often makes it easier to manage complex software requirements

## Looking Ahead



- More about ***this***
- Additional OO Concepts
- Python OO
- Oldschool JavaScript OOP