

Classes Part 2

JavaScript



Static Properties and Methods

You may have noticed that some classes we've had a look at, have methods or properties that work even without an object instance to operate on!

A couple of examples that you may have noticed would include the [Math.PI](#) property, or the [Math.sqrt\(\)](#) method. No object necessary in order to use these.

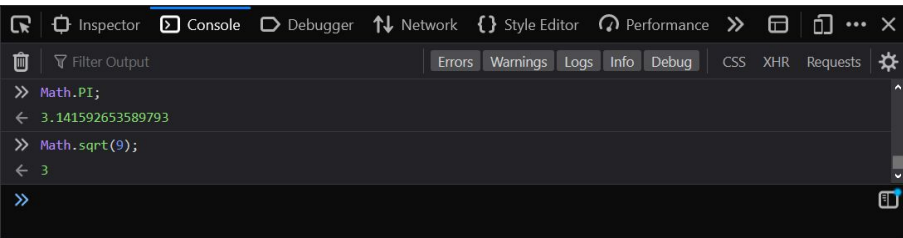
Give'em a try...

Properties and methods that are not run in instances of a class are referred to as [static](#).

These are typically used for utility values or methods—anything that may be unchanging or useful program-wide outside of specific objects.

Math.PI is a great example as it is a math value that doesn't change, and may be used anywhere in your program. There's no reason for us to have a Math object that stores just this number, the class does this well on its own.

This also helps us stay organized—all the math utilities are located in one descriptively-named class.

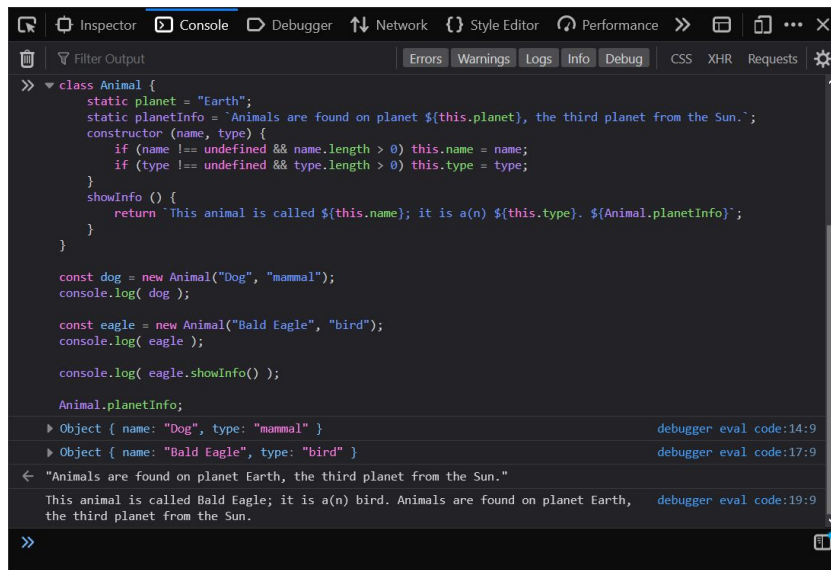


Writing a Class with Static Properties and Methods

When writing any class, you have the option of making any properties or methods you add static.

Any property or method in a class you write preceded by the static keyword will behave as a static property or method respectively.

Note that you will not be able to call upon these static fields from instances of your class, only directly from the class itself. For example, calling `dog.planet` below would result in an error.



```
>> class Animal {
  static planet = "Earth";
  static planetInfo = `Animals are found on planet ${this.planet}, the third planet from the Sun.`;
  constructor(name, type) {
    if (name !== undefined && name.length > 0) this.name = name;
    if (type !== undefined && type.length > 0) this.type = type;
  }
  showInfo() {
    return `This animal is called ${this.name}; it is a(n) ${this.type}. ${Animal.planetInfo}`;
  }
}

const dog = new Animal("Dog", "mammal");
console.log(dog);

const eagle = new Animal("Bald Eagle", "bird");
console.log(eagle);

console.log(eagle.showInfo());

Animal.planetInfo;
▶ Object { name: "Dog", type: "mammal" } debugger eval code:14:9
▶ Object { name: "Bald Eagle", type: "bird" } debugger eval code:17:9
◀ "Animals are found on planet Earth, the third planet from the Sun."
This animal is called Bald Eagle; it is a(n) bird. Animals are found on planet Earth, the third planet from the Sun. debugger eval code:19:9
>>
```

Accessors (Getters and Setters)

Accessors—or, computed properties—allow for us to describe both how a property should be read as well as updated.

Getters afford you a chance to format, manipulate, or otherwise prepare a value when it is called upon like a property from an object instance.

Setters, in contrast, provide you with a way to format, manipulate, or prepare a value in some way during assignment as a property on an object instance.

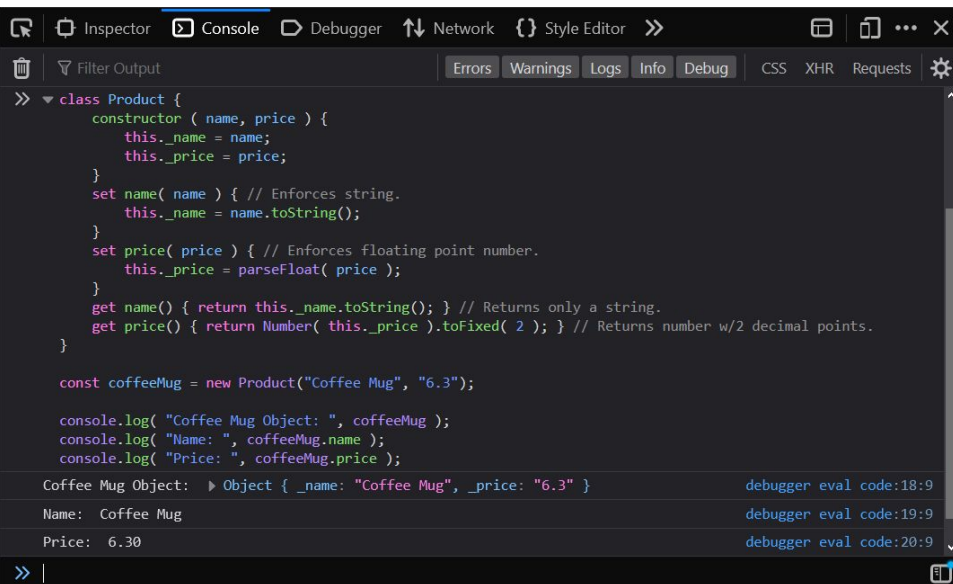
The get and set Keywords

For you to access a protected value, you will need a get.

For you to update a protected value, you will need a set.

Note that you typically will not have a traditional public property by the same name as a protected (get / set) one.

To follow this convention, people often store a property by a similar name, simply preceded by an underscore as seen below...



```
>> class Product {  
  constructor ( name, price ) {  
    this._name = name;  
    this._price = price;  
  }  
  set name( name ) { // Enforces string.  
    this._name = name.toString();  
  }  
  set price( price ) { // Enforces floating point number.  
    this._price = parseFloat( price );  
  }  
  get name() { return this._name.toString(); } // Returns only a string.  
  get price() { return Number( this._price ).toFixed( 2 ); } // Returns number w/2 decimal points.  
}  
  
const coffeeMug = new Product("Coffee Mug", "6.3");  
  
console.log( "Coffee Mug Object: ", coffeeMug );  
console.log( "Name: ", coffeeMug.name );  
console.log( "Price: ", coffeeMug.price );
```

Coffee Mug Object: ▶ Object { _name: "Coffee Mug", _price: "6.3" }	debugger eval code:18:9
Name: Coffee Mug	debugger eval code:19:9
Price: 6.30	debugger eval code:20:9

Polymorphism

If you're hoping to make a class that is largely the same as one you've built previously, you can have it inherit the same properties and methods via [polymorphism](#).

In JavaScript, this functionality is largely demonstrated by use of the [extends](#) keyword. When writing a class we can use this keyword to define which blueprint our new class will inherit from—that is to say, it can be considered a child class of the original (parent) class.

This also helps if we don't want to change a class, or can't, as the new child class won't affect it.

It is important to note that you can even extend built-in JavaScript classes, which is recommended if you do decide you want to add or change properties or methods in these pre-existing blueprints.

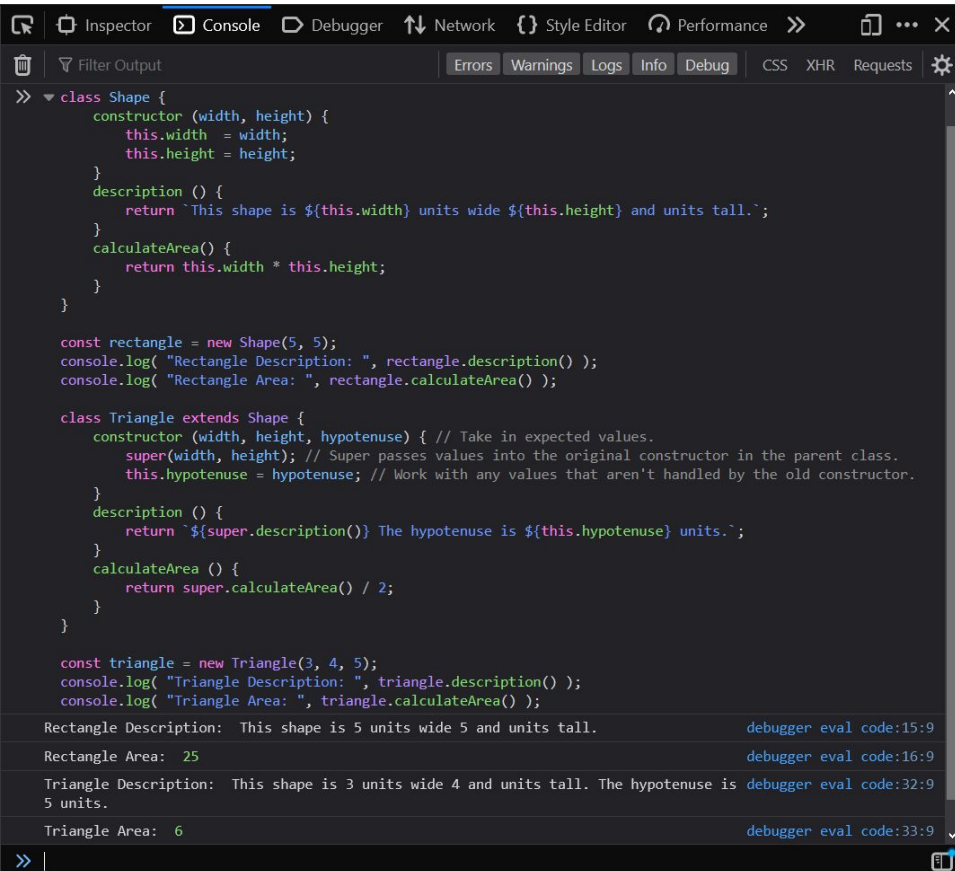
Avoid updating or adding to existing classes directly, as it often causes unexpected behaviours and makes troubleshooting very difficult.

To access a parameter, property, or method from a parent class in the child class, you can use the [super](#) keyword.

The extends and super Keywords

We can make a child class from an existing class using the extends keyword.

Give the following a try...



```
>> class Shape {
  constructor (width, height) {
    this.width = width;
    this.height = height;
  }
  description () {
    return `This shape is ${this.width} units wide ${this.height} and units tall.`;
  }
  calculateArea() {
    return this.width * this.height;
  }
}

const rectangle = new Shape(5, 5);
console.log( "Rectangle Description: ", rectangle.description() );
console.log( "Rectangle Area: ", rectangle.calculateArea() );

class Triangle extends Shape {
  constructor (width, height, hypotenuse) { // Take in expected values.
    super(width, height); // Super passes values into the original constructor in the parent class.
    this.hypotenuse = hypotenuse; // Work with any values that aren't handled by the old constructor.
  }
  description () {
    return `${super.description()} The hypotenuse is ${this.hypotenuse} units.`;
  }
  calculateArea () {
    return super.calculateArea() / 2;
  }
}

const triangle = new Triangle(3, 4, 5);
console.log( "Triangle Description: ", triangle.description() );
console.log( "Triangle Area: ", triangle.calculateArea() );
```

Rectangle Description: This shape is 5 units wide 5 and units tall.	debugger eval code:15:9
Rectangle Area: 25	debugger eval code:16:9
Triangle Description: This shape is 3 units wide 4 and units tall. The hypotenuse is 5 units.	debugger eval code:32:9
Triangle Area: 6	debugger eval code:33:9

Class Property and Method Accessibility

There are access permission levels afforded to class properties and methods to help us control more finely how our classes will be used and if or how properties can be accessed / modified.

In JavaScript, these features / keywords are currently proposed, and not released in modern browsers as of yet.

By default [all properties and methods in a JavaScript class are considered public](#) (unless they are stated or set to behave otherwise.) This means that we can access, or potentially even reassign, properties and methods in objects following the classes (blueprints) that we write.

[Private properties and methods](#) are accessible only from within the class itself. That is, to say, that a private property defined within the class is only accessible in that class' code block. Private properties or methods are prepended with an octothorpe (#).

If you instantiate an object from a class, and a property or method that you're attempting to access or execute is private, you'll simply be met with an error as a response instead of the desired effect.

As a developer this helps us control values and our functionality more closely, and prevent unexpected use-cases from coming up in our objects.