Learn JavaScript Syntax: Classes

Static Methods

Within a JavaScript class, the `static` keyword defines a static method for a class. Static methods are not called on individual instances of the class, but are called on the class itself. Therefore, they tend to be general (utility) methods.

```
class Dog {
  constructor(name) {
    this._name = name;
  }

  introduce() {
    console.log('This is ' + this._name +
' !');
  }

  // A static method
  static bark() {
    console.log('Woof!');
  }
}

const myDog = new Dog('Buster');
myDog.introduce();

// Calling the static method
Dog.bark();
```

Class

JavaScript supports the concept of *classes* as a syntax for creating objects. Classes specify the shared properties and methods that objects produced from the class will have.
When an object is created based on the class, the new object is referred to as an *instance* of the class. New instances are created using the `new` keyword.
The code sample shows a class that represents a `Song`. A new object called `mySong` is created underneath and the `.play()` method on the class is called. The result would be the text `Song playing!` printed in the console.

```
class Song {
  constructor() {
    this.title;
    this.author;
  }

  play() {
    console.log('Song playing!');
  }
}

const mySong = new Song();
mySong.play();
```

Class Constructor

Classes can have a `constructor` method. This is a special method that is called when the object is created (instantiated). Constructor methods are usually used to set initial values for the object.

```
class Song {
  constructor(title, artist) {
    this.title = title;
    this.artist = artist;
  }
}

const mySong = new Song('Bohemian
Rhapsody', 'Queen');
console.log(mySong.title);
```

Class Methods

Properties in objects are separated using commas. This is not the case when using the `class` syntax. Methods in classes do not have any separators between them.

```
class Song {
  play() {
    console.log('Playing!');
  }

  stop() {
    console.log('Stopping!');
  }
}
```

`extends`

JavaScript classes support the concept of inheritance — a child class can *extend* a parent class. This is accomplished by using the `extends` keyword as part of the class definition.
Child classes have access to all of the instance properties and methods of the parent class. They can add their own properties and methods in addition to those. A child class constructor calls the parent class constructor using the `super()` method.

```
// Parent class
class Media {
  constructor(info) {
    this.publishDate = info.publishDate;
    this.name = info.name;
  }
}

// Child class
class Song extends Media {
  constructor(songData) {
    super(songData);
    this.artist = songData.artist;
  }
}

const mySong = new Song({
  artist: 'Queen',
  name: 'Bohemian Rhapsody',
  publishDate: 1975
});
```

Learn JavaScript Syntax: Error Handling

Runtime Error in JavaScript

A JavaScript runtime error is an error that occurs within code while its being executed. Some runtime errors are built-in objects in JavaScript with a name and message property. Any code after a thrown runtime error will not be evaluated.

The `throw` Keyword in JavaScript

The JavaScript `throw` keyword is placed before an `Error()` function call or object in order to construct and raise an error. Once an error has been thrown, the program will stop running and any following code will not be executed.

```
// The program will raise and output this Error object with message 'Something went wrong'
throw Error('Something went wrong');

//The program will stop running after an error has been raised, and any following code will not be executed.
console.log('This will not be printed');
```

Javascript Error Function

The JavaScript *Error()* function creates an error object with a custom message. This function takes a string argument which becomes the value of the error's `message` property. An error created with this function will not stop a program from running unless the `throw` keyword is used to raise the error.

```
console.log(Error('Your password is too weak.')); //Error: Your password is too weak.
```

javascript try catch

A JavaScript `try` ... `catch` statement can anticipate and handle thrown errors (both built-in errors as well as those constructed with `Error()`) while allowing a program to continue running. Code that may throw an error(s) when executed is written within the `try` block, and actions for handling these errors are written within the `catch` block.

```
// A try...catch statement that throws a constructed Error()
try {
  throw Error('This constructed error will be caught');
} catch (e) {
  console.log(e); // Prints the thrown Error object
}

// A try...catch statement that throws a built-in error
const fixedString = 'Cannot be reassigned';
try {
  fixedString = 'A new string'; // A TypeError will be thrown
} catch (e) {
  console.log('An error occurred!'); // Prints 'An error occurred!'
}

console.log('Prints after error'); // Program continues running after the error is handled and prints 'Prints after error'
```

Print   Share ▼

0