

# Classes

Learn how JavaScript classes offer us a new way to implement inheritance and object-oriented programming. Master the new syntax and see how we can leverage it to write more elegant code. Learn how easy it makes inheritance and what the JavaScript engine does for us with this new syntax.

Classes in JavaScript are nothing more than special functions. They are created by using the `class` keyword and being given a name. Like a normal function, when invoked with `new`, a class returns an object.

```
class Person {}  
  
const alex = new Person();  
console.log(alex); // -> Person {}
```



## Class `constructor`

Classes have a special method on them called `constructor`. This is the method that is invoked when the class is invoked. Think of the `constructor` method as the function body of the class.

We add methods on a class as if it were an object.

```
class Person {  
  constructor(first, last, age = 25) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
  }  
}  
  
const alex = new Person('Alex', 'Smith');  
console.log(alex);  
// -> Person { firstName: 'Alex', lastName: 'Smith', age: 25 }
```



We can see that calling `new Person()` results in the class constructor method being called.

At this point, the above code is equivalent to this.

```
function Person (first, last, age = 25) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
}

const alex = new Person('Alex', 'Smith');
console.log(alex);
// -> Person { firstName: 'Alex', lastName: 'Smith', age: 25 }
```

## Class Methods

Classes can have methods placed on them. A method placed on a class gets attached to its `prototype`.

```
class Person {
  constructor(first, last, age = 25) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
  }

  growOlder() {
    this.age++;
  }
}

const alex = new Person('Alex', 'Smith', 20);
alex.growOlder();
alex.growOlder();
console.log(alex);
// -> Person { firstName: 'Alex', lastName: 'Smith', age: 22 }
```

`growOlder` exists on `Person.prototype` and `alex` uses it through inheritance.

## Static Methods

Static methods are created by writing the keyword `static` before the method. They are placed directly on the class itself, not on its prototype. An object created through the class (using `new`) cannot call a static method using dot notation. They can only be accessed through the name of the class itself.

Static methods are generally utility functions. They can't reference the instance of the class through `this` because they're not on the prototype.

```
class Person {
  constructor(first, last, age = 25) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
  }

  growOlder() {
    this.age++;
  }

  static describePersonClass() {
    console.log('This class creates a person object.');
```

This class is now equivalent to the following.

```
function Person (first, last, age = 25) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
}

Person.prototype.growOlder = function() {
  this.age++;
}

Person.describePersonClass = function() {
  console.log('This class creates a person object.');
```

## Summary

Classes give us an alternative way to use object oriented programming. Instead of dealing with `prototype`s directly, we place methods directly on the class. The `constructor` function is what actually runs when we call `new Person()`. A `static` method is placed directly on the class itself.

## Extending Classes

Classes can *extend* other classes using the `extends` keyword. The class being extended is called the superclass and the new class is called the subclass.

**An object created by a subclass prototypically inherits from the superclass.** Let's say we invoke a subclass and put the object we get back into `obj`. Then `obj.__proto__` is equal to the subclass's `prototype` property. `obj.__proto__.__proto__` is equal to the superclass's `prototype` property.

```
class Person {
  constructor(first, last, age = 25) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
  }

  growOlder() {
    this.age++;
  }

  static describePersonClass() {
    console.log('This class creates a person object.');
```

```
  }
}

class Boy extends Person {
  constructor(first, last, height, age = 10) {
    super(first, last, age);
    this.height = height;
  }

  printHeight() {
    console.log(this.height + ' ft');
  }
}

const tim = new Boy('Tim', 'Johnson', 3.2, 8);
tim.growOlder();
tim.printHeight(); // -> 3.2 ft
console.log(tim);
// -> Boy { firstName: 'Tim', lastName: 'Johnson', age: 9, height: 3.2 }
```

```
console.log(tim.__proto__ === Boy.prototype); // -> true
console.log(tim.__proto__.__proto__ === Person.prototype); // -> true
```




Note that we call a function called `super` on line 19. In a subclass's constructor, we have access to the `super` function. It calls the superclass's constructor function, binding the current instance of `this` to the superclass. If `super` is to be called, it must be called before using `this` in the constructor.

The above code is equivalent to the following.

```
function Person(first, last, age = 25) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
}

Person.prototype.growOlder = function() {
  this.age++;
}

Person.describePersonClass = function() {
  console.log('This class creates a person object.');
```



```
}

function Boy(first, last, height, age = 10) {
  Person.call(this, first, last, age);
  this.height = height;
}

Boy.prototype = Object.create(Person.prototype);
Boy.prototype.constructor = Boy;

Boy.prototype.printHeight = function() {
  console.log(this.height + ' ft');
};

const tim = new Boy('Tim', 'Johnson', 3.2, 8);
tim.growOlder();
tim.printHeight(); // -> 3.2 ft
console.log(tim);
// -> Boy { firstName: 'Tim', lastName: 'Johnson', age: 9, height: 3.2 }

console.log(tim.__proto__ === Boy.prototype); // -> true
console.log(tim.__proto__.__proto__ === Person.prototype); // -> true
```



## Conclusion

Classes will take some time getting used to. Once you embrace them, you'll find that they streamline inheritance a lot. We no longer have to manually link up prototypes or set up inheritance ourselves. `extends` and `super` take care of that for us.

Understanding classes is key to using React.js as it's currently used in industry. Modern React code is almost exclusively written in ES2015 and based entirely on classes.