> If you're working with modern evergreen browsers, classes is the prefered way to create reusable code. They work in all modern browsers according to caniuse.com out of the box. Prototypal inheritance will work if you need to support older browsers.

Prototypal Inheritance

Before ES6 we created reusable content using functions to create objects and attaching the methods to the object's prototype. This way every object that inherits from from the object will also inherits the methods attached to the prototype.

In the example below the Person has the following properties: `first`, `last`, age, gender, and `interests`. all these properties are required for all Person objects.

It also defines two methods for the Person object with the syntax `Person.prototype.methodName` to create a function representing the method.

Both methods return alerts with strings that interpolate values using string concatenation to pull elements from the Person object to display it.

The `interests` attribute is an array of one or more elements. For these examples, I've made them arrays of one item.

The name attribute is an object made of two children: `first` and `last`. This is why the methods refer to `this.name.first` rather than `this.first`.

Be careful when using string interpolation that you match the quotation marks and that you escape quotation marks that are part of the sentence rather than the string. Look at how we escaped the apostrophe in `I'm` to make sure that it didn't close the string and caused errors when we run the code.

```
function Person(first, last, age, gender, interests) {
  this.name = {
    first,
    last
  };
```

```
  this.age = age;
  this.gender = gender;
  this.interests = interests;
};

//
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name.first + '.');
};

Person.prototype.farewell = function() {
  alert(this.name.first + ' has left the building. Bye for now!');
}
'Hi! I\'m '
```

We can then create new People based on the Person object with their four attributes and getting the two methods we defined. We use the `object.method` syntax to use the methods defined in the object's prototype.

```
const kirk = new Person('Jim', 'Kirk', 39, 'male', ['starshi'Kirk'
const mccoy = new Person('Leonard', 'McCoy', 44 ['Medicine']);

kirk.greeting();
'Leonard'// Hi! I'm Jim.
mccoy.farewell();
// Leonard has left the building. Bye for now!
```

ES2015 Classes

ES2015 introduces classes to JavaScript as a way to give a cleaner syntax for reusable code. This syntax will be familiar if you've written code in C++ and Java. We'll convert the Person and Teacher examples from prototypal inheritance to a Class.

The constructor element builds the function that represents our class, in our example Person. All person objects will have these items.

`greeting()` and `farewell()` are class methods. They are defined as part of the class but are not required like the items inside the constructor. We can have a person without greeting and farewell but we can not have a farewell or a greeting without a person.

Both class methods use string literals rather than string concatenation in the class methods to make string interpolation easier and the code easier to read.

```
class Person {
  constructor(first, last, age, gender, interests) {

    this.name = {
      first,
      last
    };

    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }


  greeting() {
    alert(`Hi! I'm ${this.name.first}`);
  };

  farewell() {
    alert(`${this.name.first} has left the building. Bye for now!`);
  };
}
```

We then instantiate Person objects using the new `Person()` syntax and call methods using the object.method syntax.

```
let han = new Person('Han', 'Solo', 25, 'male', ['smug'Solo']);
han.greeting();
// Hi! I'm Han
```

```
let leia = new Person('Leia', 'Organa', 19, 'female' ['government']]);
leia.farewell();
'Leia'// Leia has left the building. Bye for now
```

Under the hood, JavaScript engines will convert the classes into Prototypal Inheritance models. You don't have to worry about it, it's all done for you.

Inheritance: Creating Classes Based On Other Classes

We created a class to represent a Person. They have a series of attributes that are common to all people; Let's say that we want to create a specialized type of Person: a Teacher.

To save ourselves the typing we'll base our teacher class on the person we've already created. This is called creating a subclass or subclassing.

To create a subclass we use the `extend` keyword to tell Javascript the class we want to base our class on.

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    this.name = {
      first,
      last
    };

  this.age = age;
  this.gender = gender;
  this.interests = interests;
  // subject and grade are specific to Teacher
  this.subject = subject;
  this.grade = grade;
  }
}
```

We can make the code more readable by using the super() declaration as the first

item in the class constructor. This will call the parent class' constructor to take care of items defined there. We can further refine the Teacher example by calling super in the constructor; this will call Person's constructor and set up its values.

We can also add additional items that are specific to the class we are creating like subject and grade in the Teacher example.

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    super(first, last, age, gender, interests);

    // subject and grade are specific to Teacher
    this.subject = subject;
    this.grade = grade;
  }
}
```

The teacher class uses values from Person and adds functionality that goes beyond the parent class. We didn't write greeting() and farewell() methods for Teacher but they are available as part of Person, so we can still use them, we get them "for free" in Teacher objects.

```
let snape = new Teacher('Severus', 'Snape', 58, 'male', ['potions'],'Snape
snape.greeting(); // Hi! I'm Severus.
snape.farewell(); // Severus has left the building. Bye for now.
```

Like we did with Teachers, we could create other subclasses of person to make them more specialized without modifying the base person class.

Getters and Setters

There may be times when we want to change the values of an attribute in the classes we create or we don't know what the final value of an attribute will be. Using the Teacher example, we may not know what subject the teacher teaches or the subject may change from the time we create the teacher object to the time we use it.

We can do it with getters and setters.

We'll use the Teacher class we defined earlier and enhance it with getters and setters. The class starts the same than it was the last time we looked at it.

Getters and setters work in pairs. Getters return the current value of the variable and setters change the value of the variable to the ones it defines.

The modified Teacher class looks like this:

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    super(first, last, age, gender, interests);
    // subject and grade are specific to Teacher
    this._subject = subject;
    this.grade = grade;
  }

  get subject() {
    return this._subject;
  }

  set subject(newSubject) {
    this._subject = newSubject;
  }
}
```

In our class above we have a getter and setter for the subject property. We use _ to create a backing field to store our name property. Without this convention, we will get stack overflow errors every time we call get or set.

To show the current value of the _subject property of the snape object use `snape._subject`. To assign a new value to the _subject property use `snape._subject="new value"`. The example below shows the two features in action:

```
// Check the default value
snape._subject // Returns "potions"

// Change the value
```

```
snape._subject="magic arts"
"magic arts"// Sets subject to "magic arts"

// Check it again and see if it matches the new valueect // Returns "magi
"magic arts"
```