ENCAPSULATION

Encapsulation includes the idea that the data of an object should not be directly exposed. Instead, callers that want to achieve a given result are coaxed into proper usage by invoking methods (rather than accessing the data directly).

Encapsulation refers to enclosing all the functionalities of an object within that object so that the object's internal workings (its methods and properties) are hidden from the rest of the application. This allows us to abstract or localize specific set of functionalities on objects.

Encapsulation is the ability of an object to be a container (or capsule) for its member properties, including variables and methods. As a fundamental principle of object oriented programming (OOP), encapsulation plays a major role in languages such as C++ and Java. However, in scripting languages, where types and structure are not actively enforced by the compiler or interpreter, it is all-too-easy to fall into bad habits and write code that is brittle, difficult to maintain, and error-prone.

While JavaScript does not support Classes per se, it does allow for some of their main features, including data hiding, which is one of the main consequences of encapsulation. Several JS frameworks have objects that mimic classes in several ways, but if you don't want to add extra load time to your pages, it's easy enough to do it yourself.

AGGREGATION

Javascript aggregation is just what it sounds like: it groups Javascript files into a single file as they are added during a page request. This in order to cut down on the number of HTTP requests needed to load a page. Fewer HTTP requests is generally better for front-end performance.

If fewer is better, why multiple aggregates?... Before we dig in, I said that fewer HTTP requests is generally better for front-end performance. You may have noticed that certain API's create more than one Javascript aggregate for each page request. Why not a single aggregate? The reason for this is to take advantage of browser caching.

If you had a single aggregate with all the Javascript on the page, any difference in the aggregate from page to page would cause the browser to download a different aggregate with a lot of the same code. This situation arises when Javascript is conditionally added to the page, as is the case with many modules and themes.

Ideally, we group Javascript on every page apart from conditionally added Javascript. That way the browser would download every aggregate once and subsequently load it from cache on other page requests. The optimal aggregation would be to strike a balance between making **fewer HTTP requests**, and leveraging **browser caching**.

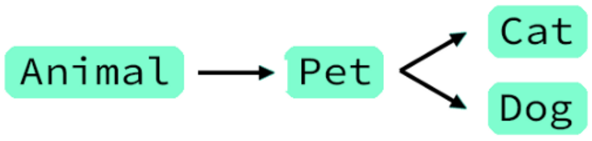
INHERITANCE

JavaScript objects use **prototype-based inheritance**. Its design is logically similar (but different in implementation) from class inheritance in strictly Object Oriented Programming languages.

It can be loosely described by saying that when methods or properties are attached to object's **prototype** they become available for use on that object and its descendants. But this process often takes place behind the scenes.

When you use **class** and **extends** keywords internally JavaScript will still use prototype-based inheritance. It just simplifies the syntax. Perhaps this is why it's important to understand how prototype-based inheritance works. It's still at the core of the language design.

Creating A Logical Hierarchy Of Object Types



Cat and **Dog** are inherited from **Pet** which is inherited from **Animal**.

A **Dog** and a **Cat** share similar traits. Instead of creating two different classes, we can simply create one class **Pet** and inherit **Cat** and **Dog** from it. But the **Pet** class itself can also be inherited from the class **Animal**.

JavaScript supports object inheritance via something known as **prototypes**. There is an object property called prototype attached to each object.

Working with the class and extends keywords is easy but actually understanding how prototype-based inheritance works is not trivial.

Functions can be used as **object constructors**. The name of a constructor function usually starts with an uppercase letter to draw the distinction between regular functions. Object constructors are used to create an instance of an object.

Some of the JavaScript built-in objects were already created following the same rules. For example **Number**, **Array** and **String** are inherited from **Object**. As we discussed earlier, this means that any property attached to Object becomes automatically available on all of its children.

Let's define a new object **Bird** and add 3 properties: **type**, **color** and **eggs**. Let's also add 3 methods: **fly**, **walk** and **lay_egg**. Something all birds can do:

```javascript
// Define class Bird
function Bird(type, color) {
  // Define properties
  this.type = type;
  this.color = color;
  this.eggs = 0;
  // Define method fly()
  this.fly = function() {
    console.log(`${this.color} ${this.type} is flying.`);
  }
  // Define method walk()
  this.walk = function() {
    console.log(`${this.color} ${this.type} is walking.`);
  }
  // Define method lay_egg()
  this.lay_egg = function() {
    this.eggs++;
    console.log(`${this.color} ${this.type} laid an egg!`);
  }
}

```

Using `class` and `extends` keywords

The ES5-style constructors can be a bit cumbersome.

Luckily we now have **class** and **extends** keywords to accomplish exactly the same thing we just did in the previous section.

class replaces **function**

```javascript
class Bird {
  constructor(type, color) {
    this.type = type;
    this.color = color;
  }
  fly() { ... }
  walk() { ... }
  lay_egg() { ... }
}
```

extends and **super()** replace **Bird.call** from the previous examples.

```
class Parrot extends Bird {
  constructor(type, color) {
    super(type, color);
    this.type = type;
    this.color = color;
  }
  talk() { ... }
}
```

Note we must use **super()** which calls the constructor of the parent class.

This syntax looks a lot more manageable!

Now all we have to do is instantiate the object:

```
let parakeet = new Parrot("parakeet", "orange");
parakeet.talk();
```

Overview

Class inheritance helps establish a hierarchy of objects.

Classes are the fundamental building blocks of your application design and architecture. They make working with code a bit more human.

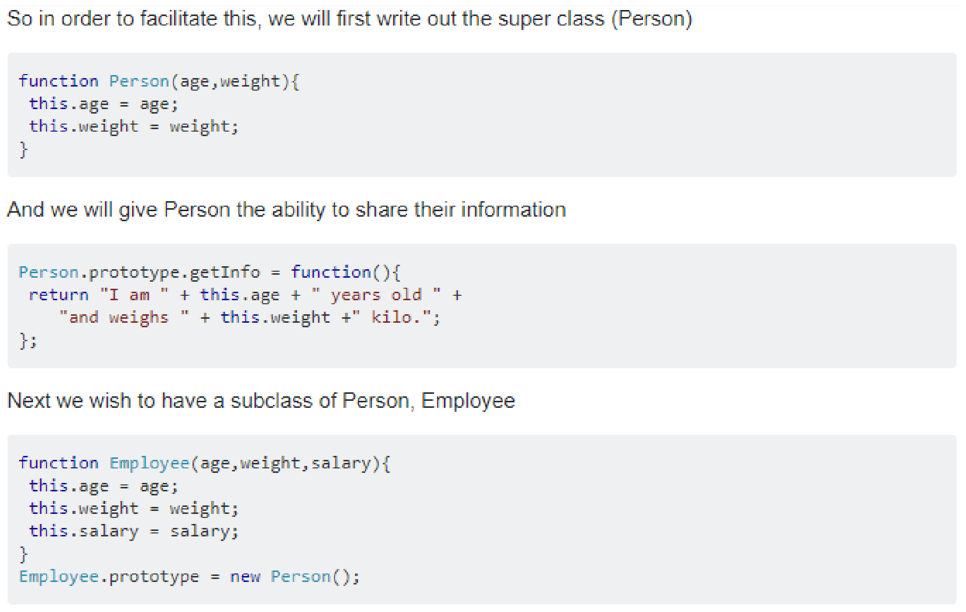
Of course, **Bird** was just an example. In a real-world scenario, it could be anything based on what type of application you're trying to build.

Vehicle class can be a parent of **Motorcycle**, **Car**, or **Tank**.

POLYMORPHISM

The polymorphism is a core concept of an object-oriented paradigm that provides a way to perform a single action in different forms. It provides an ability to call the same method on different JavaScript objects. As JavaScript is not a type-safe language, we can pass any type of data members with the methods.

Let's take a look at an example of people and employees. All employees are people, but all people are not employees. Which is to say that people will be the super class, and employee the sub class. People may have ages and weights, but they do not have salaries. Employees are people so they will inherently have an age and weight, but also because they are employees they will have a salary.



So in order to facilitate this, we will first write out the super class (Person)

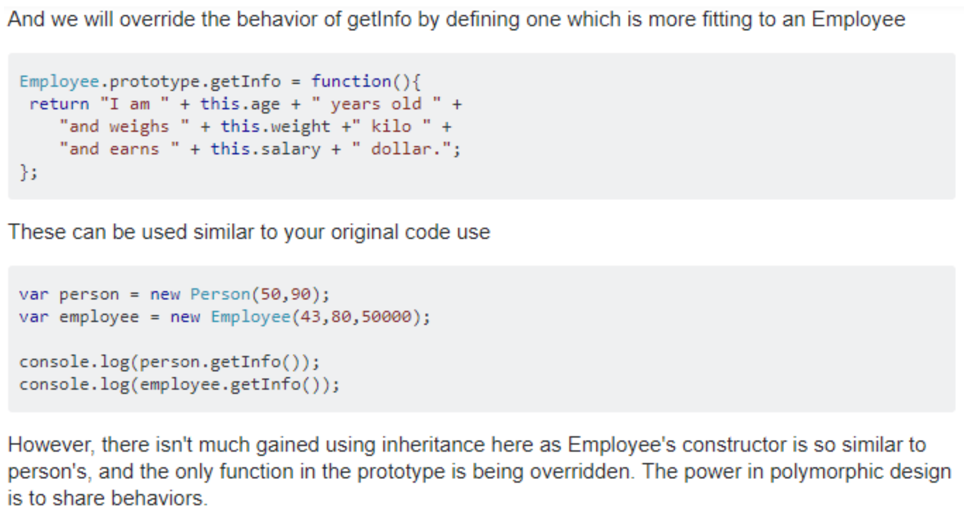
```
function Person(age,weight){
 this.age = age;
 this.weight = weight;
}
```

And we will give Person the ability to share their information

```
Person.prototype.getInfo = function(){
 return "I am " + this.age + " years old " +
    "and weighs " + this.weight +" kilo.";
};
```

Next we wish to have a subclass of Person, Employee

```
function Employee(age,weight,salary){
 this.age = age;
 this.weight = weight;
 this.salary = salary;
}
Employee.prototype = new Person();
```



And we will override the behavior of getInfo by defining one which is more fitting to an Employee

```
Employee.prototype.getInfo = function(){
 return "I am " + this.age + " years old " +
    "and weighs " + this.weight +" kilo " +
    "and earns " + this.salary + " dollar.";
};
```

These can be used similar to your original code use

```
var person = new Person(50,90);
var employee = new Employee(43,80,50000);

console.log(person.getInfo());
console.log(employee.getInfo());
```

However, there isn't much gained using inheritance here as Employee's constructor is so similar to person's, and the only function in the prototype is being overridden. The power in polymorphic design is to share behaviors.