**WHAT IS OOP**

A programming paradign centered around objects rather than functions.

OOP is not a programming language or tool. It’s a style of programming or programming paradigm.

There are several programming languages that support this style:

C#, Java, Ruby, Python, JavaScript. Many frameworks use it, like Angular.

**4 Pillars of Object-oriented Programming:**

**Encapsulation,**

**Abstraction,**

**Inheritance,**

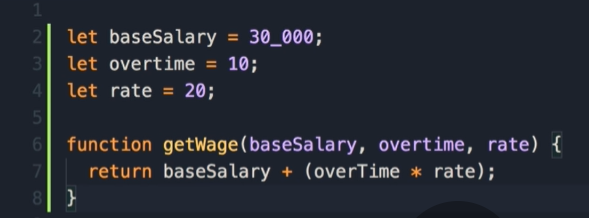
**Polymorphism.**

Before OOP, we had procedural programming that devided a programm into a set of functions. So there was a bunch of variables and functions that operated on those variables. But as you programm grows, you’ll end up with a bunch of functions all over the place, you might find yourself copying and pasting lines of code repeatedly. That's what we call **spaghetti code.**

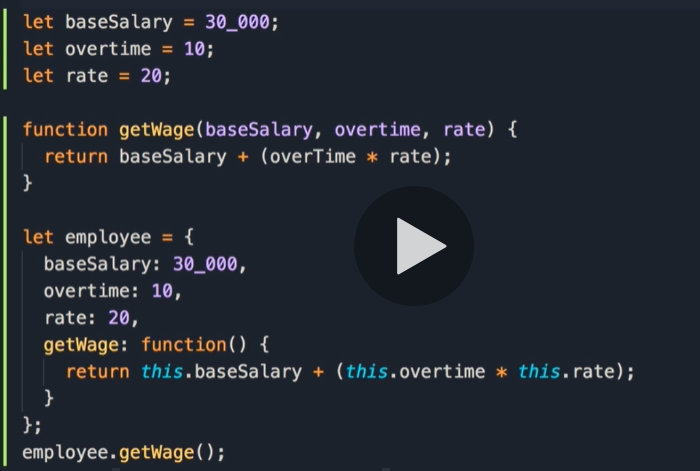
OOP came to solve this problem. In OOP we combine a group of related variables and functions into a unit, called object. We refer to these variables as properties and the functions as methods.

In OOP we group related variables and functions that operate on them into objects. This is what we call **encapsulation.**

**procedural**



**Oop**



**GetWage** from employee object has no parameters, this is because all the parameters are modeled as properties of this object. These properties and getWage function are related and so part of one unit.



That’s encapsulation.

**ABSTRACTION -** Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

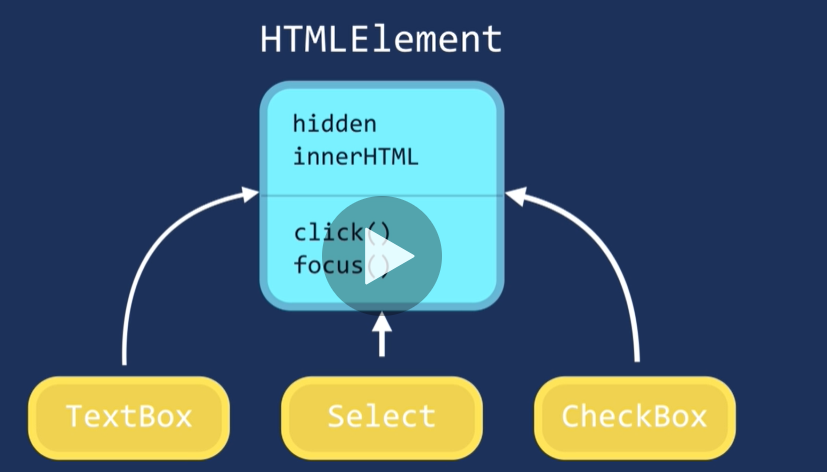
This gives us a couple of benefits.

1. Simpler Interface. Understanding an object with a few properties and methods is much simpler.

2. Reduce the impact of change. You can make changes and none of those changes will have an impact on the application.

**INHERITANCE**

Inheritance is a mechanism that helps reducing reduncant code.



**Inheritance** is a mechanism in which one object acquires all the properties and behaviors of a parent object. When you **inherit** from an existing class, you can reuse methods and fields of the parent class.

**Polymorphism** means may forms. and it occurs when we have many classes that are related to each other by inheritance. In OOP polymorphism is a technique that allows you to get rid of long if and else or switch and case statements.



With OOp we can implement a render method in each of these objects, and this method will behave differently depending on the object we are referencing.

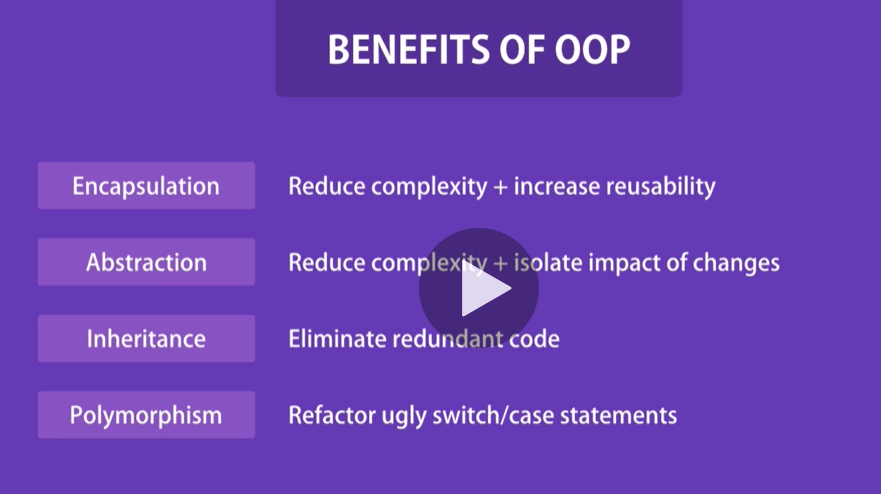
**BENEFITS OF OOP**

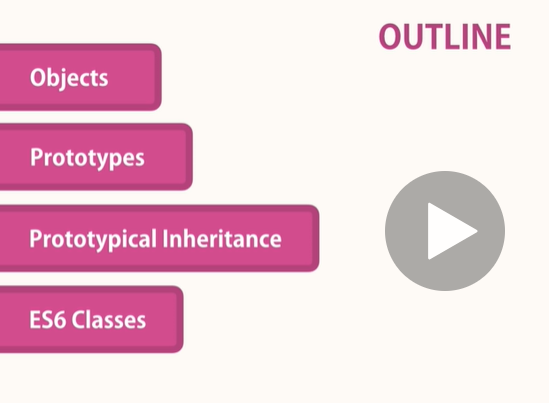
Using Encapsulation we group related variables and functions together and this way we can reduce complexity.

With Abstraction we hide the details and the complexity and show only the essentials.

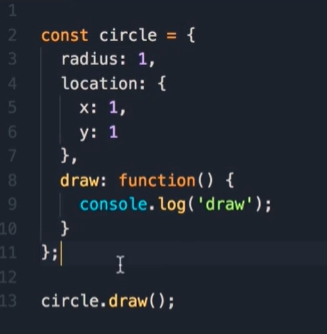
With Inheritance we eliminate redundant code.

With Polymorphism we refactor ugly switch case statements.





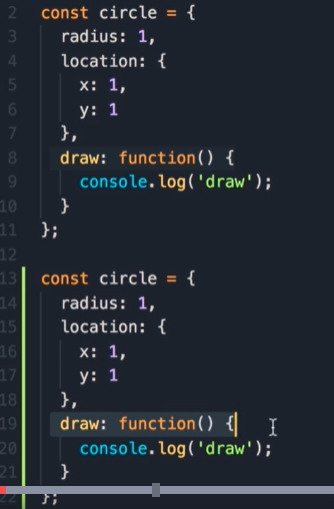
**Object Literals**



Object literal is a simple way to define an object. But we can also define objects using factories and constructors.

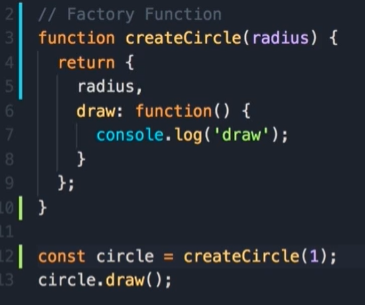
**Factories**

Imagine we want to create another circle. With the current implementation we have to duplicate all this code.



If an object has one or more methods, we say that that object has behavior. Creating objects using object literal syntax is an issue only if that object has behavior.

The solution is to use a factory or a constructor function.

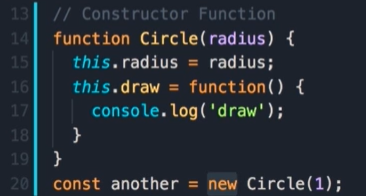


Now, just by calling this function a new object is created.

There’s another way to create an object, using a constructor function. It’s like defining a class in other languages, but in JS we don’t have classes.

Inside we use “this” keywork to set the properties of this object. THIS is a reference to the object that is executing this piece of code.

We use the “new” keyword to create an object.



When using new a few things happen under the hood. 1. New creates an empty object, then 2. it will set this to point to that object, and finally 3. it will return that object from Circle function. We don’t have an explicit return statement.

If you don’t use the new keyboard when creating an object from a constructor function, then this defines those variables on the global object (window object).

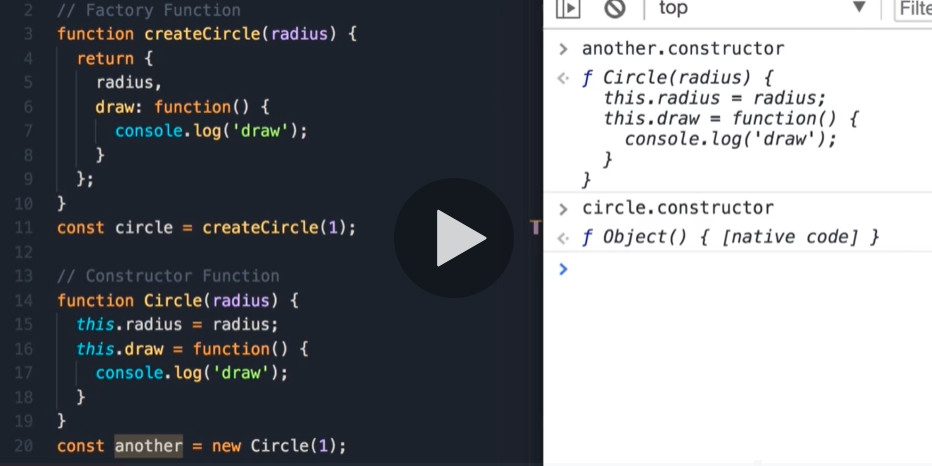
If we return an object from a function, we refer to that function as a factory function.

In contrast, if we use this keyword with the new operator, we refer to that function as a constructor function.

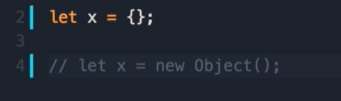
As a developer you should ne familiar with both those patterns.

**Constructor property**

Every object in JS has a property called constructor, and that references the function that was used to construct or create that object.

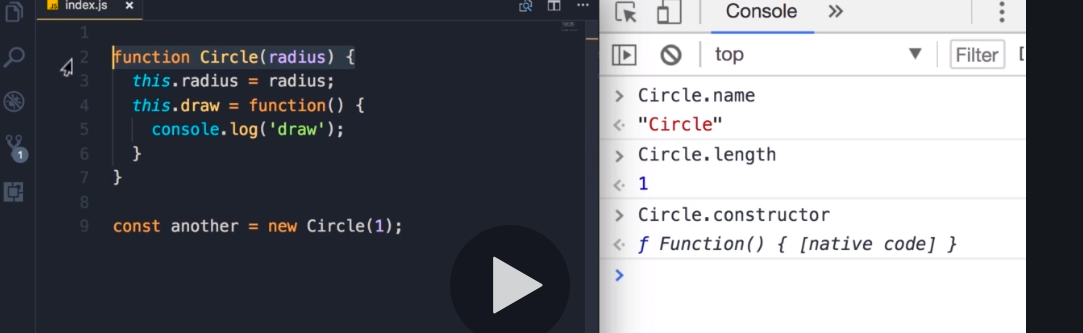


Object is a build in constructor function. When creating an object using the object literal syntax, internally JS engine uses this constructor function.

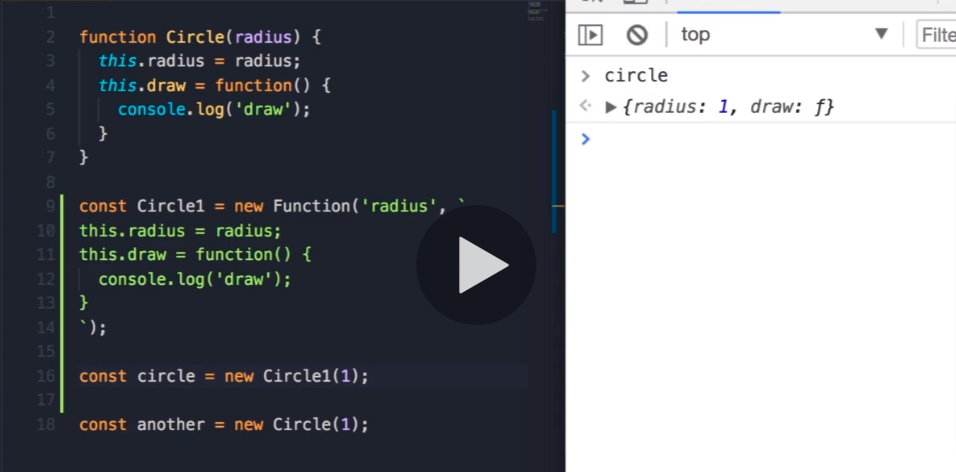


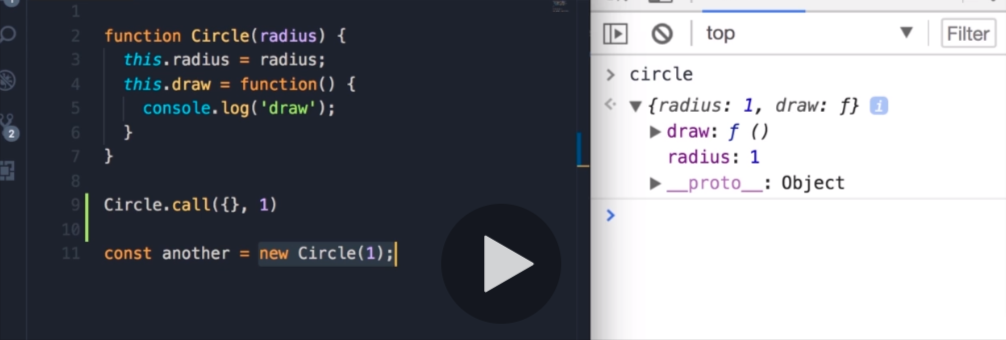
Every object has a constructor property and that references the function that was used to create that object.

**Functions are objects**



Here we see another built-in constructor called Function. When we declare a function using function name() {} syntax, internally JS engine will use this function constructor to create this object (Circle).





Line 9 and 11 are the same. By using call method we call a function (Circle), and this from inside the Circle function will reference {} from the call method.

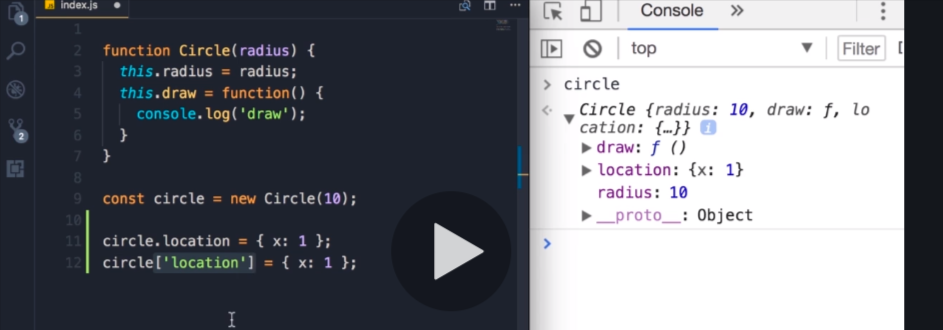
By using new operator, internally it will create a new object and pass it as the first argument to the call method. And this object will determine the context (this).

If you’re not using the new operator, then this will reference the global object window.

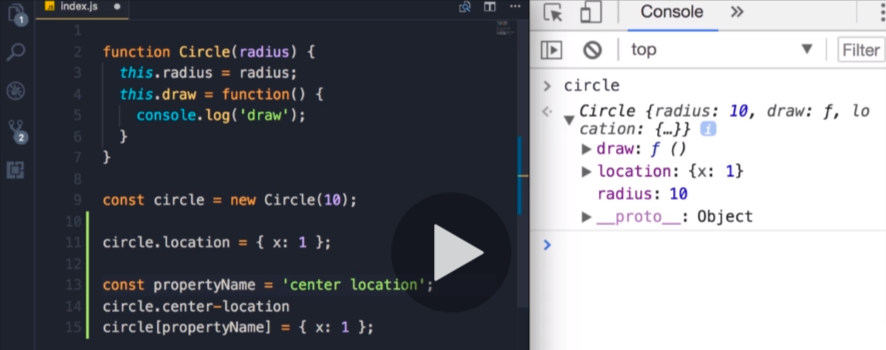
**The take away is, in JS functions are objects.**

**Adding or removing properties**

Every object created by the constructor functions are dynamic, which means that after creation you can add or delete properties from them. And you can do that whenever you like. This makes JS very powerfull and easy to work with. For such scenarios to work in Java you have to go back and change your class.



Use the bracket notation when you want to access dynamically a property name.



Another use-case is when you have property names that are not valid notation identifiers.

You can also delete properties. For example, when you have on the server a certain object that contains properties you don’t want to send to the client.

delete circle.location

**Abstraction**

Add some comlexity to our example.

Abstraction means, we should hide the details and show only the essentials.



In this example, we should hide defaultLocation and computeOptimumLocation. These are implementation details, part of the complexity of this object. We must hide them from the consumer of this onject. Instead we want to expose only the essentials.

If you refuse to comply with this concept, you’ll encounter a number of issues:

Every time you change the implimentation of that object, you have to modify many different places in your code.

**Private Properties and Methods**

By defining local variables and methods, you can easily hide them from the outside.



So the public interface of this object is simpler to work with.

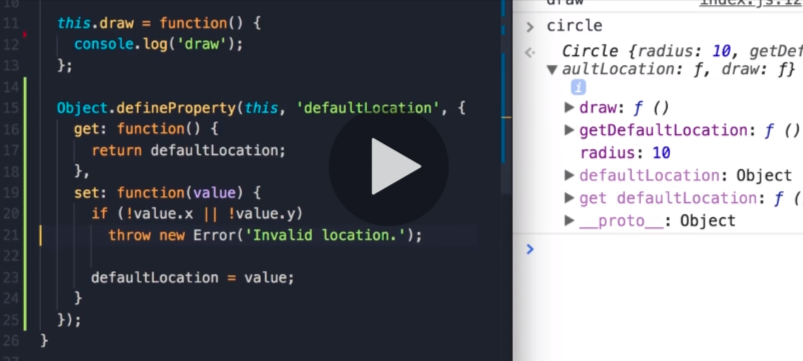
**Getters and setters**

defaultLocation and computeOptimumLocation are not private members of the circle object, they are just local variables. But what if somewhere in our application we want to access those variables?

We need a property from the outside which we can only read.

One way is using Object.defineProperty.

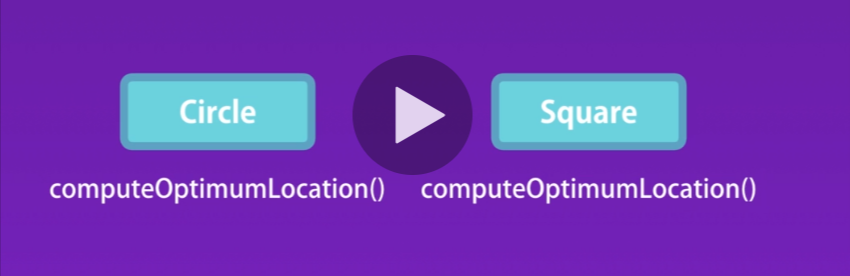
In case of set, we can define some validation before setting defaultLocation’s property value.



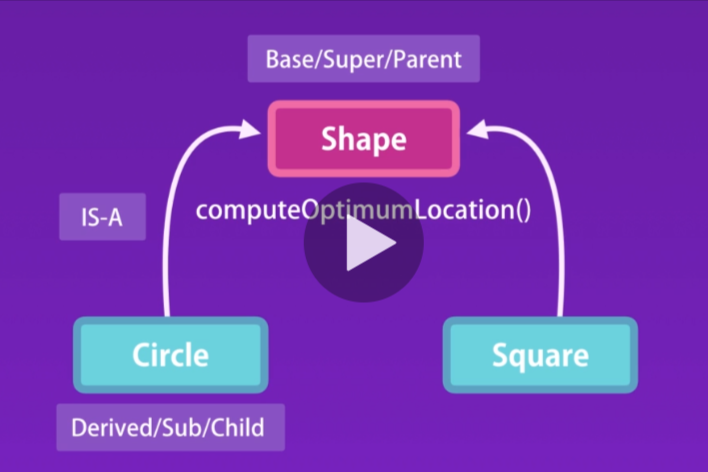
! Check positive and negative for setting a value.

**Inheritance**

Inheritance is one of the core concepts of OOP. It enables an object to take on the properties and methods of another object. Inheritance enables code reusability.



We can create a new class called Shape, put this method there and have Circle and Square classes inherit this method from the Shape class.



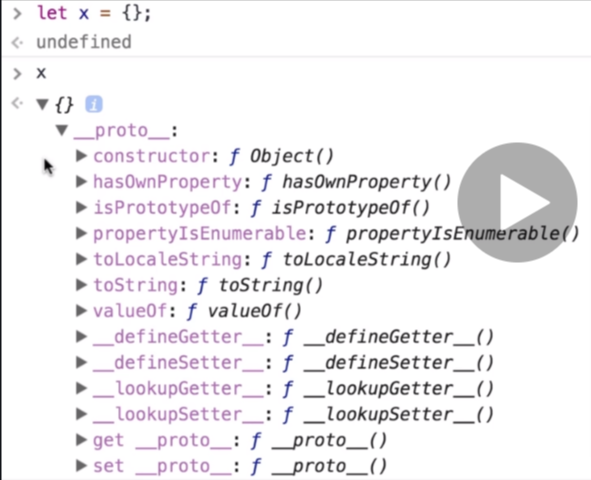
IS-A relationship – We say Circle is a Shape or Square is a Shape.

This is the classical definition of inheritance. But in JS we don’t have classes.

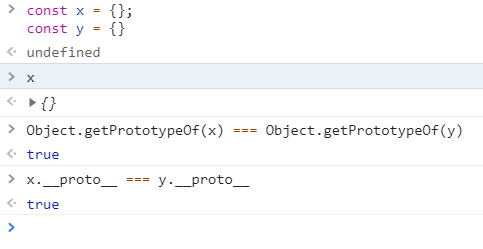
This is when prototypical inheritance comes into picture.

We refer to shape object as the prototype. So, a prototype is a parent of another object.

When you hear the word prototype, just think, parent.



Every object created in JS inherits from Object. Object is the root of all objects in JS and it doesn’t have a prototype/parent.



Because \_\_proto\_\_ is deprecated it is not recommended to use is in our code.

By calling x.toString() JS engine searches for this method on our object itself. If it cannot find it, it will look at the prototype of that object all the way up to the object base. So JS walks the prototype chain to find that method.

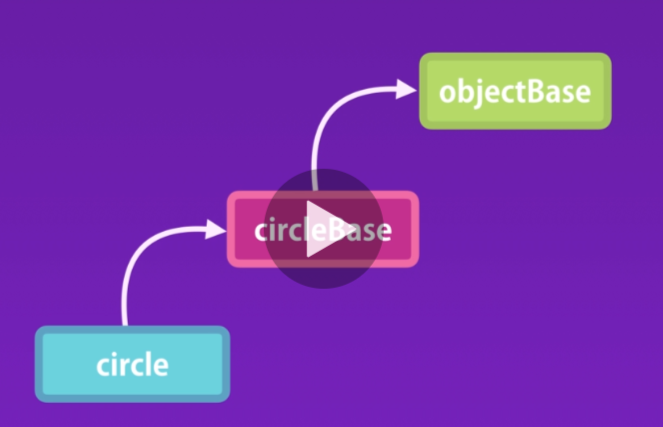


**Multi-level inheritance**



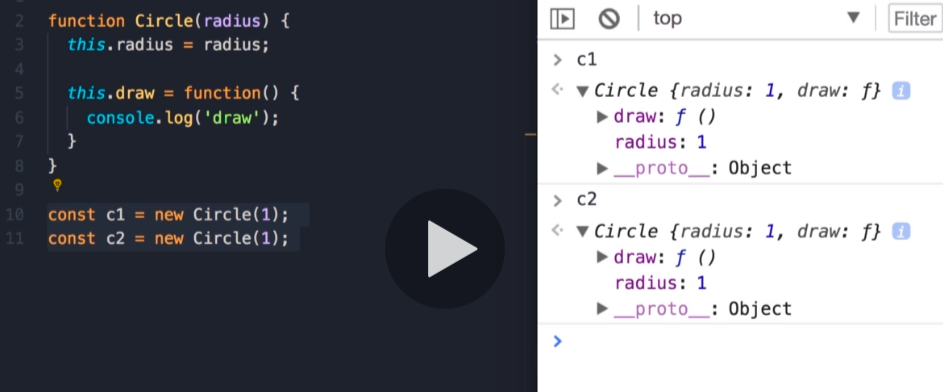
All circle objects created by the circle constructor will have the same prototype.

Circle object inherits from Circle base and Circle base inherits from Object base.



(another example is by creating an array, which inherits from Array, which itself inherits from root Object);

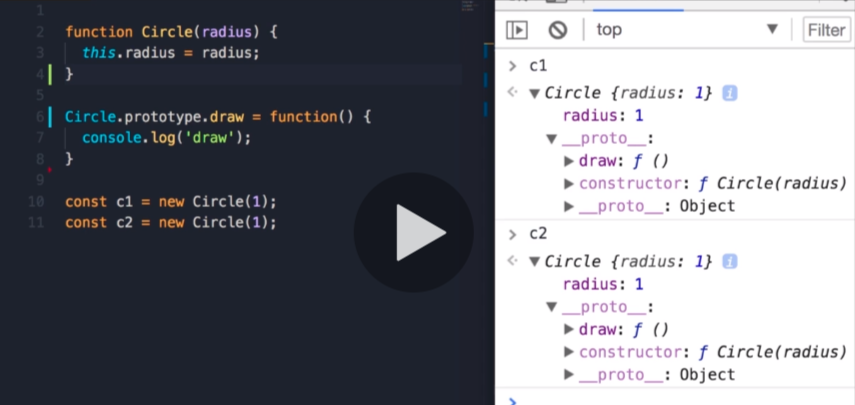
**Prototype vs Instance memebers**



Both c1 and c2 have a draw method. If we have 1000 circle objects in memory, we are going to have 1000 copies of the draw method. You are wasting a lot of memory by keeping copies of this method.

You know how prototypical inheritance works. When we access a property or a method on an object, JS engine looks up the object itself. If it cannot find that property/method, it will look at the prototype of that object.

So, we can take this draw method out of the Circle object and put it in its prototype. We are going to have a single instance of this method in memory.



So, we have two kinds of properties and methods ins JS:

Instance properties/methods - instance members, and prototype members.

Object.keys returns only instance members.

For in returns instance and prototype members (which are enumerable).

**Create your own prototypical inheritance.**

Add another method duplicate to the prototype of Circle. Let’s say we need another object square which needs duplicate method with exact same implementation. We don’t want to repeat the same method. Instead we want to use inheritance. We create a Shape object and put the duplicate method there and then have Circle and Square inherit from it.

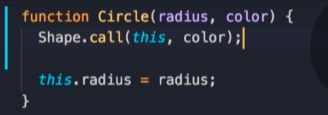


Object.create creates and object which inherits from Shape base. Before this change, the prototype of Circle was Circle.prototype = Object.create(Object.prototype).

This is prototypical inheritance in action. When you reset the prototype, make sure to reset the constructor as well.

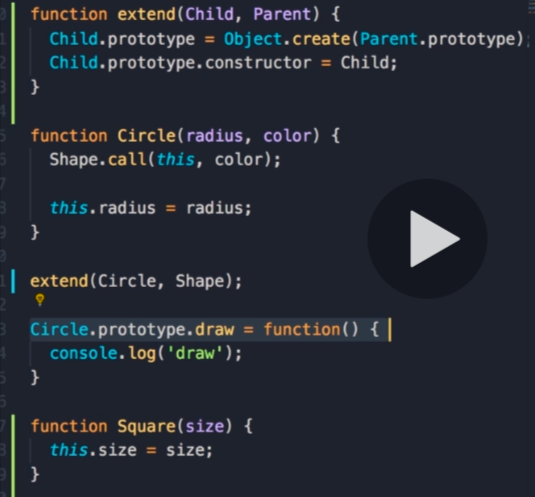
**Calling the super constructor**

Let’s assume that Shape receives a property called color. How do we set that property to the each new instance of Circle constructor?



**Intermediate function inheritance**

If we define another Constructor function which inherits from Shape, it is better to abstract that logic into a function. This is what we call intermediate function inheritance.

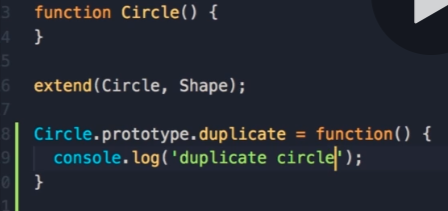


**Method overriding.**



There can be the case when implementation of duplicate method in Parent object is not ideal in a child object.

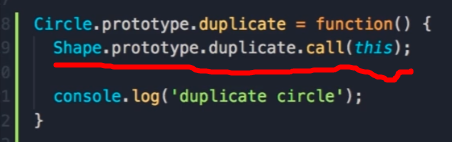
That’s when we use method overriding.



It’s very important to override it after extending the circle, because when calling extend we are resetting the prototype.

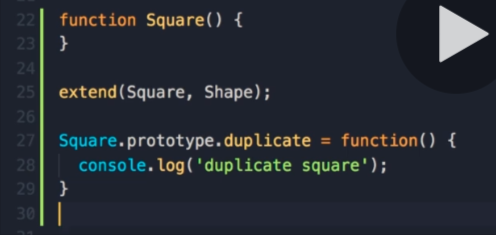
It works because of how PI in JS works. When we access a property/method on an object, JS engine walks the prototype chain and picks the first implementation.

Sometimes you may want to use the impl on the parent object as well.



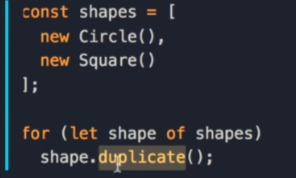
**Polymorphism**

Another very important and powerful concept in oop is Polymorphism.



So, now in the hierarchy we have on the top the Shape and two derivatives/child objects (Circle, Square). Each object will provide a different implementation of the duplicate method. So, we have many implementations/forms of the duplicate method. That’s what we call polymorphism.

If we create an array with different shapes and iterate over it, depending on the type of the shape object a different implementation/form of the duplicate method will be called.

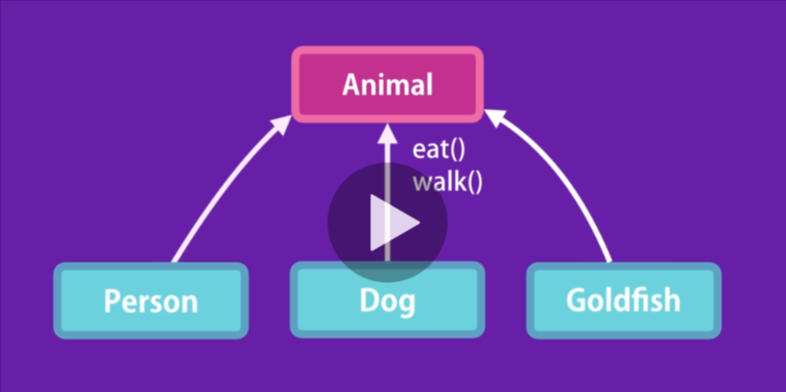


If not using the oop way, in this case polymorphism, we would have to add an if check for every type of object in the loop. In contract, when we encapsulate variables and functions into objects and use inheritance, we can execute many forms of a method with a single line of code. That’s polymorphism in action.

**When to use inheritance?**

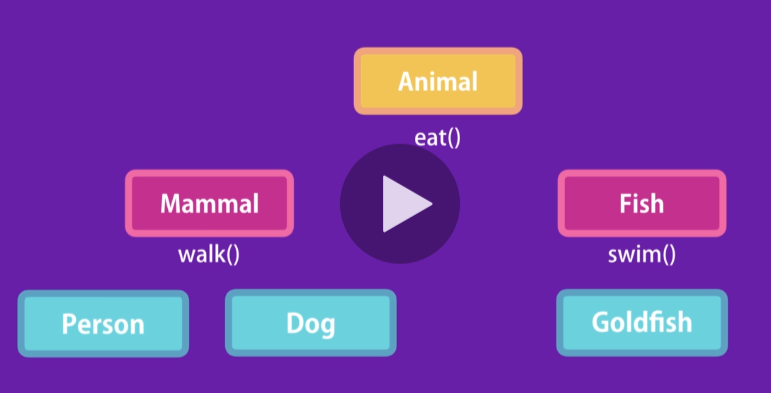
You have to be careful about using it. It can make your code complex and fragile. So don’t use it just for the sake of using it in a small project. Start with simple objects, then if you see a number of these objects share similar features, then perhaps you can encapsulate those features inside a more generic object and have your other objects inherit from it. But this is not the only solution to enable code reuse. These’s another technique called **composition**.

We have Animal object with two methods eat and walk. And we have two objects that derive from Animal: Person and Dog. If we add a new object Goldfish that derives also from Animal. We brake our code, because Goldfish cannot walk. It can swim.



To solve this problem, we need to change out hierarchy.

On top should be the Animal object with eat method. Under that Mammal and Fish. Mammals can walk, Fish can swim. And then Person and Dog derive from Mammal and Goldfish derive from Fish.



What would happen if we had 10 different types of animals? The hierarchy will get more and more complex and would have to go back and forth to determine the right place to implement a method.



If you want to use inheritance, keep it to one level.

There is a saying:

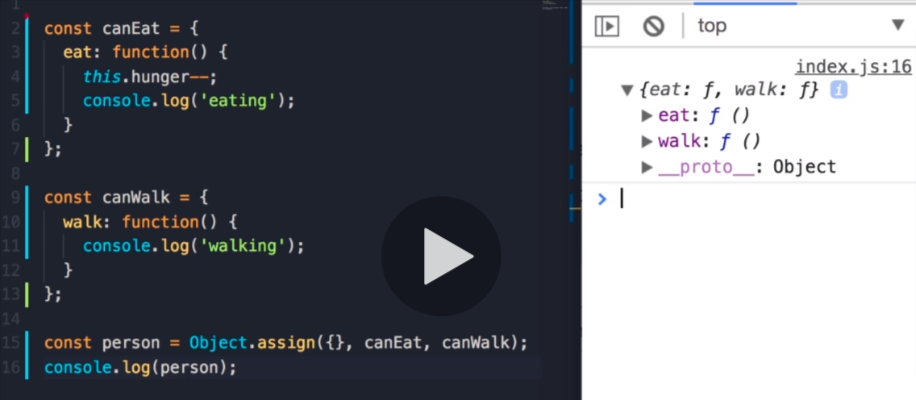


Instead of having a complex hierarchy, we can compose a few objects together to create a new object. This technique gives us great flexibility.

Instead of inheritance we can define various features for our animals as independent objects.

Each of these 3 objects are plain JS objects with certain properties and methods.We simply compose can walk and can eat to compose a Person object. If we have a Goldfish object, then we can compose can eat and swim together to produce it. We can combine any objects to create new objects. In JS we can use Mixins to achieve composition.

**Mixins**



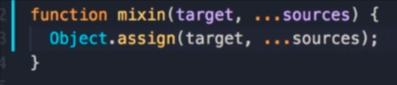
We can compose these objects together to create a Person that can eat and walk.

By using Object.assign we copy properties and methods from one object to another.



We can use this technique with a constructor function. Instead of an empty object, we pass Person.prototype.

So, composition or mixin gives us great flexibility.

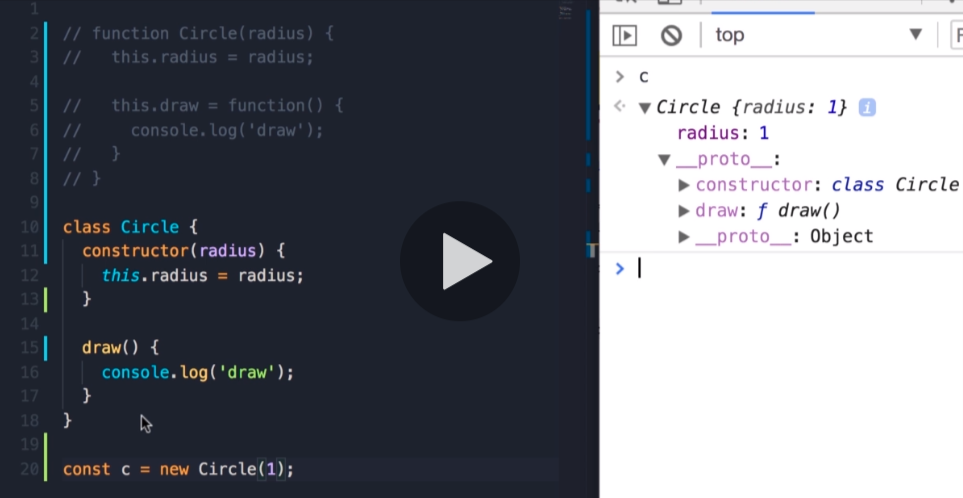


To make the code more readable, we can extract this logic into a function called mixin.

Ex. mixin(Goldfish.prototype, canEat, canSwim);

**Es6 Classes**

**ES2015** is a modern version of JS. There is a new way to create objects and implement inheritance. Using classes. But they are different than classes in Java or C#. In JS classes are syntactic sugar over prototypical inheritance. That’s why we learned about PI first, before looking at the new cleaner and simpler syntax.



To define a method in a class, we don’t use this.methodName in the constructor. Instead, methonds are declared in the body of this class.

Typeof Cicle is a function

**Static methods**

In classical OOP languages we have two types of methods: Instance methods and static methods.



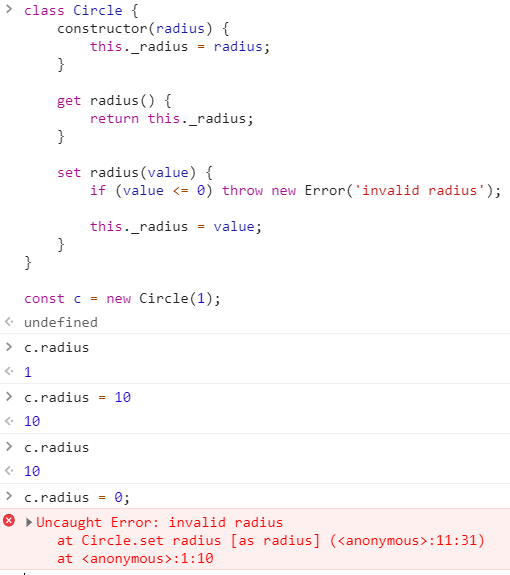
Instance because those methods are available on an instance of a class.

In contrast we have static methods. Static methods are available on the class itself, not the instance. They are used to create utility functions, that are not specific to a given object.

Static methods are accessible on the class reference. So to call static methods, we don't need to create an instance of a class.

**Getters and Setters**

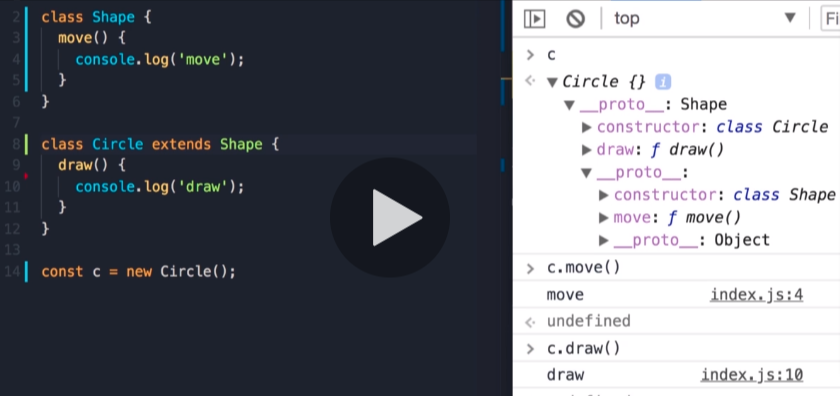
To implement a getter we implement a method which receives the keyword **get** in front of it. It looks like a method, but we can actually access it like a regular property.



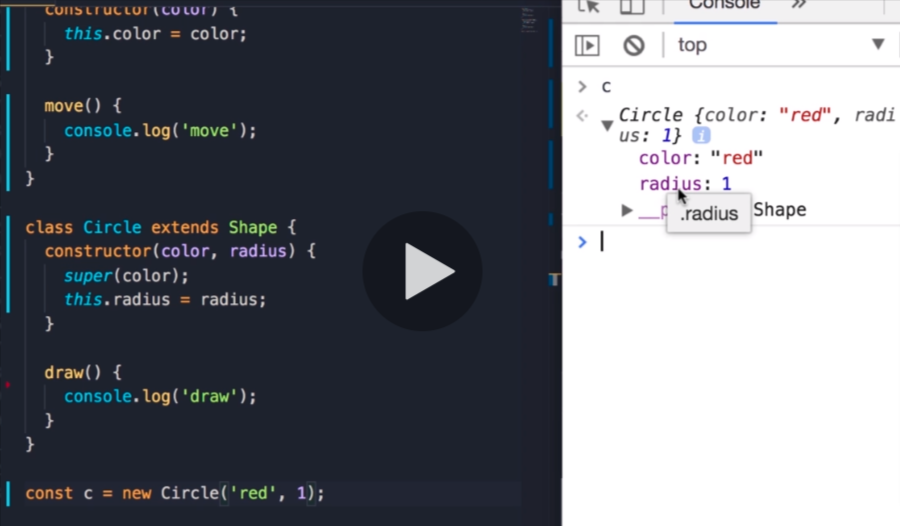


**Inheritance**

To have the Circle inherit from the Shape class, all we have to do is add extends Shape. We don’t have to reset the prototype and the constructor.



Let’s imagine all our shapes need a color. If you have a constructor in the Parent class and also have one in the child class, then inside the child constructor you should make sure to call the parent constructor first to initialize the base object. By using the super keyword we reference the parent object.



We inherited color from our parent class and we have radius on the circle class itself.