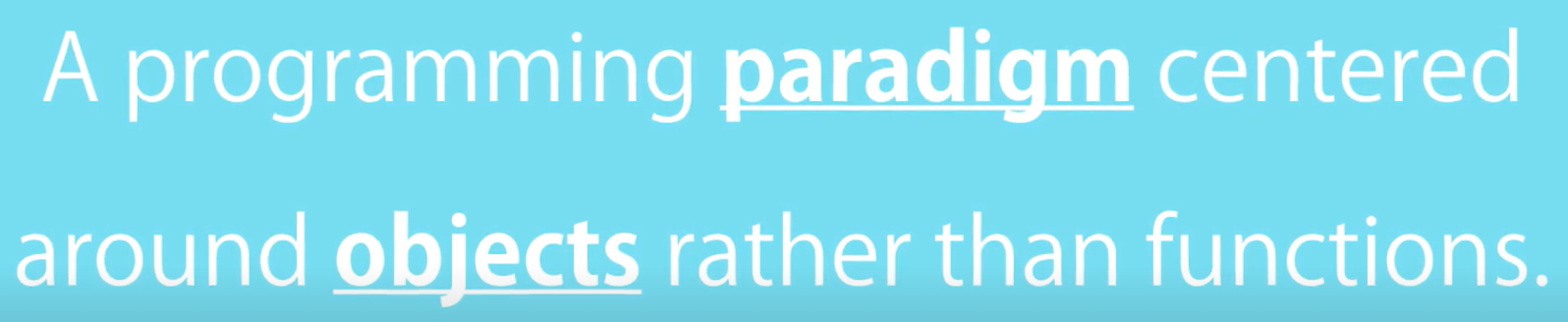
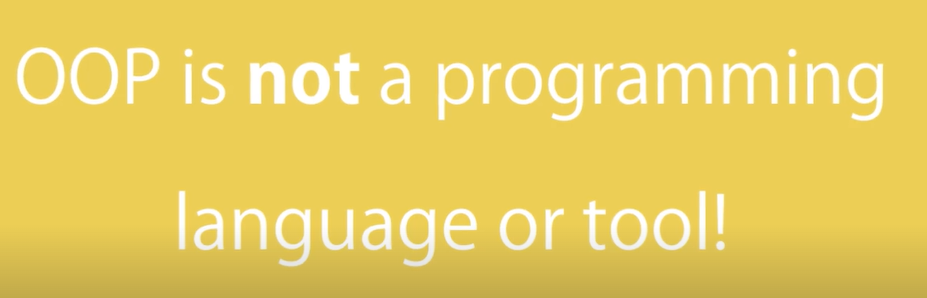


**What is OOP**:

Object Oriented Programming is:



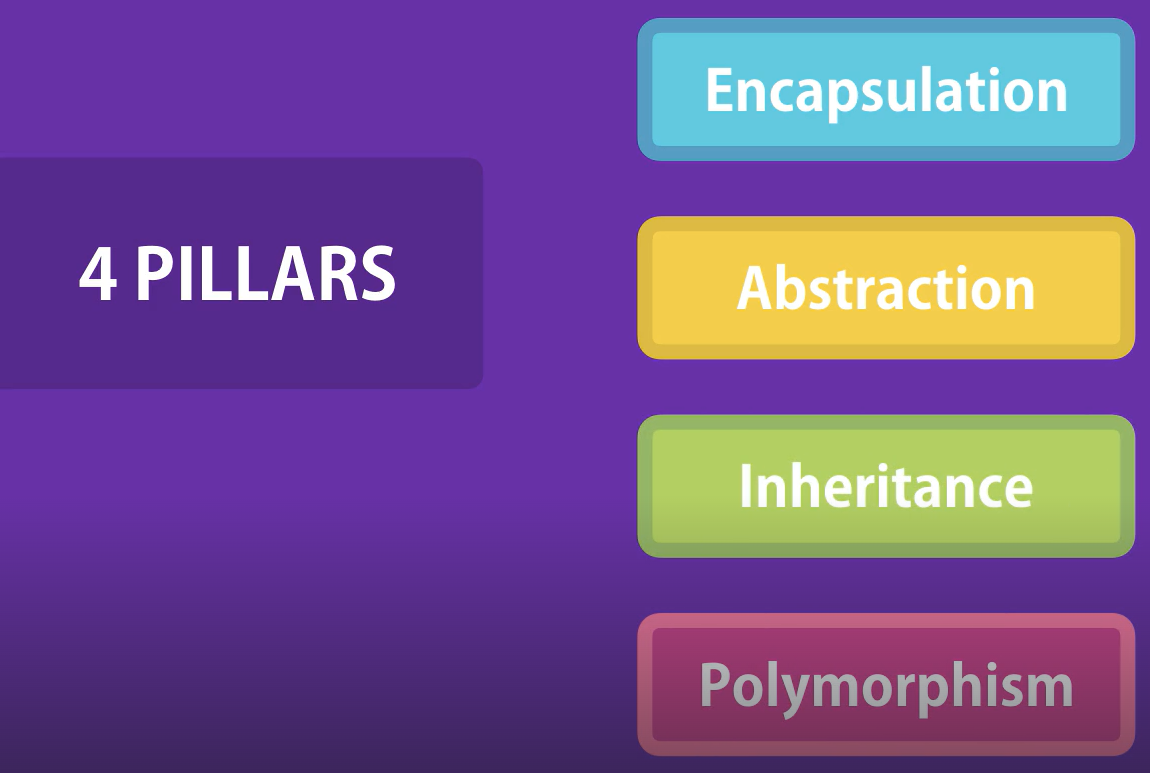
Programming languages and frameworks have come and gone but OOP stayed since the 70s, It is because



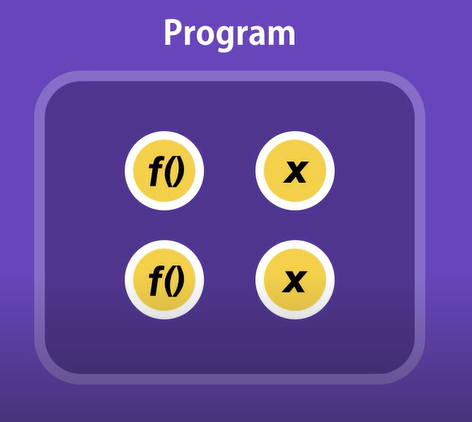
“*It is style of programming or a programming paradigm*”.

There are several programming languages out there that support object oriented programming such as C#, Java, Ruby, Python, JavaScript(*although controversial*) and more…

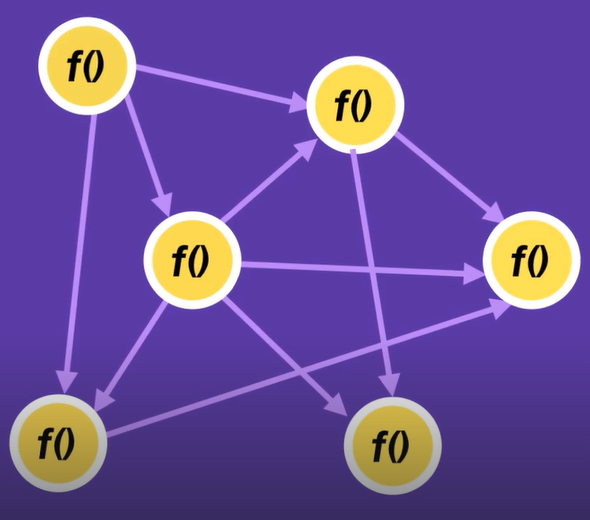
**4 Pillars of Object Oriented Programming**:

🡨Pillars of OOP

Before OOP we had **procedural programming**, that divided a program into a set of functions. So *we have data stored in a bunch of variables and functions that operate on the data*.

🡨 It is pretty simple and straight forward,

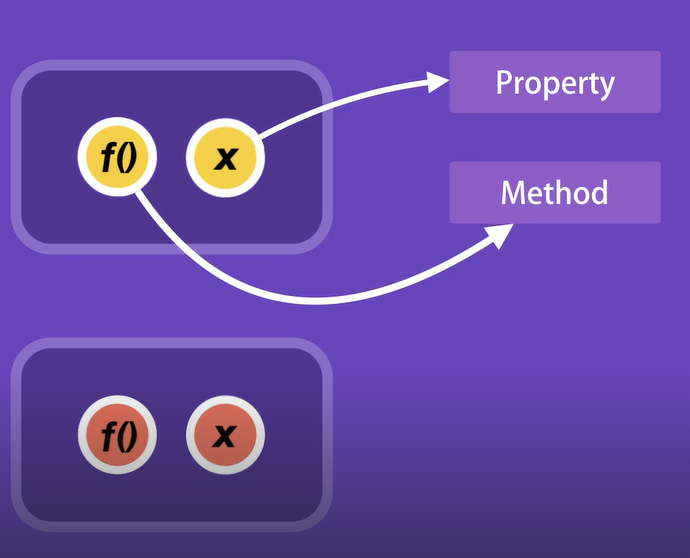
but as our programs grow, you will end up with a bunch of functions that are all over the place.

🡨 You will find yourself copying and pasting lines of code over and over to change one function and then several other functions break(*spaghetti code*).

There is so much interdependency between functions that it becomes problematic.

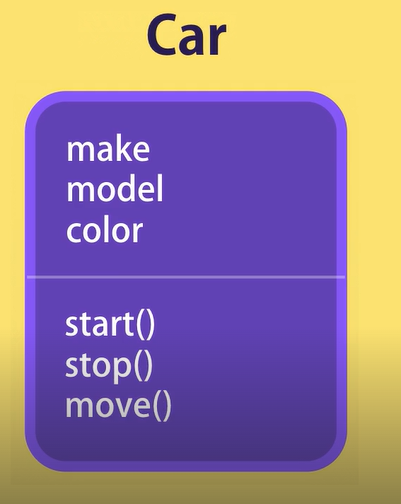
This is where ***Object Oriented programming*** came in picture.

In OOP, we combine a group of related variables and functions into a unit and we call that unit an object. We refer to these variables as properties and the functions as methods.



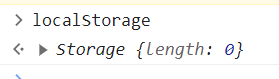
Here is a real world example of an object, a *Car*.

Car is an object with properties such as make, model and color and methods like start, stop and move.



*Real programming example*,

Think of the local storage object in your browser,



Every browser has a local storage object that allows you to store data locally. This local storage object has a property like *length* which returns the number of objects in the storage and methods like *setItem* and *removeItem*.

***What is encapsulation***?

In object related programming we *group related variables and functions that operate on them into objects* and this is what we call ***encapsulation***.

Look at this example which uses procedural programming as a way to calculate total salary of an employee.

let baseSalary = 30\_000;

let overtime = 10;

let rate = 20;

function getWage(baseSalary, overtime, rate) {

  return baseSalary + overtime \* rate;

}

console.log(getWage(baseSalary, overtime, rate)); //30200

Here the *variables and functions are decoupled*.

Now let us look at Object Oriented way of implementing this logic,

let employee = {

  baseSalary: 30\_000,

  overtime: 10,

  rate: 20,

  getWage: function () {

    return this.baseSalary + this.overtime \* this.rate;

  },

};

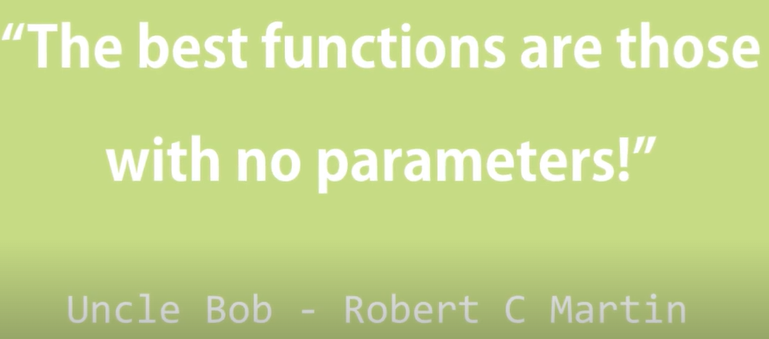
console.log(employee.getWage()); //30200

Here we created an employee object with three properties and one method called *getWage*().

Notice the getWage() method, this one has no parameters.

The reason is *all the parameters this method requires are modeled as properties of this object*. Since parameters and method are highly related, so they are part of one unit.

***“****When we start writing code in object oriented way, our functions start having fewer parameters****”***.



**The fewer the number of parameters, the easier it is to use and maintain that function**.

*What is abstraction*:

Think of a DVD player as an object, which has a complex logic board on the inside and few buttons outside to interact with. You simply press the play button and you do not care what happens on the inside. *All that complexity is hidden from you*. This is ***abstraction*** in practice.

We can use the same technique in our objects, where

***“****We can hide some of the properties and methods from the outside***”** and it gives us these benefits.



🡪 Interface of the objects become simpler.

Using and understanding an object with a few properties and methods is easier than an object with several properties and methods.

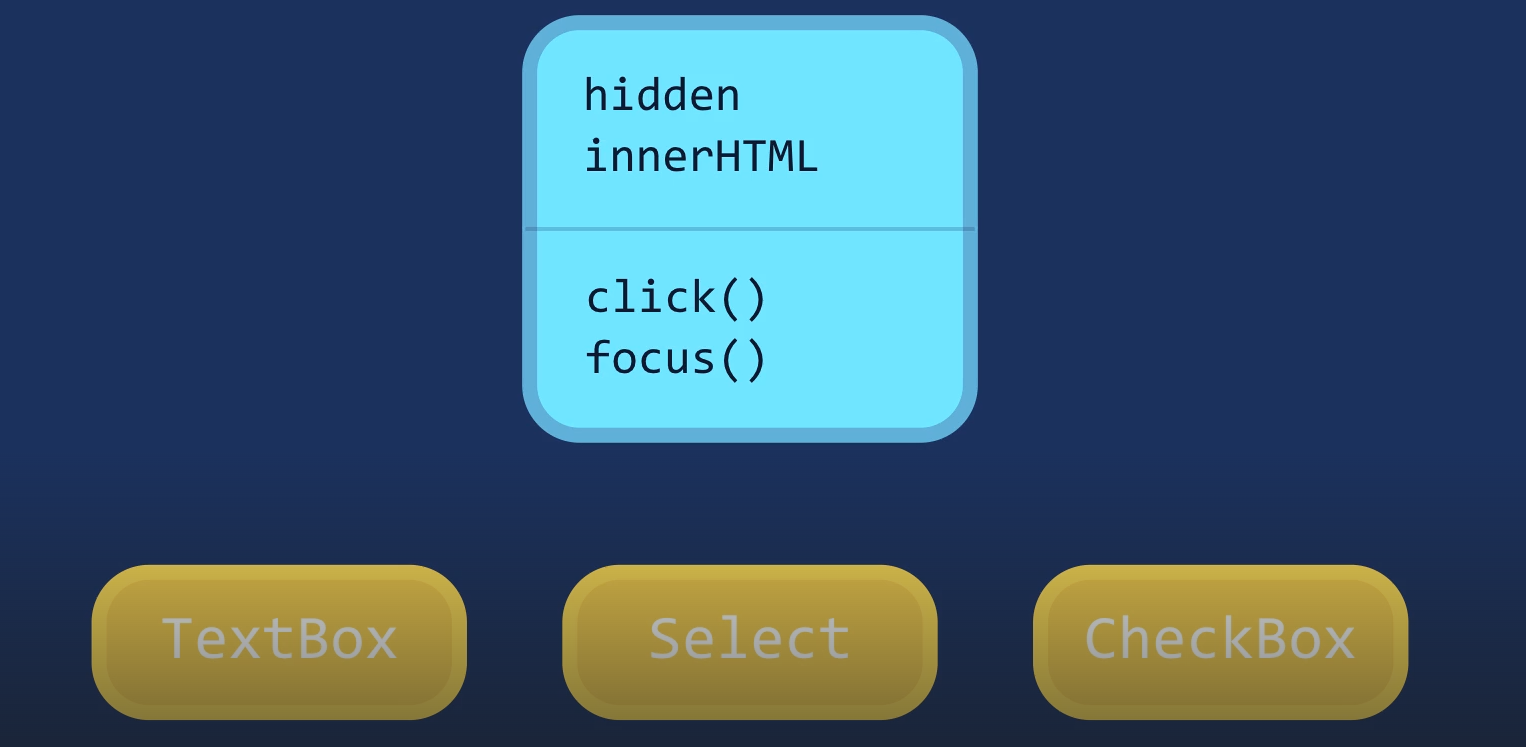
🡪 It helps us reduce the impact of change.

Let us imagine that tomorrow we change these inner or private methods. None of these changes will be leaked to outside. Because we do not have any code that touches these methods and outside of the containing object. You may delete a method or change its parameters, but none of the changes will affect the rest of the application code.

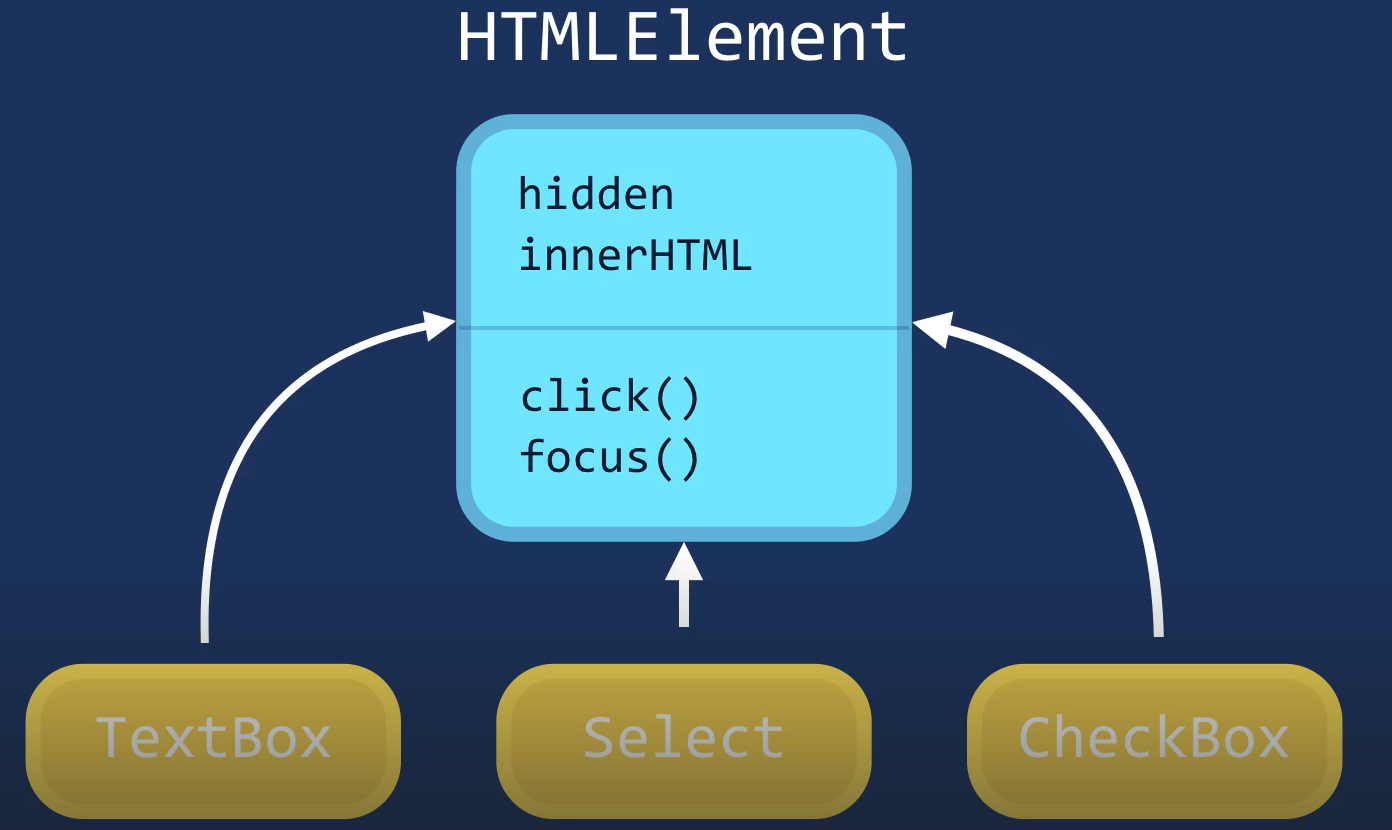
*What is* ***Inheritance***:

“*It is a mechanism that allows you to eliminate redundant code*”

Example, Think of HTML elements like Textbox, Dropdown, and Checkbox and so on. All these elements have a few things in common. They should have properties like *hidden*, *innerHTML* and methods like *click*() and *focus*().



Instead of redefining all these properties and methods for every type of HTML element, *we can define the once in a generic object call it* ***HTMLElement*** *and have other objects inherit these properties and methods*.



So Inheritance helps us eliminate redundant code.

*What is Polymorphism*:

Poly means many and morph means forms.



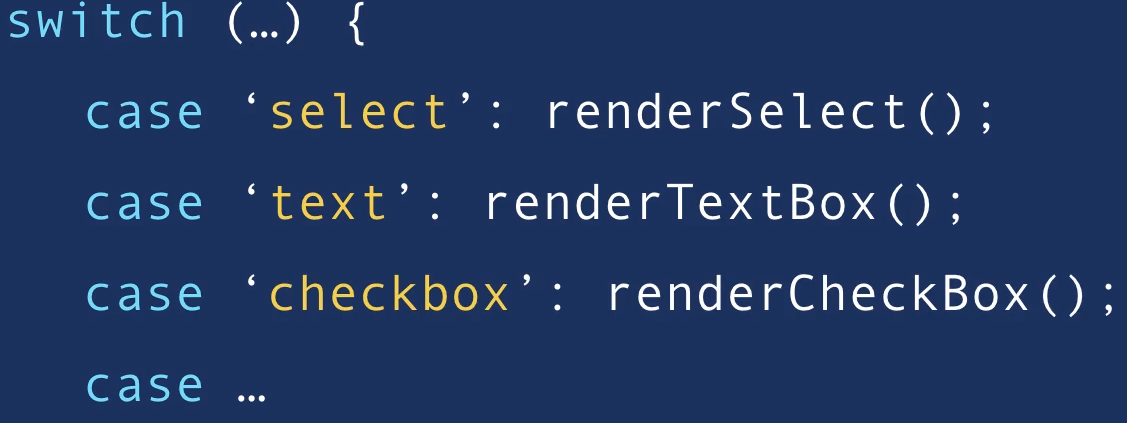
In OOP *polymorphism is a technique that allows you to get rid of very long if – else or switch – case statements*.

In our previous example, all these objects should have the ability to be rendered on a page.

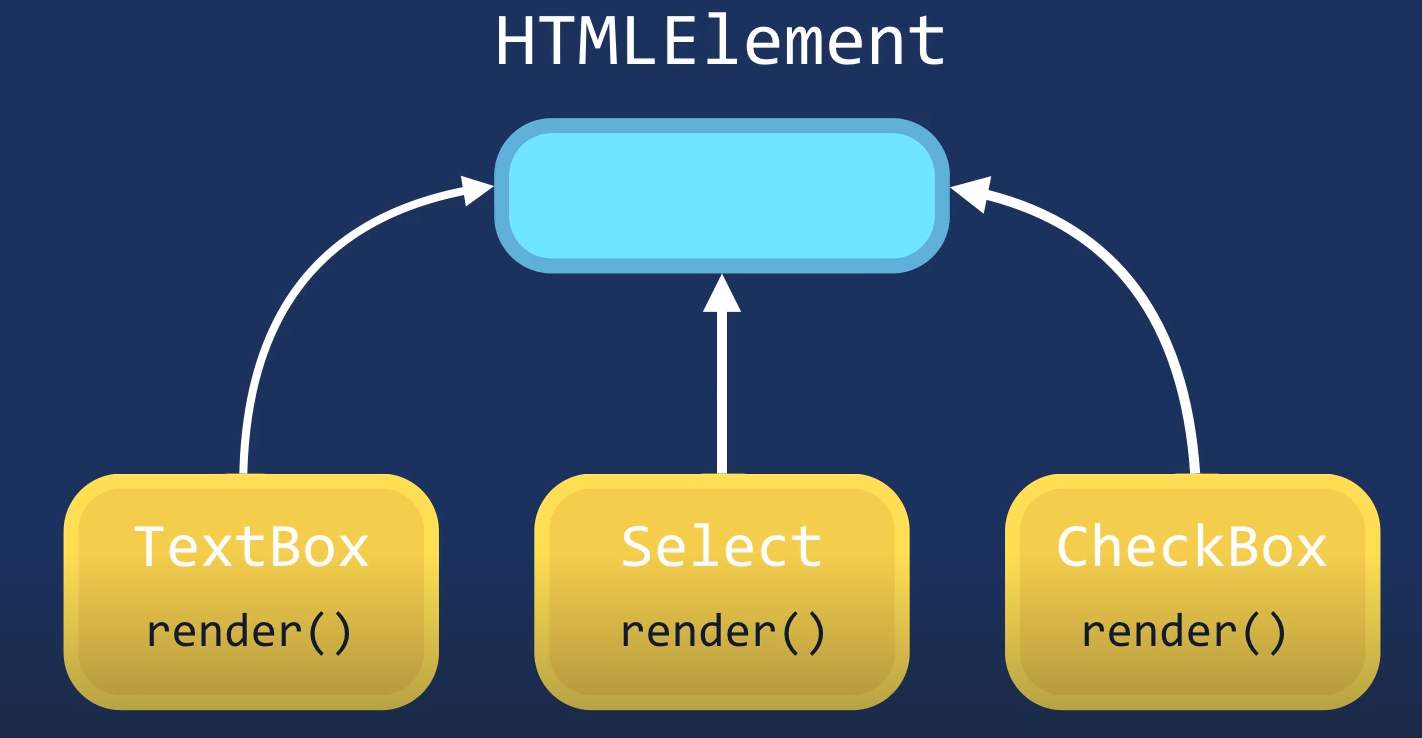


But the way each element is rendered is different from the others.

If you want to render multiple HTML elements in a procedural way, our code will look like this.



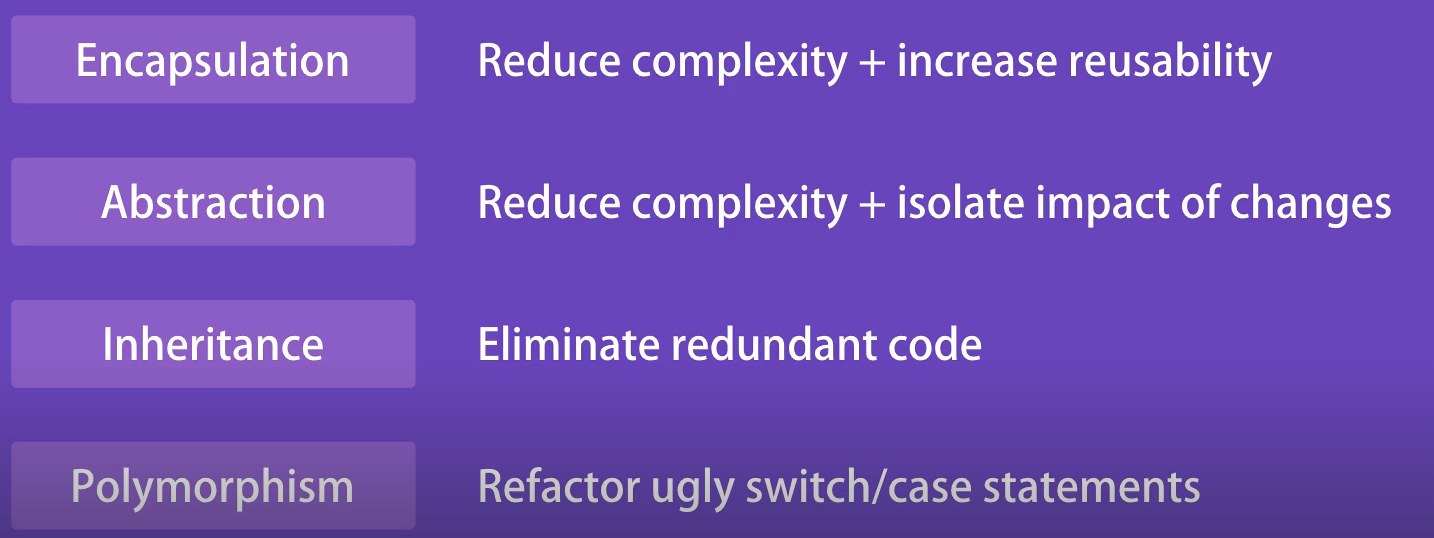
But with object orientation, **“***we can implement a render method in each of these objects and the render method will behave differently depending on the type of object we are referencing****”***.



So we can get rid of this nasty switch – case and write code in one line like this…



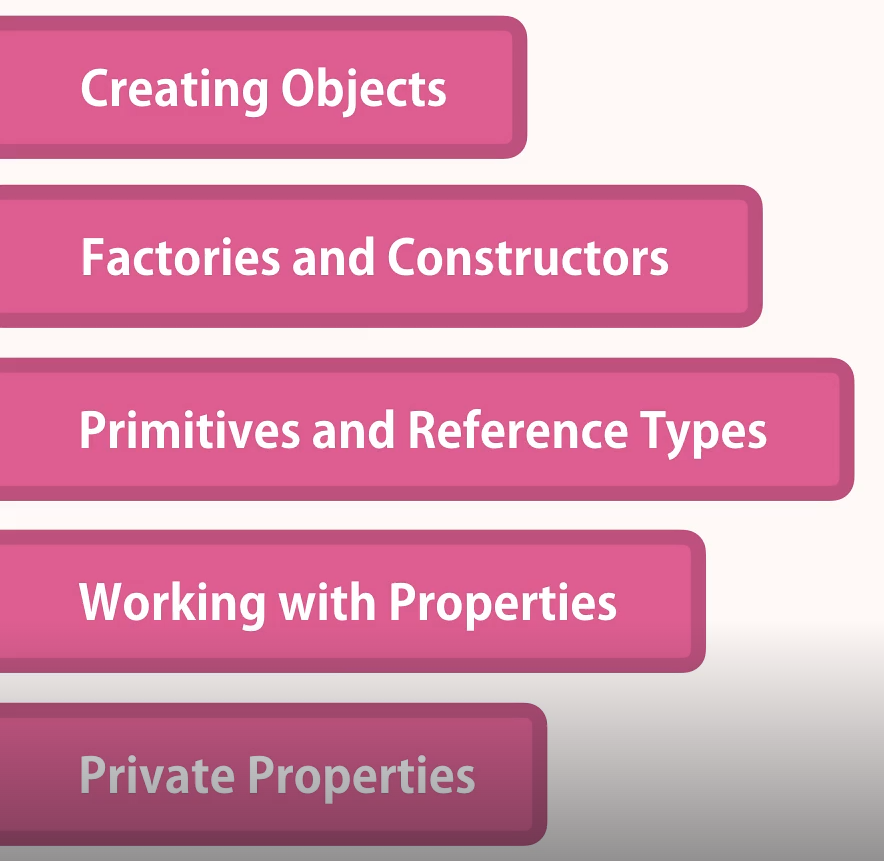
**Summary**:



**OBJECTS**

**Introduction**:

Things we will learn,



Getters/ Setters and so on…

**Object Literals**:

Let us declare a constant called *circle* and set it to an object.

const circle = {};

This way of declaring an object, by using curly braces { } is called an ***object literal syntax***.

***“****An object in JavaScript is essentially a collection of key – value pairs***”**

const circle = {

  radius: 1,

  location: { 🡪 Here the *key* “location” is set to an object

    x: 1, which further has x and y as its keys

    y: 1,

  },

};

We can even add a key inside our object, where we set its value to a function,

const circle = {

  radius: 1,

  location: {

    x: 1,

    y: 1,

  },

  draw: function () {

    console.log("draw");

  },

};

This *circle* object has three members. *radius*, *location* and *draw*.

*If a member is a function which define some logic, we refer to it as a* ***method*** *and the members which hold values are called* ***properties***.

Now we can access these object members using *dot* notation.

circle.draw(); //draw

Object literal { } is a simple way to define an object but we can also define an object using factories and constructors.

**Factories**:

Imagine, we want to create another circle, if we use object literal syntax we will need to duplicate the entire code.

Furthermore if there is a bug in our *draw* method in that case, we have to fix that in multiple places.

Note: If an object has one or more method, we call that object has a *behavior* like a person. So if an object has a *behavior*, **do not** **duplicate** it using object literal syntax.

The solution is to use a ***Factory function***,

function createCircle(radius) { 🡪 This is a Factory function

  return {

    radius, 🡪 since radius: radius (key – value are same)

    draw: function () {

      console.log("draw");

    },

  };

}

Note: In ES6 we have a feature where if the key and value are same, we can simplify the code by just writing key.

Now, we can call this function simply and store its returned object like this

let circle = createCircle(1);

circle.draw(); //draw

**Constructors**:

Another way of creating objects is by using ***constructors***. The first difference you will see between a constructor and a normal function is the naming convention which is *Pascal Convention*.

function Circle(){ 🡪 *Pascal convention* in function name means **Constructor**

}

Inside constructor function we use *this* keyword to set the properties of this object(*where this is basically a reference to the object that is executing this piece of code*).

So imagine we have a new empty object in memory now we are going to use *this* *to reference that object* and then use *dot* notation *to set various properties on that object*.

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

    console.log("draw");

  };

}

Here we set *radius* property to this *radius* argument and similarly we can define a method called *draw* and set it to a function.

Now, we can create a new *circle* object using ***new*** operator on this constructor function like this,

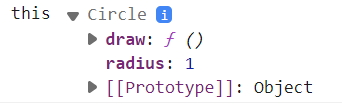
const circle2 = new Circle(1);

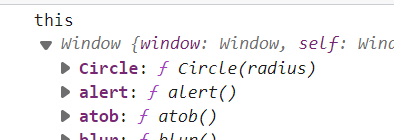
circle2.draw(); //draw

When we use *new* operator, a few things happen under the hood,

🡪 It creates an empty object. { }

🡪 Then it set *this* to point to that object(*because by default this points to global object in node and window object in browsers*).

🡨 If *new* operator used

🡨 If *new* operator **not** used.

🡪 Finally it will return an object from constructor function.

Note: We do not have an explicit *return* statement from the constructor function saying *return this*;, it happens automatically when we use the ***new*** operator.

**Constructor Property**:

Every object in JavaScript has a property called ***constructor*** which *references the function that was used to construct or create that object*.

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

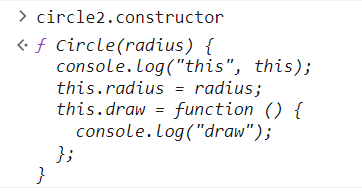
    console.log("draw");

  };

}

const circle2 = new Circle(1);

Let us look at constructor property of *circle2* object.



As we can see, it returns our ***Circle*** function that we used to create this object.

Now Let us look at another object,

function createCircle(radius) {

  return {

    radius,

    draw: function () {

      console.log("draw");

    },

  };

}

let circle = createCircle(1);

and its constructor property,



Judging from *f* and uppercase name, we can tell that *this is a built in constructor function in JavaScript*.

*When we create an object using object literal syntax, internally the JavaScript engine used this constructor function*.

Here is an example, let us create an empty object,

let x = {};

Internally, JavaScript engine will translate it to,

let x = new Object();

So this circle object that we created and returned from Factory function uses this ***Object*** constructor function.

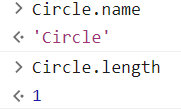


**Functions are Objects**:

One of the confusing concepts in JavaScript is that functions are objects!

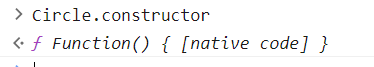
When you use dot operator on a function, you will see properties and methods available for object.

 Note: Purple icons are *methods* and blue icons are *properties*.

🡨 Circle.length returns number of arguments.

As we know that every JavaScript object has a constructor property which references to the function that was used to create that object.

*Who do you think created this* ***Circle*** *object*?



Here we have another built in constructor called Function. When we declare our very own constructor function, internally JavaScript engine will use Function constructor to create this object.

Let us construct a constructor function using Function constructor,

const Circle1Fun = new Function(

  "radius",

  ` this.radius = radius;

this.draw = function () {

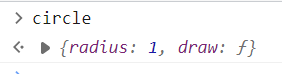
  console.log("Draw");

};`

);

const circle = new Circle1Fun(1);

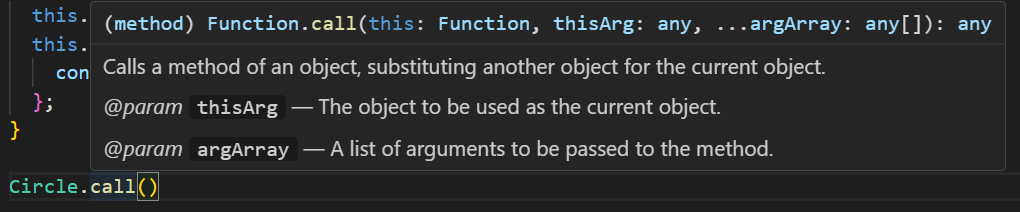
First argument is radius parameter and second argument is body of the function.

We created a real circle object with two parameters.

*Some other methods available in our functions*:

call method:

With the call method, we can call a function. Look at the arguments,



First argument is *this* argument.

Here we need to pass an empty object {} and *this* will reference the object that we passed.

Circle.call({})

After that we add our arguments explicitly,

Circle.call({}, 1);

Note: Both these expressions are exactly the same

Circle.call({}, 1);

new Circle(1);

When we use *new* operator, it will internally create an empty object {} and pass that as the first argument of call method which will determine the context of this.

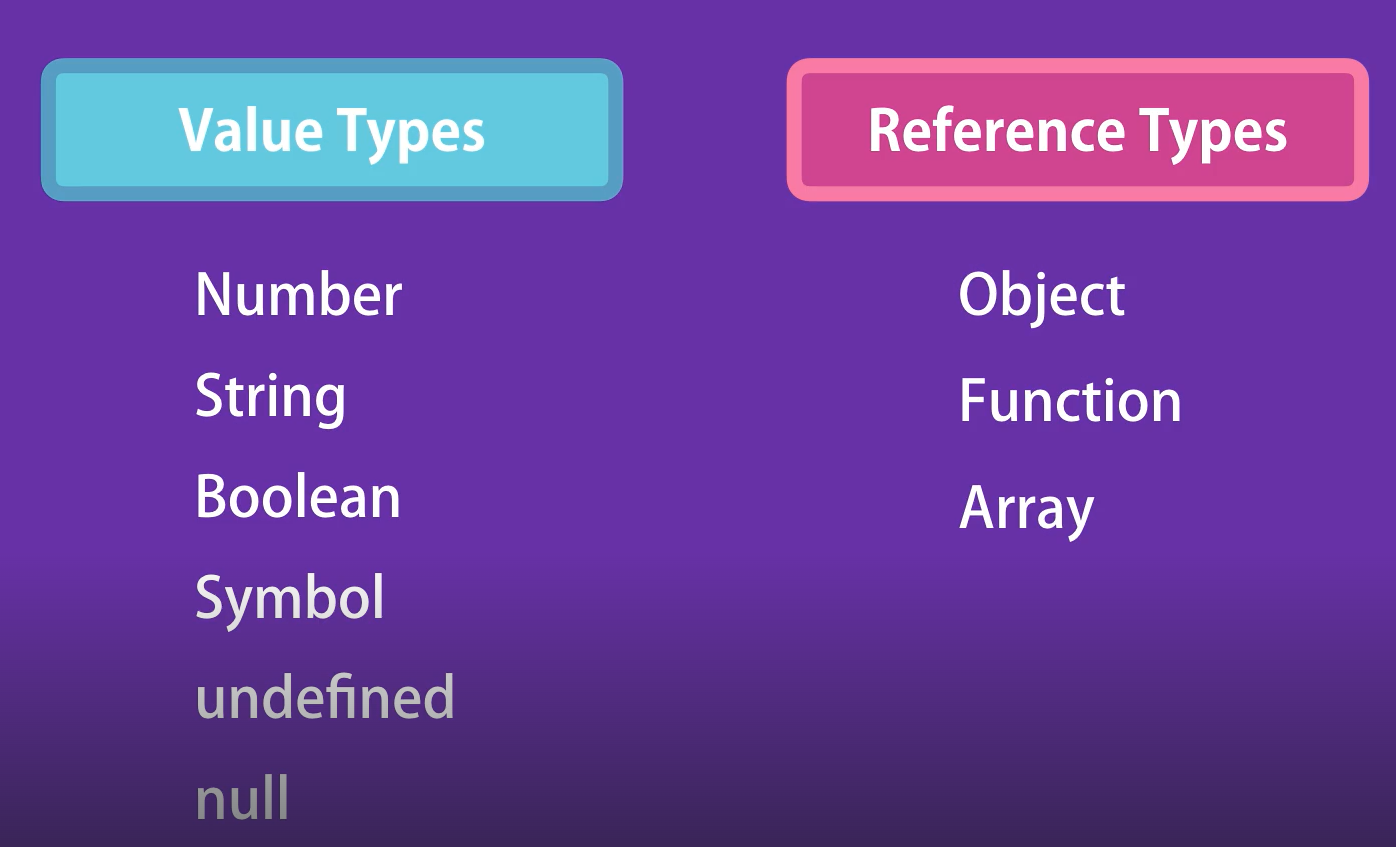
apply method:

Similar to call method, we have a apply method, Only difference is that instead of passing arguments explicitly we can pass them in form of an array.

Circle.apply({}, [1,2,3]);

**Value vs Reference types**:

In JavaScript we have two categories of types. On one side we have Value types called primitives and on other side we have Reference types.



In last lecture we learned that functions are objects, same is true for arrays as well. So concisely we have *primitives and objects*. We need to learn how both of these types behave differently.

*primitive types example*:

let x = 10;

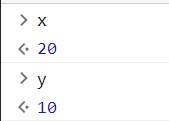
let y = x;

x = 20;

What should be the value of y?

Hint: Both x and y are *independent* variables.

Result:



Explanation:

The value 10 was stored initially inside variable X and when we copied that variable, the value that was stored 🡪10 was copied in to the new variable Y.

So both these variables X and Y are independent of each other.

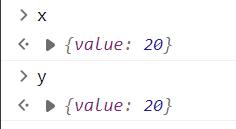
*reference / objects example*:

let x = { value: 10 };

let y = x;

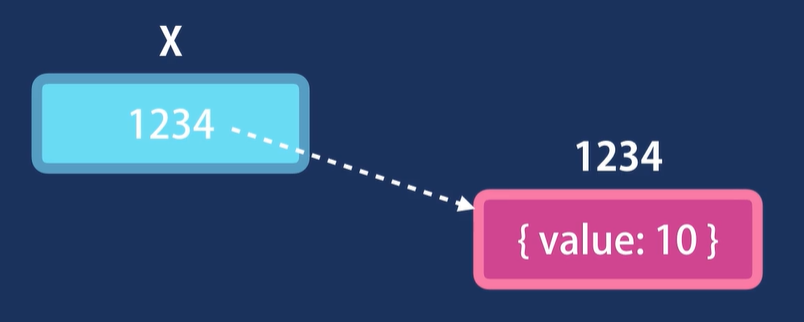
x.value = 20;

Result:

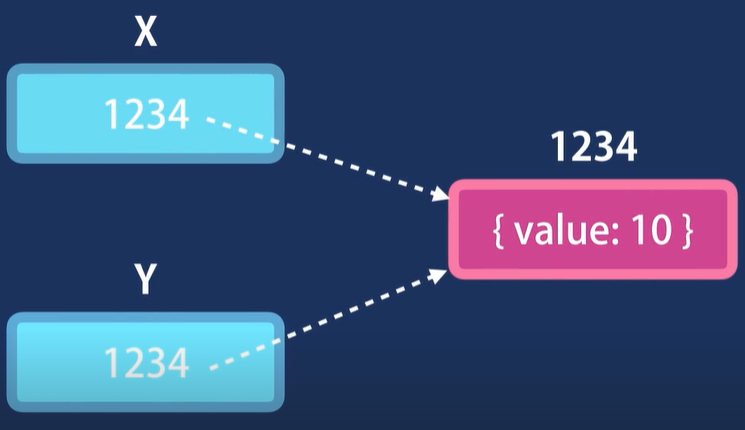


Explanation:

When we use an object {value: 10} that object is not stored in the variable X. That object is stored somewhere else in the memory and the address of that memory location is stored inside that variable.

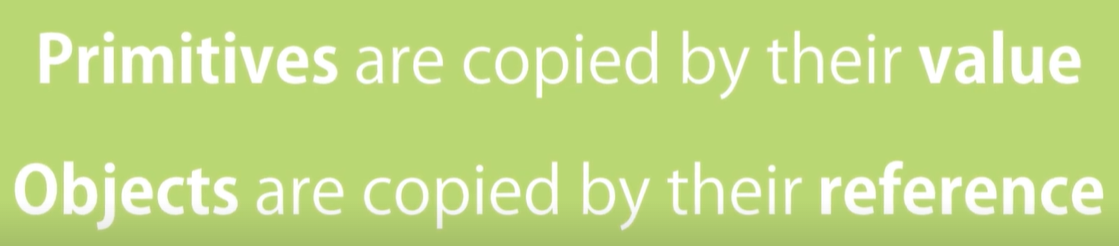


So when we copy X in to Y, it is the address of the reference that is copied. *In other words both X and Y are pointing to the same object in memory*.



And when we modify that object by either X or Y, our changes are immediately visible to the other variables.

*Conclusion*:



*Another example of primitive types*:

let number = 10;

function increase(number) {

  number++;

}

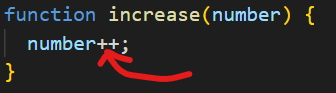
increase(number);

console.log(number);

What should be the value of number?

Solution: 10

Explanation: When we call *increase* function and pass this *number* variable, its value is copied into the parameter which is *local* in this function.

So this number 🡪 

and this number 🡪 

are completely independent of each other. Inside the function value of number is 11 but outside the scope of this function value of number is still 10.

*Another example of reference/object types*:

let obj = { value: 10 };

function increase(obj) {

  obj.value++;

}

increase(obj);

console.log(obj);

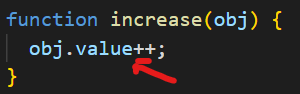
What should be the value of *obj* object now ?

Solution:

{ value: 11 }

Explanation:

It is because when we call increase function and pass this object *obj*, this object is passed by its reference.

So local parameter 🡪 

and 

are same.

Here we are not dealing with two independent copies instead we have two variables that are pointing to the same object.

Any changes we make to this object (**let obj = { value: 10 };**) will be visible to the other variable.

**Adding or Removing Properties**:

Here we have our Circle constructor function and using that we created a circle object.

function Circle(radius) {

  this.radius = radius;

  this.draw = function () {

    console.log("draw");

  };

}

const circle = new Circle(10);

Now these objects in JavaScript are dynamic which means after creating them we can add extra properties or delete existing ones.

*Real world example*:

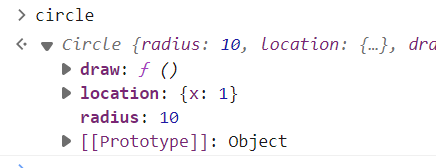
Imagine you are working with a *user* object and the client (*web or mobile application*) is going to send a user object to the server.

On the server, we get this *user* object and we add additional stuff to it, for example we can add a token property that we generate on the fly on the server. So we can always add something extra to an existing object.

Here we can add a new property to our *circle* object called *location* like this,

circle.location = { x: 1 };

Now on the console we can see our *location* property as well.



We have another way to access object’s properties using ***Bracket notation***.

circle["location"] = { x: 1 };

As you can see , dot notation is simpler and less verbose then bracket notation.

But in some scenarios bracket notation is more useful. For example *when we want to dynamically access a property name*.

Second, if you have special characters in a property name (*like center- location*) *which are not valid identifiers*. We cannot use dot notation there.

circle.center-location //will get an error if use dot operator

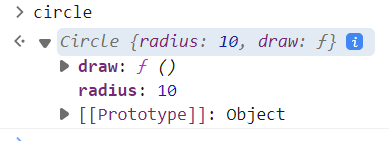
circle['center-location']  //it is OK, we can use bracket notation

We can also delete a property from an existing object,

For example when we get a user object from database and we want to return it to the client but only after removing certain properties from that object like password, credit card info etc.

In that case we will dynamically delete one or more properties from an object by using ***delete*** operator.

delete circle.location;



**Enumerating properties of an Object**:

Here we have circle object with radius property and draw method.

const circle = {

  radius: 1,

  draw() {

    console.log("draw");

  },

};

Earlier in the course we learned how to use *for – in* , *for – of* loops to iterate over the properties in object.

for (let key in circle) {

  console.log(key, circle[key]);

}

We get the name of the key and its corresponding value.



If we want to get only the properties and not methods, we can use *typeof* operator to check the type of this value,

for (let key in circle) {

  if (typeof circle[key] !== "function") {

    console.log(key, circle[key]); // radius 1

  }

}

We can use a *for – of* loop with iterables, *but a JavaScript Object is not iterable*,

for (let keys of circle) {

  console.log(keys);

}

TypeError: circle is not iterable

So instead we use Object.keys method, which returns an array and *Arrays are iterable*.

for (let keys of Object.keys(circle)) {

  console.log(keys); //radius draw

}

*What happens in background of Object.keys method*?

Since *Object* is a built in constructor function and each function in JavaScript is an object. So this *Object* has properties and methods that we can access using dot notation.

Another useful method of *Object* is ***entries***,

for (let entry of Object.entries(circle)) {

  console.log(entry);

}

This method returns each key – value pair as an array.

O/P:

[ 'radius', 1 ]

[ 'draw', [Function: draw] ]

Sometimes we want to check if an object has a given property or a method, we can use *in* operator,

if ("radius" in circle) {

  console.log("Yes"); //Yes

}

**Abstraction**:

Now let us introduce some complexity in our example.

function Circle(radius) {

  this.radius = radius;

  this.defaultLocation = { x: 0, y: 0};

  this.computeOptimumLocation = function () {

    // here we do something...

  };

  this.draw = function () {

this.computeOptimumLocation();

    console.log("draw");

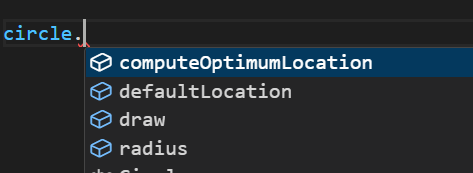
  };

}

const circle = new Circle(1);

Here I defined a new property called *defaultLocation* and set it to an object and a method *computeOptimumLocation* which we call inside *draw* method

There is a problem with implementation like this. Here we can see members of the *circle* object.



Not all of these methods should be accessible to consumer or client of this object.

*The Issue is*:

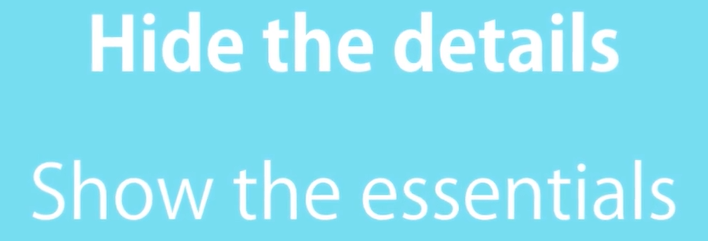
If I set this *defaultLocation* to false, it will completely mess up this object or this *computeOptimumLocation* method should only be called inside *draw* method.

So If I accidently call this method like this,

circle.computeOptimumLocation();

It might put the object in a bad state.

So in OOP we have this core concept called ***abstraction*** which means,



In this particular example, we should hide *defaultLocation* property and *computeOptimumLocation* method. These are implementation details which are part of the complexity of this object.

*We should hide these from the consumers of this object and expose only the essentials*.

Imagine if this *computeOptimumLocation* method requires a *factor* argument,

 this.computeOptimumLocation = function (factor) {

    // here we do something...

  };

Now everywhere we have used this method directly like this,

circle.computeOptimumLocation(3.14);

we have to pass an argument for this method.

In contrast if this method was not accessible from the outside then we would not have to modify this everywhere. We would just modify here inside *draw* method,

  this.draw = function () {

    this.computeOptimumLocation(3.14); 🡨

    console.log("draw");

  };

}

**Private Properties and Methods**:

So how can we hide certain members from outside? Earlier we learned that *this* refers to the current object.

So if I declare a *local* variable like this,

function Circle(radius) {

  this.radius = radius;

  let defaultLocation = { x: 1 }; 🡪 Local variable

It would not be the part of the object, because we have not set this variable as a *property*. So it is just a local variable inside this function, when we get out of the function this variable will goes out of scope and dies.

With this technique we can easily hide certain members from the outside.

With the same token we can convert our method into a private method like this,

  let computeOptimumLocation = function (factor) {

    // here we do something...

  };

Now we can access this private property and method inside our *draw* method without using *this* keyword.

This works because we have a concept of ***closure*** in JavaScript.

function Circle(radius) { 🡪 Parent function

  this.radius = radius;

  let defaultLocation = { x: 1 };

  let computeOptimumLocation = function (factor) {

    // here we do something...

  };

  this.draw = function () { 🡪 inner function

    defaultLocation;

    computeOptimumLocation();

    console.log("draw");

  };

}

Here we have a function inside another function. In inner function we can declare certain variables like x and y.

  this.draw = function () {

    let x, y; 🡪 local variables

These local variables are only available inside this function since their scope is limited to this function. So when we stop executing this function x and y will go out of scope.

In contrast to scope, we have *closure*. *It determines what variables will be available to the inner function*. So *this inner function will be able to access all the local variables as well as variables defined in in its parent function*.

Note: Do not confuse scope with closure because scope is temporary and it dies. Since every time we call draw method x and y will be recreated and reinitialized and after this function they will die.

  this.draw = function () {

    let x, y; 🡪 will die after execution completes

    defaultLocation; 🡪 will stay in memory

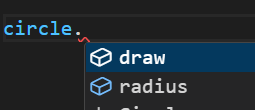
    computeOptimumLocation(0.1); 🡪 will stay in memory

    console.log("draw");

  };

But closure stays there even after executing this function variables like defaultLocation and computeOptimumLocation will continue to stay in memory. They will preserve their state because they are part of the closure of this *draw* function.

Now if we access members of this object, we only see,



So the public interface of this object is simpler and easier to work with and will prevent issues down the road.

**Getters and Setters**:

In last lecture we converted these members, *defaultLocation* and *computeOptimumLocation* into private members.

But more accurately these are not private members of the *circle* object, these are local variables that we have defined inside *Circle* constructor function. But from object oriented programming point of view, we can refer to them as private members of the *circle* object.

Let us simplify the code and focus on *defaultLocation* property.

function Circle(radius) {

  this.radius = radius;

  let defaultLocation = { x: 1 };

  this.draw = function () {

    console.log("draw");

  };

}

const circle = new Circle(1);

We cannot access this property from outside using dot notation.

But what if we want to display this *defaultLocation* somewhere in our application. Note here that *we do not want to modify it but just read it*.

One way of doing it is *by creating another method* *getDefaultLocation* like this,

  this.getDefaultLocation = function () {

    return defaultLocation;

  };

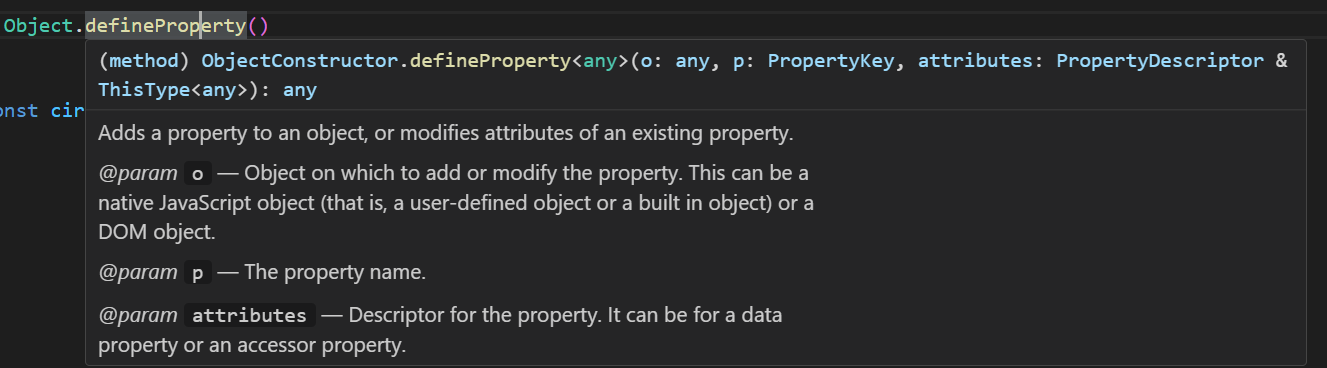
Then we can call this method from outside,

const circle = new Circle(1);

console.log(circle.getDefaultLocation()); //{ x: 1 }

But we need to be able *to access this information as a property* (*but not set it*)like circle.defaultLocation not by using a method.

Now in order to define a property like that we use Object.DefineProperty().



  Object.defineProperty(this, "defaultLocation", {

    get: function () {

      return defaultLocation;

    },

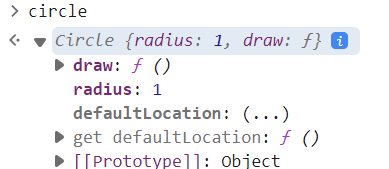
  });

First argument of this method is the object that you want to add a new property to. That object is the one which is referenced by “*this’* .

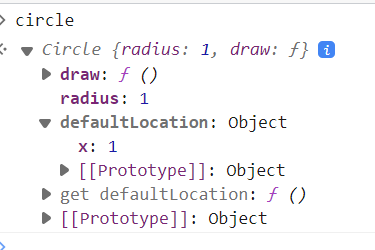
Second argument is the name of the property, we call it *defaultLocation*.

And third argument is an object and in this object we add a key value pair. The key is *get*, which is a special keyword for JavaScript and value is a function which will be called when *circle*.*defaultLocation* is executed and return *defaultLocation*.

Note: We do not see defaultLocation in intellisense, however we can see it in our browser console.



Notice *three dots* in front of defaultLocation because it is a computed property. When we click on it , *get* function will be executed and we get our defaultLocation.



So *getter is a function that is used to read a property*.

Now if you want to change the value of this property from the outside then we define a ***setter*** by adding another key – value pair in Object.defineProperty.

  Object.defineProperty(this, "defaultLocation", {

    get: function () {

      return defaultLocation;

    },

    set: function (value) {

      defaultLocation = value;

    },

  });

}

This *set* function takes an argument called *value* and we can set our defaultLocation to this value.

Note: Since we are using a function to set the value, in this we can perform some validations for our *value* before setting it.

    set: function (value) {

      if (value < 0) {

        throw new Error("Invalid location");

      } else defaultLocation = value;

    },

Note: Extensive discussion on why use getters and setters on stack overflow…

<https://stackoverflow.com/questions/1568091/why-use-getters-and-setters-accessors>

**My stopwatch implementation**:

function StopWatch() {

  let isStarted = false;

  let startTime, stopTime;

  this.duration = this.duration;

  this.start = function () {

    if (isStarted) {

      console.log("Stopwatch has already started");

    } else {

      isStarted = true;

      console.log("stopwatch has started");

      startTime = new Date();

    }

  };

  this.stop = function () {

    if (!isStarted) {

      console.log("stopwatch has already been stopped");

    } else {

      isStarted = false;

      console.log("stopwatch has stopped now");

      stopTime = new Date();

    }

  };

  Object.defineProperty(this, "duration", {

    get: function () {

      return (stopTime - startTime) / 1000;

    },

  });

}

const sw = new StopWatch();