Instance Methods

Example

```
class Hello
   {
      test()
      {
         console.log('test method');
      }
   }
   new Hello().test();
```

Example

```
class Person{
   name = "Yash"; // Public Variables or Instance Variable
   age = "Singh";
   constructor(name,age){
      this.name = name;
      this.age = age;
   }

   getFullName(){
      return `Hello ${this.fullName} your age is ${this.newAge}`
   }

}

let p = new Person("Rohit",27)

console.log(p)
console.log(p.getFullName())
```

Prototype Methods

- Prototype methods can be called by the instance of the class.
- A prototype method is a method that is only accessible when you create an instance of the class.
- Prototype methods also include getters and setters.
- All prototype methods have writable, non-enumerable, and configurable.

Example

```
class Person{
   name = "Yash"; // Public Property
   age = "Singh"; // Public Property
   constructor(name,age){
      this.fullName = name; // Instance Property
      this.newAge = age; // Instance Property
   }
}

Person.prototype.prntName = function(){ // Prototype Method
   return `Hello ${this.fullName} and ${this.name}`
}

let p = new Person("Rohit","Singh")


console.log(p.prntName()) // Hello Rohit & Yash
console.log(Person.prototype.prntName()) // Undefined
```

Things to remember about prototype methods

The arrow function does not have a prototype property, so we cannot use the arrow function in order to create prototype methods.

Example

```
class Person{
   name = "Yash"; // Public Property
   age = "Singh"; // Public Property
   constructor(name,age){
      this.fullName = name; // Instance Property
      this.newAge = age; // Instance Property
   }

}

Person.prototype.prntName = () => `Hello ${this.fullName}`

let p = new Person("Rohit","Singh")

console.log(p.prntName()) // Hello Undefined
console.log(Person.prototype.prntName()) // Hello Undefined
```

Private Methods

- These methods are only accessible within the class that instantiated the object.
- We can create private methods inside of class by using '#' followed by method name.
- They are only accessible only inside the class.

Example

```
class Person {
   #getName;
   constructor(name) {
      this.#getName = name;
   }
   prntName() {
      return this.#getName;
   }
   #prntMessage() {
      return "Hello This message is from private method";
   }
   prntPrivateMethod(){
      return this.#prntMessage();
   }
}
const p = new Person("Rohit Singh");
console.log(p.prntName())
console.log(p.prntPrivateMethod())
```

Protected Methods

Example:

```
class Person {
    _protectedMethod()
    {
        console.log("here is the protected method");
    }

}
new Person()._protectedMethod();
```

Example:

```
class Person {
    _protectedMethod()
    {
        console.log("here is the protected method");
    }

}
class Check
{
    callProtectedMethod()
    {
        console.log("calling protected method of Person class");
        new Person()._protectedMethod();
    }
}
new Check().callProtectedMethod();
```

Note : The declaration of protected properties and protected methods is just to tell the developers about the respective scope. But we can access the protected properties and method outside the class.

Static Methods

- The method which relates with class, directly.
- Use static keyword to define the static methods.

Example

```
class Person {
   static myFunction()
   {
      console.log('static method executed');
   }

}
Person.myFunction();
```

Example

```
class Person {
      name='nitin';
      static myFunction()
      {
         console.log('The name is ',this.name);
      }
  }
```

Note : Using this.name inside the static function returns the name of the class, because static function relates with their respective class.

Example

```
class Person {
    name='nitin';
    static myFunction(name)
    {
        console.log('The name is ',name);
    }

  }
  var p=new Person();
  name=p.name;
  Person.myFunction(name);
```

Or we can use,

```
class Person {
    name='nitin lawania';
    static myFunction()
    {
        var p=new Person();
        name=p.name;
        console.log('The name is ',name);
    }

  }

  Person.myFunction();
```

Inheritance

- Use to inherit or extends the functionality of a class.
- To inherit a class, we use 'extends' keyword.
- We can't inherit more than one class in a single class, so ES6 doesn't support the concept of multiple inheritance.
- Using inheritance, we implement the concept of code reusability.

Example

```
class Person {
      showMessage()
      {
         console.log('parent class message');
      }
   }
   class Child extends Person{
      showAnotherMessage()
      {
         console.log("child class message");
      }
   }
  var obj=new Child();
  obj.showAnotherMessage();
  obj.showMessage();
```

Example

```
class Person1 {          }
   class Person2    {
   }
   class Child extends Person1,Person2{
         }
```

Note : above concept (multiple inheritance) not allowed.

Example – When we have constructor in child and parent class

```
class Person{
    constructor()
    {
        console.log("parent class constructor")
    }
}

class Child extends Person{
    constructor()
    {
        super();
        console.log("child class constructor")
    }
}

new Child();
```

Note :

- call to super class/parent class constructor using 'super()' statement can be used only once inside the body of child constructor.
- It's not necessary that the super() statement always should be the first statement of the child class constructor.
    - You can use 'super()' statement anywhere inside the body of child class constructor depending upon the requirements.

Example

```
class Person{
    constructor()
    {
        console.log("parent class constructor")
    }
}

class Child extends Person{
    constructor()
    {

        console.log("child class constructor")
        super();
    }
}

new Child();
```

Polymorphism

- When an entity acts differently under different circumstances.
- In ES6, the concept of polymorphism is implements by using
    - Function overriding

Function Overriding

- When we have more than one function with same name inside both parent & child class then this is known as function overriding.
- The parameters of the function must be same in terms of number and identifiers.

Example

```
class Person{
    show(message)
    {
        console.log(message);
    }
}

class Child extends Person
{
    show(message)
    {
        console.log("The message is ",message);
    }
}

var obj=new Child();
obj.show('Hello students');
```

Example

```
class Person{

    show(message)
    {
        console.log(message);
    }
}

class Child extends Person
{
    show(message)
    {
        console.log("The message is ",message);
        super.show('nitin');
    }
}

var obj=new Child();
obj.show('Hello students');
```

Homework:

- How we can implement the concept of Abstraction?
- Can we create abstract class in ES6? If not, then what is the substitute & if yes, then how we can create it?
- Can we have interfaces concept in ES6? ? If not, then what is the substitute & if yes, then how we can create it?
- How we can implement the concept of Encapsulation in ES6?