Object-Oriented Programming (OOP)

Definition: OOP is a programming style that uses "objects" to represent real-world things. It helps organize code by grouping data and behaviors together, making it easier to manage and reuse.

Key Features:

- **Encapsulation:** Keeping data safe inside objects.
- **Abstraction:** Hiding unnecessary details and showing only the important parts.
- **Inheritance:** Allowing new objects to take properties and methods from existing ones.
- **Polymorphism:** Letting objects of different classes be treated as the same type.

Class

Definition: A class is a blueprint for creating objects. It defines properties (data) and methods (functions) that the objects will have.

Key Points:

- **Properties:** Variables that store data about the object.
- **Methods:** Functions that define what the object can do.
- **Instantiation:** Creating an object from a class.

- Attributes: in OOP are the properties or data stored inside an object.

Object: in OOP is an instance of a class. Objects are created from classes and hold the actual data.

Example:

javascript Copy code

```
class Car {
    constructor(brand, model) {
        this.brand = brand; // property
        this.model = model; // property
    }

    drive() { // method
        console.log(`Driving a ${this.brand} ${this.model}`);
    }
}

const myCar = new Car('Toyota', 'Camry'); // creating an object
myCar.drive(); // Outputs: Driving a Toyota Camry
```

- Instantiation: is the process of creating an object from a class.

1. Abstraction

Definition: Abstraction means showing only the important features of something while hiding the details. It helps to reduce complexity.

- **Abstract Class:** A class you can't create objects from; it's just a blueprint for other classes.
- **Abstract Method:** A method that doesn't have a body and must be defined in a subclass.
- **Concrete Class:** A class that is fully defined and can be used to create objects.
- **Interfaces:** A set of methods that a class must implement, like a contract.

Local Variables: Variables declared within a function are not part of the object. For example,

declaring a color variable in a function will not affect the object's properties.

Closure: A function that retains access to its lexical scope, even when the function is executed outside that scope.

Example of Closures:

javascript Copy code

```javascript
function outerFunction() {
    let x = 10; // local variable
    function innerFunction() {
        console.log(x); // accesses the outer function's variable
    }
    return innerFunction;
}
const inner = outerFunction();
inner(); // Outputs: 10
```

Private Members: You can use local variables and inner functions to create private members:

javascript Copy code

```javascript
function Circle(radius) {
    let defaultLocation; // private member
    function computeOptimumLocation() {
        // logic here
    }
    this.radius = radius; // public property
    this.draw = function() {
        computeOptimumLocation();
        // drawing logic
    };
}
```

Accessing Private Members:

javascript Copy code

```javascript
Circle.prototype.getDefaultLocation = function() {
    return defaultLocation; // access private member
};
```

Read-Only Properties and Setters for Validation:

javascript Copy code

```javascript
Object.defineProperty(this, 'defaultLocation', {
    get: function() {
        return defaultLocation; // getter
    },
    set: function(value) {
        if (!value.x || !value.y) {
            throw new Error('Invalid location');
        }
        defaultLocation = value; // set private member
    }
});
```

Access Modifiers:

- `readonly`
- `private`
- `protected`
- `public`

Keywords: `abstract`, `extends`, `implements`

---

2. **Encapsulation**

Definition: Encapsulation is like putting a protective shield around your data. It keeps everything safe inside a class and hides the details from the outside.

- Access Modifiers: Keywords that decide who can see or use a class's properties (`private` means no one outside can see it, `public` means everyone can).
- Getters and Setters: Special methods to read or change the values of private properties safely.
- Modules: Separate files that keep code organized and hidden from the outside.
- Private Members: Variables and methods that only the class can use.

Keywords: `private`, `protected`, `public`, `get`, `set`

---

3. **Inheritance**

Definition: Inheritance allows a new class to take properties and methods from an existing class. It helps to reuse code and create a family of classes.

- Parent Class: The class that provides properties and methods to another class.
- Child Class: The new class that gets features from the parent class.
- Method Overriding: Changing a method in the child class that already exists in the parent class.
- Interfaces Implementation: A child class agrees to follow a set of methods defined by an interface.

Keywords: `extends`, `super`, `implements`

---

4. **Polymorphism**

Definition: Polymorphism allows different classes to be treated as if they are the same type. It lets you use the same method name in different ways.

- Method Overriding: Changing how a method works in a child class compared to its parent class.
- Method Overloading: Having the same method name but different parameters in the same class (only in TypeScript).
- Interface-based Polymorphism: Different classes can use the same interface, making them

interchangeable.

- **Generics:** A way to make functions and classes work with any type of data, providing flexibility.

Keywords: `override`, `<T>` (for generics), `extends` (for constraints)

Technical Terminology

Implementation: Writing the actual code that makes a class's methods and properties work.

Concrete Class Definition: A concrete class is a type of class that is fully defined, meaning it has all its properties and methods implemented. You can create objects from a concrete class, which means you can use it in your programs.

- **Concrete Class:** Complete and can be used to create objects.
- **Abstract Class:** Incomplete, serves as a template, and cannot be instantiated directly.

Example:

javascript Copy code

```javascript
abstract class Animal {
    abstract makeSound(); // Abstract method (no implementation)
}

class Cat extends Animal {
    makeSound() {
        console.log("Meow");
    }
}
```

Summary

- Concrete Class: Complete and can be used to create objects.
- Abstract Class: Incomplete, serves as a template, and cannot be instantiated directly.

Primitives are copied by their value: **Number String Boolean Symbol undefined null**

Non-Primitive are copied by their reference: **Object Function Array**

```
let student1 = { name: "Suhaib", age: 20, isPresent: true };
let student2 = { name: "Suhaib2", age: 19, isPresent: false };
console.log(student1);
console.log(student2);
// If there are 100 students in a class, I have to make 100 objects for each student.
// Then I have to console each student every time, leading to 600 lines of code. 😱
// The solution for this is a factory function.
// ------------------FACTORY FUNCTION-----------------------//
function makeObj(name: string, age: number, isPresent: boolean) {
  return {
    name,
    age,
    isPresent,
    greet() {
      console.log(`Hi ${name}`);
    },
  };

  // Storing and calling back the function
  let object1 = makeObj("Fahim", 20, true);
  console.log("object1: ", object1);
  object1.greet();
  let object2 = makeObj("Nofil", 27, true);
  console.log("object2: ", object2);
  object2.greet();
  // But it is making copies instead of sharing greet.
  console.log("But it is making copies instead of sharing greet:");
  console.log(`Here:`, object1.greet === object2.greet); //False

  // ------------------Constructor Function Requirements-----------------------//
  // Capitalize the first letter
  // The `this` keyword
  // Filling this keyword at least once will turn it green; hover to see it's a constructor function.
  // Using the `new` keyword while calling the function.

  // ------------------PLAIN FUNCTION-----------------------//

  function MakeObj(name, age, isPresent) {}

  let Student1 = MakeObj("Fahim", 20, true);
  console.log("Student1: ", Student1); //! undefined > Because we didn't use the `new` keyword.

  function MakeObj(name, age, isPresent) {
```

.js 9+ ×

.js > makeObj

```
function makeObj(name: string, age: number, isPresent: boolean) {
  function MakeObj(name, age, isPresent) {
    console.log(this);
  } //? We use `this` without `new`, and the answer is the same.

  let Student1 = MakeObj("Fahim", 20, true);
  console.log("Student1: ", Student1); //! undefined > Because we didn’t use the `new` keyword.

  // -----------------PLAIN FUNCTION with new and this
  // This keyword returns an empty object-----------------------//
  function MakeObj(name, age, isPresent) {
    console.log(this);
  } // MakeObj {}

  let Student1 = new MakeObj("Fahim", 20, true);
  console.log("Student1: ", Student1); // Student1: MakeObj {}

  // ------ Unreachable code detected. ts(7027)  ect here
  console.lo No quick fixes available
    `Fill an object in JavaScript! Only works in JavaScript, not TypeScript.`
  ); //! Only works in JavaScript, not TypeScript
  let Obj = {};
  console.log(`When empty`, Obj);
  Obj.name = "Suhaib";
  console.log(`When filled`, Obj);

  // -----------------PLAIN FUNCTION to a
  // Constructor Function-----------------------//
  function MakeObj3(name, age, isPresent) {
    console.log(this);

    this.name = name;
    this.age = age;
    this.isPresent = isPresent;
  }

  let Student2 = new MakeObj3(
    "Finally, it is a constructor function.",
    1,
    true
  );
  console.log("Student2: ", Student2);
  console.log("Because it isn't coming from class interface/type aliases.");

  // ------------Add FUNCTION in a Constructor Object (called method)
```

```
function makeObj(name: string, age: number, isPresent: boolean) {
  function MakeObjWithMethod(name, age, isPresent) {
    console.log(this);

    this.name = name;
    this.age = age;
    this.isPresent = isPresent;
    this.greet = function () {
      console.log(`Hello ${this.name}`);
    };
  }

  let object1 = new MakeObjWithMethod("Fahim", 20, true);
  console.log("object1: ", object1);
  object1.greet();

  let object2 = new MakeObjWithMethod("Nofil", 27, true);
  console.log("object2: ", object2);
  object2.greet();

  // But it is still making copies instead of sharing greet.
  console.log("But it is still making copies instead of sharing
  greet:");
  console.log(`Here:`, object1.greet === object2.greet);

  /*If I have to make objects for all students and work until Tuesday 9
  at 12, I would have to make 100 objects, leading to 400 lines of
  code. This function allows me to create objects efficiently, and I
  keep passing their values in parameters, which is called a factory
  function. Now we can also create a function that retains an object.
  This function is called a method—not a key-value pair. The topic we
  make methods separately but inside the function allows every student
  to be a unique copy. Please, we don't create 100 copies, but it
  consumes memory. */
  // To create a constructor, start with a capital letter. Use the `new`
  keyword before calling it; otherwise, it will act like a normal
  function and won't return the expected results.
//Consider this: I can create an object, and we have to pass values in
it. A function like this:
function MakeFunction(name, age) {
  // Nothing in the body means nothing in the output.
  console.log(this); // and undefined (when you call MakeObject {})
  return this; // nothing in this cases.
}
let std1 = new MakeFunction("Fahim", 20);
console.log(std1); // MakeObj {}
```

Now, if I make a function inside this, it will still make copies, and nothing will change. So what should we do?

We use prototypes; every object comes with one prototype, which shares one copy with all. Whenever we call any property using the object or function, it checks if it is present; if not, it searches in the prototype. For instance, when I do `sweet` or press Control + Space, it gives a string from the prototype type.

When I log my object/function/array in the browser, they are visible in the directory structure but not in the terminal of VS Code. So how to create a prototype?

1. `MakeObj.__proto__.key = "value";`
2. `MakeObj.prototype = { ... };`

`this` returns the whole function, but in the case of a constructor function, it returns an empty object, and then we add keys and values in it.

In JavaScript, typing is minimal, but TypeScript requires more, so OOP helps here. We will use an interface to define or structure our code, using classes to define structures. The constructor is automatically created for a class, but we can see that it is on the backend, returning "this".

Now we cover factory functions. It creates objects in bulk. A function is returning an object since the object can contain other functions called methods. Both are key-value pairs or outputs. This approach addresses memory issues, so we move to constructor functions with `new` and `this`.

`this` is used to push values into an empty object, but the issue remains unresolved with `.prototype` vs. `.__proto__`.

Constructor functions in TypeScript can be complex (too much code), so we move to OOP in TypeScript, where we assume class behavior like an interface/type alias.

If we use the `new` keyword, we create an instance of that class and get all properties of that class.

Whenever we use inheritance from a class, we use the `extends` keyword. If a class inherits properties from another class, the `super()` call is at the top of the constructor function, passing required arguments. If the parent class lacks a constructor method, we simply write an empty `super()`.

`super`: A way to call inherited properties.

An abstract class can inherit other classes and allow child classes to inherit while restricting direct instantiation.