**Assignment 1**

1. **Instance** 🡪 **form in JS & TS**

In JavaScript and TypeScript, the concept of an "instance" generally refers to an object created from a class or a function constructor. This concept is foundational in object-oriented programming (OOP), where classes define the blueprint of an object and instances are the actual objects created based on that blueprint.

1. **JS:**

* **Function Constructors:**

Before ES6, function constructors were a common way to create objects. A function constructor is simply a regular function that is used with the new keyword to create an object instance.

* **ES6 Classes:**

With the introduction of ES6, JavaScript added a class syntax, which is syntactic sugar over the existing prototype-based inheritance. This syntax makes the creation of objects and the definition of inheritance more intuitive.

1. **TS:**

TypeScript, being a superset of JavaScript, uses the same basic principles for creating instances but adds static typing. This enhances code quality and maintainability by providing type checks at compile time.

* **Classes with Type Annotations:**

In TypeScript, you can define the types of properties and methods in a class, which helps in catching type-related errors during development.

* **Interfaces and Inheritance:**

TypeScript also supports interfaces, which can be used to define the shape of an object, and classes can implement these interfaces. This allows for strong typing and better code organization.

An "instance" refers to a specific realization of a class. In JavaScript and TypeScript, forms (often referring to HTML forms) are used to collect user input. The interaction between class instances and forms typically involves using instances to manage form data or behavior.

1. **Forms** 🡪 **in JS**

In JavaScript, "forms" typically refer to HTML forms, which are a fundamental part of web applications for collecting user input. JavaScript interacts with forms to handle data validation, submission, and user interaction. Here's an overview of how forms work in JavaScript:

super and this

* **Basic Form Structure in HTML:**

An HTML form is defined using the <form> element, which can contain various input elements such as text fields, checkboxes, radio buttons, and more. The <form> tag often includes attributes like action (the URL to send the form data to) and method (the HTTP method to use, such as GET or POST).

<form id="myForm" action="/submit" method="POST"></form>

* **Handling Form Submission with JavaScript:**

JavaScript can be used to handle form submission, validate input, and control the behavior of the form without reloading the page. This is often done using event listeners.

document

  .getElementById("myForm")

  .addEventListener("submit", function (event) {});

* Form Data Collection and AJAX Submission:

JavaScript can also be used to collect form data and submit it asynchronously using AJAX (Asynchronous JavaScript and XML). This allows the page to send data to the server and update the page without reloading.

document.getElementById('ajaxForm').addEventListener('submit', function (event) {

      event.preventDefault(); // Prevent the default form submission

      const formData = new FormData(this);

      fetch('/submit', {

        method: 'POST',

        body: formData

      })

        .then(response => response.json())

        .then(data => {

          document.getElementById('response').textContent = data.message;

        })

        .catch(error => console.error('Error:', error));

    });

1. **Super and This**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **super** | **this** |
| **Definition** | Refers to the parent class (superclass). | Refers to the current instance of the class. |
| **Usage** | Used to call the constructor or methods of the parent class. | Used to access or set properties and methods of the current instance. |
| **Location** | Must be used inside constructors and methods of a subclass. | Used in constructors, methods, and elsewhere in class methods. |
| **Constructor Call** | Must be called before using this in a subclass constructor. | Initializes or refers to the properties of the class instance. |
| **Method Call** | Can call a parent class method that is overridden in the subclass. | Refers to the current instance's method or property, even if overridden. |
| **Context** | Refers to the parent class scope. | Refers to the instance of the class where it's used. |
| **Initialization** | Helps initialize the parent class. | Used to initialize the instance's properties. |

**What is Best About Inheritance (super or this)?**

Both super and this play crucial roles in class-based inheritance, but they serve different purposes and are both important in their own ways:

* **super** is best for:
* **Code Reusability**: By calling parent class constructors and methods, it facilitates the reuse of existing functionality in the parent class, reducing duplication.
* **Polymorphism**: It allows subclasses to override methods but still access the original method implementation from the parent class, which is essential for polymorphism.
* **Encapsulation and Initialization**: Ensures that the base class is properly initialized, maintaining encapsulation and proper setup of class hierarchies.
* **this** is best for:
* **Instance Management**: Directly manages and accesses the properties and methods of the current instance, making it essential for the internal state and behavior of objects.
* **Dynamic Context**: Provides a way to refer to the instance dynamically, which is crucial in method definitions and when working with callbacks or event handlers.

super is vital in inheritance for extending parent class functionality and ensuring proper initialization and method access. It supports polymorphism and encapsulation in subclasses. this is essential for managing the current instance of a class.

1. **Type of** 🡪 **Inheritance**
2. **Single Inheritance**

Single inheritance means that a class inherits from one superclass. This is the most common form of inheritance and is directly supported in both JavaScript and TypeScript.

class Animal {}

class Dog extends Animal {}

1. **Multilevel Inheritance (Everyone inherits from one tribe to another)**

Multilevel inheritance occurs when a class is derived from another derived class, forming a chain of inheritance.

class Animal {}

class Mammal extends Animal {}

class Dog extends Mammal {}

1. **Hierarchical Inheritance (Each one inherits from the parent)**

Hierarchical inheritance occurs when multiple classes inherit from a single superclass. This type of inheritance is useful for creating a common base class with shared properties and methods**.**

class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}

1. **Multiple Inheritance (Not Directly Supported)**

Multiple inheritance, where a class inherits from more than one superclass, is not directly supported in JavaScript or TypeScript due to potential complexity and ambiguity (often referred to as the "diamond problem").

**However, JavaScript and TypeScript can achieve similar effects through mixins or interfaces.**

**In JS:**

const CanRun = (Base) =>

  class extends Base {};

class Animal {}

class Dog extends CanRun(Animal) {}

**In TS:**

interface CanRun {}

interface CanBark {}

class Dog implements CanRun, CanBark {}

1. **constructor types**
2. Default Constructor: Automatically provided if no constructor is defined.

class Person {

  // No constructor defined, so a default constructor is provided

}

1. Parameterized Constructor: Accepts parameters to initialize an object's properties.

class Person {

  name: string;

  age: number;

  constructor(name: string, age: number) {

    this.name = name;

    this.age = age;

  }

}

1. Copy Constructor: Emulated in JavaScript/TypeScript to create a new object from an existing one.

class Person {

  name: string;

  age: number;

  constructor(name: string, age: number) {

    this.name = name;

    this.age = age;

  }

  // Copy constructor

  static copy(person: Person): Person {

    return new Person(person.name, person.age);

  }

}

const original = new Person("Alice", 30);

const copy = Person.copy(original);

1. Private Constructor: Prevents instantiation of a class from outside (supported in TypeScript).

class Singleton {

  private static instance: Singleton;

  private constructor() {

    // Private constructor ensures the class can't be instantiated from outside

  }

  static getInstance(): Singleton {

    if (!Singleton.instance) {

      Singleton.instance = new Singleton();

    }

    return Singleton.instance;

  }

}

1. Constructor Overloading: TypeScript allows multiple constructor signatures, but all must be implemented within one actual constructor function.

class Person {

  name: string;

  age: number;

  constructor(name: string);

  constructor(name: string, age: number);

  constructor(name: string, age?: number) {

    this.name = name;

    this.age = age !== undefined ? age : 0;

  }

}

const person1 = new Person("Alice");

const person2 = new Person("Bob", 25);

1. Nested Class (also known as an inner class):

Nested class is defined within another class. This can be useful for logically grouping classes that are only used in one place, enhancing encapsulation, and reducing namespace pollution. JavaScript does not have built-in support for nested classes in the same way as some other languages, but you can achieve similar results using class expressions or closures. TypeScript, being a superset of JavaScript, follows the same pattern but adds type checking.

1. In JS

* Nested Class using Class Expression:

class OuterClass {

  constructor() {}

  InnerClass = class {

    constructor() {}

    innerMethod() {}

  };

  outerMethod() {}

}

* Nested Class inside a Method:

class OuterClass {

  createInnerClassInstance() {

    class InnerClass {

      constructor() {}

      innerMethod() {}

    }

    return new InnerClass();

  }

  outerMethod() {}

}

1. In TS

* Nested Class as a Static Property:

class OuterClass {

  name: string;

  constructor(name: string) {

    this.name = name;

  }

  greet(): void {}

  // Define InnerClass as a static property

  static InnerClass = class {

    innerName: string;

    constructor(innerName: string) {

      this.innerName = innerName;

    }

    innerGreet(): void {}

  };

}

* Nested Class within a Method:

class OuterClass {

  createInnerClassInstance(innerName: string) {

    class InnerClass {

      innerName: string;

      constructor(innerName: string) {

        this.innerName = innerName;

      }

      innerGreet(): void {}

    }

    return new InnerClass(innerName);

  }

}