**Class Type Annotations**

In TypeScript, class type annotations are used to specify the types of properties, methods, and constructor parameters within a class. They define the expected data types and enable TypeScript to perform type checking to ensure that class instances adhere to the defined structure. Class type annotations help catch type-related errors at compile time, improving code reliability. Here's an explanation of class type annotations:

class Product {

// Property type annotations

id: number;

name: string;

price: number;

constructor(id: number, name: string, price: number) {

// Constructor parameter type annotations

this.id = id;

this.name = name;

this.price = price;

}

// Method type annotations

getProductInfo(): string {

return `ID: ${this.id}, Name: ${this.name}, Price: $${this.price}`;

}

}

// Create an instance of the Product class

const product1 = new Product(1, "Widget", 20.0);

// Access class properties and call a method

console.log(product1.getProductInfo()); // Output: ID: 1, Name: Widget, Price: $20

In this example:

* Class Product defines properties id, name, and price, each with type annotations specifying their expected data types.
* The constructor of the Product class takes three parameters (id, name, and price), and their types are annotated.
* The getProductInfo method is defined with a return type annotation (string) to specify the expected return type.
* An instance product1 of the Product class is created and initialized with data.
* The getProductInfo method is called on product1, and the result is logged to the console.

**class access modifiers:**

In TypeScript, class access modifiers are keywords used to control the visibility and accessibility of class members (properties and methods) from outside the class. There are three main access modifiers:

1. public: Members declared as public are accessible from anywhere, both within and outside the class. This is the default modifier if you don't explicitly specify one.

Example:

class MyClassPublic {

public name: string;

constructor(name: string) {

this.name = name;

}

}

const instancePublic = new MyClassPublic("John");

console.log(instancePublic.name); // Accessing the public property

1. private: Members declared as private are only accessible from within the class they are defined in. You cannot access private members from outside the class.

Example:

class MyClassPrivate {

private secret: string;

constructor(secret: string) {

this.secret = secret;

}

revealSecret() {

console.log(this.secret); // Accessing the private property from within the class

}

}

const instancePrivate = new MyClassPrivate("My secret");

// console.log(instance.secret); // This would result in an error because secret is private

instancePrivate.revealSecret(); // This is a valid way to access the private property

1. protected: Members declared as protected are accessible from within the class they are defined in and from subclasses (derived classes). You cannot access protected members from outside the class or unrelated classes.

Example:

class Parent {

protected familyName: string;

constructor(name: string) {

this.familyName = name;

}

}

class Child extends Parent {

introduceFamily() {

console.log(`Our family name is ${this.familyName}`);

}

}

const parent = new Parent("Smith");

const child = new Child("Johnson");

// console.log(parent.familyName); // This would result in an error because familyName is protected

child.introduceFamily(); // This is a valid way to access the protected property

These access modifiers help you encapsulate the internal details of a class and provide a level of control over how class members are accessed and manipulated. They are essential for maintaining the integrity and security of your code in larger applications.

**Class accessors**

Class accessors, also known as getters and setters, are a way to control access to the properties of a class. They allow you to get and set the values of class properties while providing additional control and logic if needed. Accessors are defined using the **get** and **set** keywords.

class ProductExAccessors {

// 'private' property

private \_price: number;

constructor(private id: number, private name: string) {

this.\_price = 0; // Default price

}

// 'public' getter for price

get price(): number {

return this.\_price;

}

// 'public' setter for price

set price(newPrice: number) {

if (newPrice >= 0) {

this.\_price = newPrice;

} else {

console.log("Price cannot be negative.");

}

}

getProductInfo(): string {

return `ID: ${this.id}, Name: ${this.name}, Price: $${this.\_price}`;

}

}

// Create an instance of the ProductExAccessors class

const productEx = new ProductExAccessors(1, "Widget");

console.log(productEx.getProductInfo()); // Default price: ID: 1, Name: Widget, Price: $0

// Use the 'setter' to update the price

productEx.price = 20.0;

console.log(productEx.getProductInfo()); // Updated price: ID: 1, Name: Widget, Price: $20

// Attempting to set a negative price triggers the setter logic

productEx.price = -5; // Price cannot be negative.

In this example:

* We've added a private property \_price to the ProductExAccessors class, which stores the product's price.
* Accessors (get and set) for the price property provide controlled access to \_price. The get method retrieves the price, and the set method allows setting it while enforcing a non-negative constraint.
* When the price property is set to a negative value, a message is logged to indicate that it's not allowed.
* The getProductInfo method still uses \_price to display the product's price.
* We demonstrate using the setter to update the price and handle a case where an invalid (negative) price is attempted to be set.

**class static members**

Static members in a class are properties or methods that belong to the class itself, rather than to instances of the class. This means you can access them without creating an instance of the class. They are defined using the **static** keyword in TypeScript. In this case, the static member nextId is used to manage unique IDs for all instances of the ProductStaticMembers class.

class ProductStaticMembers {

// 'private' property

private static nextId: number = 1;

constructor(private id: number, private name: string) {}

static generateNextId(): number {

return ProductStaticMembers.nextId++;

}

getProductInfo(): string {

return `ID: ${this.id}, Name: ${this.name}`;

}

}

// Generate unique IDs for products using the static method

const product1static = new ProductStaticMembers(ProductStaticMembers.generateNextId(), "Widget");

const product2static = new ProductStaticMembers(ProductStaticMembers.generateNextId(), "Gadget");

console.log(product1static.getProductInfo()); // ID: 1, Name: Widget

console.log(product2static.getProductInfo()); // ID: 2, Name: Gadget

In this example:

* We've added a private static property nextId to the Product class. This property keeps track of the next available ID for products.
* The generateNextId static method allows us to obtain the next ID in a controlled manner and increments nextId for the next call.
* When creating instances of the ProductStaticMembers class, we use the static method ProductStaticMembers.generateNextId() to generate unique IDs for each product.
* Each product instance retains its unique ID and other properties.

**class implement interface**

the class ensures that it provides the properties and methods required by that interface. It helps enforce a consistent structure for objects created from that class.

class ProductImplementInterface {

// 'private' property

private static nextId: number = 1;

constructor(private id: number, private name: string) {}

static generateNextId(): number {

return ProductImplementInterface.nextId++;

}

getProductInfo(): string {

return `ID: ${this.id}, Name: ${this.name}`;

}

}

// Generate unique IDs for products using the static method

const product1implement = new ProductImplementInterface(ProductImplementInterface.generateNextId(), "Widget");

const product2implement = new ProductImplementInterface(ProductImplementInterface.generateNextId(), "Gadget");

console.log(product1implement.getProductInfo()); // ID: 1, Name: Widget

console.log(product2implement.getProductInfo()); // ID: 2, Name: Gadget

In this example:

* We have an interface ProductImplementInterface that defines the structure of a product with properties id and name, as well as a method getProductInfo.
* The Product class implements the ProductImplementInterface interface using the implements keyword. It provides the required properties (id and name) and the method getProductInfo.
* Instances of the Product class, product1 and product2, are created and assigned to variables of type ProductImplementInterface . This demonstrates that the class conforms to the interface.

**abstract classes and members**

Abstract classes are used as blueprints for other classes. They cannot be instantiated on their own but can be subclassed by other classes. Abstract classes can also contain abstract methods, which are declared but not implemented in the abstract class itself. Subclasses are required to provide implementations for these abstract methods.

abstract class AbstractItem {

private static nextId: number = 1;

constructor(public id: number, protected name: string) {}

static generateNextId(): number {

return AbstractItem.nextId++;

}

abstract getItemInfo(): string;

}

class Item extends AbstractItem {

constructor(id: number, name: string) {

super(id, name);

}

getItemInfo(): string {

return `ID: ${this.id}, Name: ${this.name}`;

}

}

const item1: AbstractItem = new Item(AbstractItem.generateNextId(), "Widget");

const item2: AbstractItem = new Item(AbstractItem.generateNextId(), "Gadget");

console.log(item1.getItemInfo()); // ID: 1, Name: Widget

console.log(item2.getItemInfo()); // ID: 2, Name: Gadget

Here's a shorter explanation of the code:

1. AbstractItem is an abstract class serving as a blueprint for items. It includes:
   * A private static property, nextId, for generating unique IDs.
   * A constructor initializing id and name.
   * A static method, generateNextId(), for obtaining the next ID.
   * An abstract method, getItemInfo(), that must be implemented by subclasses.
2. Item is a concrete class extending AbstractItem. It provides a concrete implementation of getItemInfo().
3. Two Item instances, item1 and item2, are created with unique IDs and names.
4. The getItemInfo() method of each item is called, displaying their information.

**Polymorphism & Method Override**

This code example demonstrates the concept of polymorphism and method override in TypeScript, which is commonly used in object-oriented programming. It defines abstract classes and concrete subclasses to illustrate these concepts.

// Define an abstract class

abstract class AbstractEntity {

private static nextId: number = 1;

protected constructor(public id: number, protected name: string) {} // Change 'private' to 'protected'

static generateNextId(): number {

return AbstractEntity.nextId++;

}

abstract getEntityInfo(): string;

}

// Create a concrete subclass

class Entity extends AbstractEntity {

constructor(id: number, name: string) {

super(id, name);

}

getEntityInfo(): string {

return `ID: ${this.id}, Name: ${this.name}`;

}

}

// Create another concrete subclass

class AnotherEntity extends AbstractEntity {

constructor(id: number, name: string) {

super(id, name);

}

getEntityInfo(): string {

return `ID: ${this.id}, Name: ${this.name}, Additional Info: ...`;

}

}

// Generate unique IDs for entities using the static method

const entity1: AbstractEntity = new Entity(AbstractEntity.generateNextId(), "Widget");

const entity2: AbstractEntity = new AnotherEntity(AbstractEntity.generateNextId(), "Gadget");

// Polymorphism: Call 'getEntityInfo' method on different concrete subclasses

console.log(entity1.getEntityInfo()); // ID: 1, Name: Widget

console.log(entity2.getEntityInfo()); // ID: 2, Name: Gadget, Additional Info: ...

In this example:

1. **Abstract Class (AbstractEntity)**:
   * Defines an abstract class named **AbstractEntity**, which serves as a blueprint for other classes.
   * Features a private static property **nextId** to generate unique entity IDs.
   * The constructor now uses **protected** instead of **private** for the **name** property to allow access in derived classes.
2. **Concrete Subclasses (Entity and AnotherEntity)**:
   * Two concrete subclasses extend **AbstractEntity**: **Entity** and **AnotherEntity**.
   * Each class has its constructor, passing **id** and **name** to the parent class's constructor via **super(id, name)**.
   * They override the **getEntityInfo** method to provide custom entity information.
3. **Usage of Subclasses**:
   * Instances of **Entity** and **AnotherEntity\*\* are created and stored in variables of the abstract class type (\*\*AbstractEntity`**).
   * The **generateNextId** method from the abstract class generates unique IDs.
   * Polymorphism is demonstrated as the overridden **getEntityInfo** methods are called on these instances, even though they have an abstract class type.
4. **Console Output**:
   * The **getEntityInfo** method is called on both instances, and the results are logged, showing different information based on the subclass's implementation.

This code illustrates polymorphism and method overriding in TypeScript. It enables the creation of various entity types while adhering to a common interface provided by the abstract class, showcasing the power of object-oriented programming and inheritance.