Classes & Access Modifiers

Problem Statement

Case Study: Digital Bank Account Management

A new digital bank is building a system to manage customer accounts:

- Customers should be able to view their balance and make deposits or withdrawals.
- Only the bank's internal systems (not customers or external code) should be able to update sensitive information like account numbers or transaction logs.
- Fraud detection and auditing teams need to access certain account details, but should not be able to directly change balances or sensitive fields.
- The system must ensure that no one can accidentally or maliciously tamper with protected or private data.

The challenge:
How do you design a class-based banking system that gives customers, auditors, and internal systems the right level of access—no more, no less—while keeping sensitive data secure and code maintainable?

---

2. Learning Objectives

By the end of this tutorial, you will:

- Understand how classes and access modifiers (`public` , `private` , `protected` , `readonly`) control data visibility and mutability.
- Model real-world banking entities using classes.
- Enforce encapsulation and security in your code.
- See how access modifiers protect against both accidental and intentional misuse.

Think of your code as an office building:

3. Concept Introduction with Analogy

Area/Room	Access Modifier	Who Can Enter?	What Happens Here?
Lobby & Hallways	`public`	Everyone (employees, managers, directors)	Viewing shared info, casual meetings
Manager's Office	`private`	Only the manager (the class itself)	Working on sensitive projects
Shared Meeting Room	`protected`	Managers and directors (class & subclasses)	Team planning, confidential reviews
Building Blueprint	Abstract Class	Blueprint for all rooms	Sets required features for all rooms

Sets required features for all rooms

- **public:** Open to all, like the lobby.
- **private:** Only for the room owner, like a locked office.
- **protected:** For managers and directors, like a meeting room with restricted access.
- **abstract class:** The blueprint for all rooms; you can't walk into a blueprint, but every room follows its design.

What Are Classes in TypeScript?

A **class** in TypeScript is a blueprint for creating objects with specific properties (fields) and methods (functions).
Classes support **object-oriented programming** concepts like encapsulation, inheritance, and abstraction

Access Modifiers: Controlling Visibility

Access modifiers determine **where class members (properties and methods) can be accessed**.

Public

- **Default** modifier.
- Members are accessible from anywhere (inside/outside the class)

```
class Person {
public name: string;
constructor(name: string) {
  this.name = name;
}
}
const p = new Person("Alice");
console.log(p.name); // OK
```

Private

- Accessible **only within the class** where declared.
- Not accessible in subclasses or from outside.

```
class Secret {
 private code: string;
 constructor(code: string) {
   this.code = code;
 }
 reveal() {
   return this.code;
 }
}
const s = new Secret("1234");
// console.log(s.code); // Error: Property 'code' is private
console.log(s.reveal()); // OK
```

** Why Not Just Use Public Everywhere?**

- If all fields were public, anyone could:
    - Set their own balance.
    - Erase transaction logs.
    - Change account numbers.
- Using access modifiers ensures only the right code can access or modify sensitive data, preventing both bugs and fraud.

Protected

- Accessible **within the class and its subclasses** (not outside)

```
class Animal {
protected species: string;
constructor(species: string) {
  this.species = species;
}
}
class Dog extends Animal {
bark() {
  return `Woof! I am a ${this.species}`;
}
}
const d = new Dog("canine");
console.log(d.bark()); // OK
// console.log(d.species); // Error: Property 'species' is protected
```

Read-Only

- Not an access modifier, but ensures the value can't be changed after initialization.

```
class Book {
readonly isbn: string;
constructor(isbn: string) {
  this.isbn = isbn;
}
}
const b = new Book("123-456");
// b.isbn = "789-101"; // Error: Cannot assign to 'isbn' because it is a read-only property
```

Inheritance: Extending Classes

Inheritance lets you create a new class (child/subclass) that **inherits** properties and methods from another class (parent/superclass)

```
class Vehicle {
public brand: string;
constructor(brand: string) {
  this.brand = brand;
}
drive() {
  console.log(`${this.brand} is moving`);
```

```
}
}
class Car extends Vehicle {
public model: string;
constructor(brand: string, model: string) {
  super(brand); // calls Vehicle's constructor
  this.model = model;
}
drive() {
  super.drive(); // call parent method
  console.log(`Model: ${this.model}`);
}
}
const myCar = new Car("Toyota", "Corolla");
myCar.drive();
// Output: Toyota is moving
//         Model: Corolla
```

The `super` keyword calls the parent class's constructor or methods.

**Abstract Classes: Defining Contracts

An **abstract class** is a class that **cannot be incorporated directly**. It's used as a base for other classes and can include:

- **Abstract methods**: Declared without implementation-must be implemented by subclasses.
- **Concrete methods**: Fully implemented in the abstract class.

Combining Access Modifiers with Constructors

```
class Person {
 constructor(public name: string, private age: number) {}
 public getAge(): number {
   return this.age;
 }
}
const john = new Person('John', 30);
console.log(john.name); // OK
console.log(john.getAge()); // OK
// console.log(john.age); // Error: 'age' is private
```

TypeScript allows you to declare and initialize properties directly in the constructor using access modifiers

4. Step-by-Step Data Modeling and Code Walkthrough

Let's start by designing the **Content blueprint** that every content type must follow.

Let's see how Digital Bank uses classes and access modifiers to secure their system:

A. Define the BankAccount Class

- `public` for ownerName, getBalance, deposit, withdraw.
- `private` for balance, transactionLog, addTransaction.
- `protected` for fraudFlags, flagFraud, getTransactionLog.
- `readonly` for accountNumber.

B. Modeling the Bank Account: Protecting Core Data

Case Study Context:
A bank account's number and transaction log are highly sensitive. Only the system should update them; customers and even auditors should not be able to change these fields.

Implementation:

```
class BankAccount {
  public readonly accountNumber: string;   // Exposed to all, set once
  public ownerName: string;                // Can be seen/updated by customer
  private balance: number;                 // Only modifiable inside the class
  private transactionLog: string[] = [];   // Only the class can update/read

  constructor(accountNumber: string, ownerName: string, initialBalance: number) {
    this.accountNumber = accountNumber;
    this.ownerName = ownerName;
    this.balance = initialBalance;
    this.addTransaction(`Account opened with $${initialBalance}`);
  }

  // Public method for customers to check their balance
  public getBalance(): number {
    return this.balance;
  }

  // Public method for deposits, but balance is still private
  public deposit(amount: number): void {
    if (amount <= 0) throw new Error("Deposit must be positive");
    this.balance += amount;
    this.addTransaction(`Deposited $${amount}`);
  }

  // Public method for withdrawals, with internal validation
  public withdraw(amount: number): void {
    if (amount <= 0) throw new Error("Withdrawal must be positive");
    if (amount > this.balance) throw new Error("Insufficient funds");
    this.balance -= amount;
    this.addTransaction(`Withdrew $${amount}`);
  }

  // Private method: Only the class can add transactions
  private addTransaction(description: string): void {
    this.transactionLog.push(`${new Date().toISOString()}: ${description}`);
  }
}
```

Why?

- `public readonly accountNumber` : Customers and staff can see the account number, but it can't be changed after creation, protecting against identity fraud
- `private balance` and `private transactionLog` : Only the class can modify or read these. Even subclasses (like auditors or managers) cannot change the balance directly.
- All changes to balance must go through public methods with validation, enforcing business rules, and audit trails.

C. Supporting Specialized Roles: Auditors with Protected Access

Case Study Context:
Auditors need to review transaction logs and fraud flags, but should never be able to change balances or logs.

Implementation:

```
class BankAccount {
 // ...previous code...
 protected fraudFlags: string[] = [];     // Accessible to subclasses (e.g., auditors)
 protected getTransactionLog(): string[] {
   return [...this.transactionLog];       // Expose a copy, not the original
 }

 protected flagFraud(reason: string): void {
   this.fraudFlags.push(reason);
   this.addTransaction(`Fraud flag: ${reason}`);
 }
}

class Auditor extends BankAccount {
 public reviewAccount(): { flags: string[], log: string[] } {
   // Can access protected fraudFlags and getTransactionLog()
   return {
     flags: this.fraudFlags,
     log: this.getTransactionLog(),
   };
 }
}
```

Why?

- `protected` allows subclasses (like `Auditor`) to access fraud flags and logs, but not external code
- The auditor can review but not tamper with core data, supporting the separation of duties and regulatory compliance.

D. Preventing Unauthorized Changes: Private and Readonly

Case Study Context:
No customer or external code should be able to set their own balance, erase logs, or change account numbers.

Implementation:

```
const alice = new BankAccount("111222", "Alice", 1000);
alice.deposit(200);            // OK: public method
alice.withdraw(50);            // OK: public method
console.log(alice.getBalance()); // OK: public method

// alice.balance = 999999;      // Error: private
// alice.transactionLog = [];   // Error: private
// alice.accountNumber = "000000"; // Error: readonly
```

Why?

- `private` and `readonly` enforce strict encapsulation and immutability, preventing accidental or malicious tampering.

E. Manager Role: Subclass with Additional Powers

Case Study Context:
A bank manager can flag accounts for fraud but cannot change balances directly.

Implementation:

```
class BankManager extends BankAccount {
public flagAccount(reason: string) {
  this.flagFraud(reason); // OK: protected method
}
}

const manager = new BankManager("222333", "Charlie", 5000);
```

```
manager.flagAccount("Unusual withdrawal pattern");
// manager.balance = 0; // Error: private
```

Why?

- `protected` lets managers use internal fraud-flagging logic, but not access or change private fields directly

E. Summary Table: Who Can Access What?

Role	Can View Balance	Can Change Balance	Can See Logs	Can Change Logs	Can Flag Fraud
Customer	Yes (`public`)	Yes (`public`)	No	No	No
Auditor (subclass)	Yes (`public`)	No	Yes (`protected`)	No	No
Manager (subclass)	Yes (`public`)	No	Yes (`protected`)	No	Yes (`protected`)
External code	Yes (`public`)	No	No	No	No

6. Challenge

Your Turn!

- Add a `BankManager` subclass that can flag accounts for review (call `flagFraud`), but cannot change the balance directly.
- Try to access private/protected fields from outside the class—see what errors you get.

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Using `public` for everything	Use `private` / `protected` to encapsulate
Changing `readonly` properties	Only set in constructor
Accessing private fields directly	Use `public` / `protected` methods for access
Not using inheritance	Use `extends` for specialized roles

8. Optional: Programmer's Workflow Checklist

- Define classes for each major banking entity.
- Use `private` for sensitive/internal data.
- Use `protected` for subclass access (e.g., auditors, managers).
- Use `readonly` for account numbers and constants.
- Only expose what's needed via public methods.
- Test access from outside—ensure privacy is enforced.