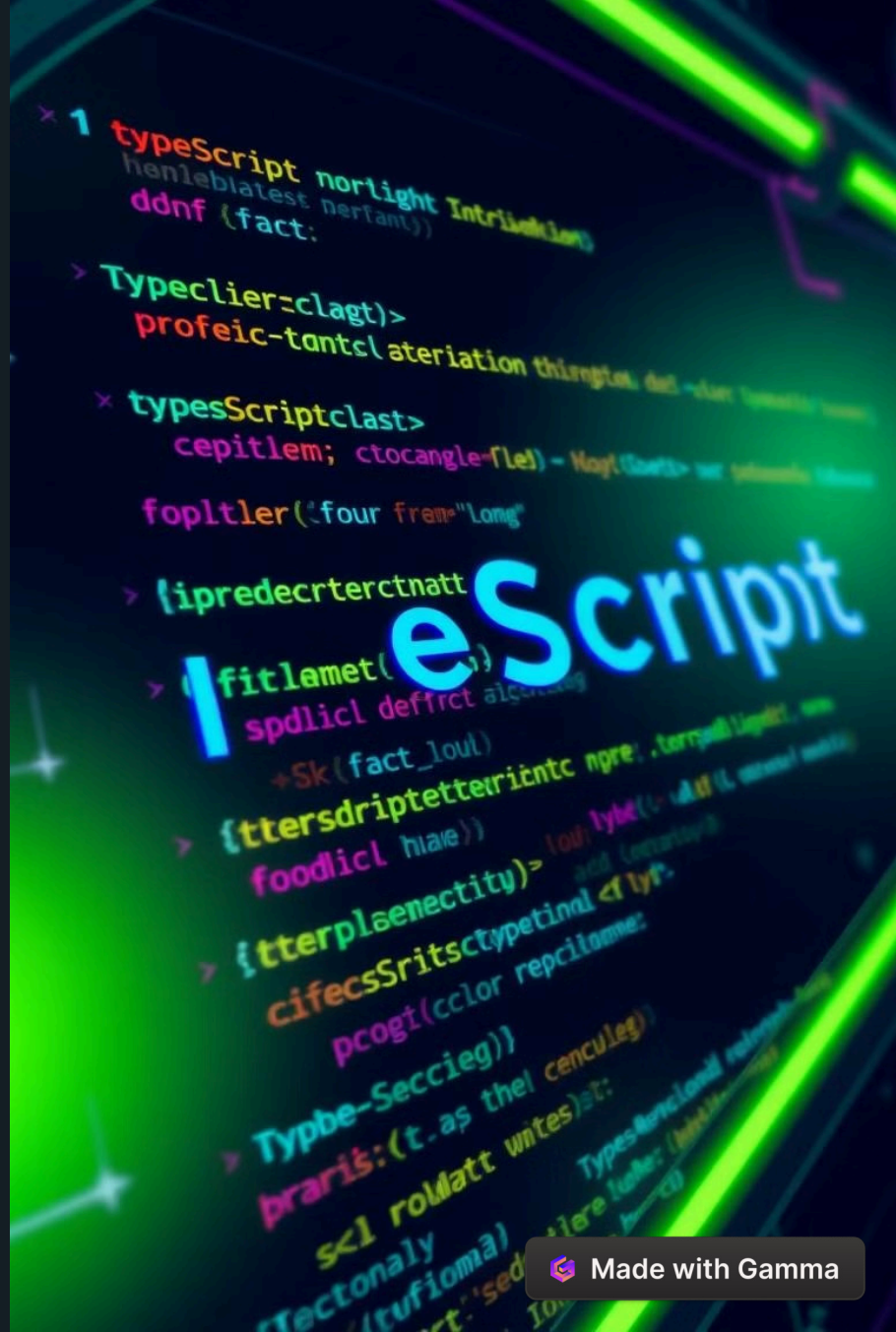
TypeScript Classes: A Comprehensive Guide

This presentation will explore the fundamentals of TypeScript classes, covering their structure, key features, and best practices for writing clean and scalable code.

por Edson Camacho



Understanding the Basics of a TypeScript Class

Declaring a Class

Use the class keyword followed by the class name. For example, class Person { ... }

Properties

Define characteristics of the object. For example, name: string; age: number;

Constructor

A special method used to initialize object properties. For example, constructor(name: string, age: number) { ... }

Methods

Define behaviors for the class. For example, greet(): string { ... }

Key Features and Benefits

1 Strong Typing

Enforces data types at compile-time, reducing runtime errors.

2 Access Modifiers

Control visibility and enforce encapsulation (public, private, protected).

3 Inheritance

Create new classes based on existing ones, inheriting properties and methods.

4 Getters and Setters

Allow controlled access to private properties.

5 Abstract Classes

Define a base structure for other classes but cannot be instantiated directly.

Creating and Extending a TypeScript Class

Creating a Class

Define a class using the class keyword, properties, a constructor, and methods.

Extending a Class

Use the extends keyword to inherit properties and methods from a parent class.

Best Practices for Writing Clean and Scalable Classes

Use Access Modifiers

Control visibility and enforce encapsulation.

Implement Interfaces

Enforce a contract for classes, improving maintainability.

Use Readonly Properties

Prevent modification of properties after initialization.

Leverage Getters and Setters

Control access to properties.

Favor Composition Over Inheritance

Reduce complexity by using composition instead of inheritance when possible.

Use Abstract Classes

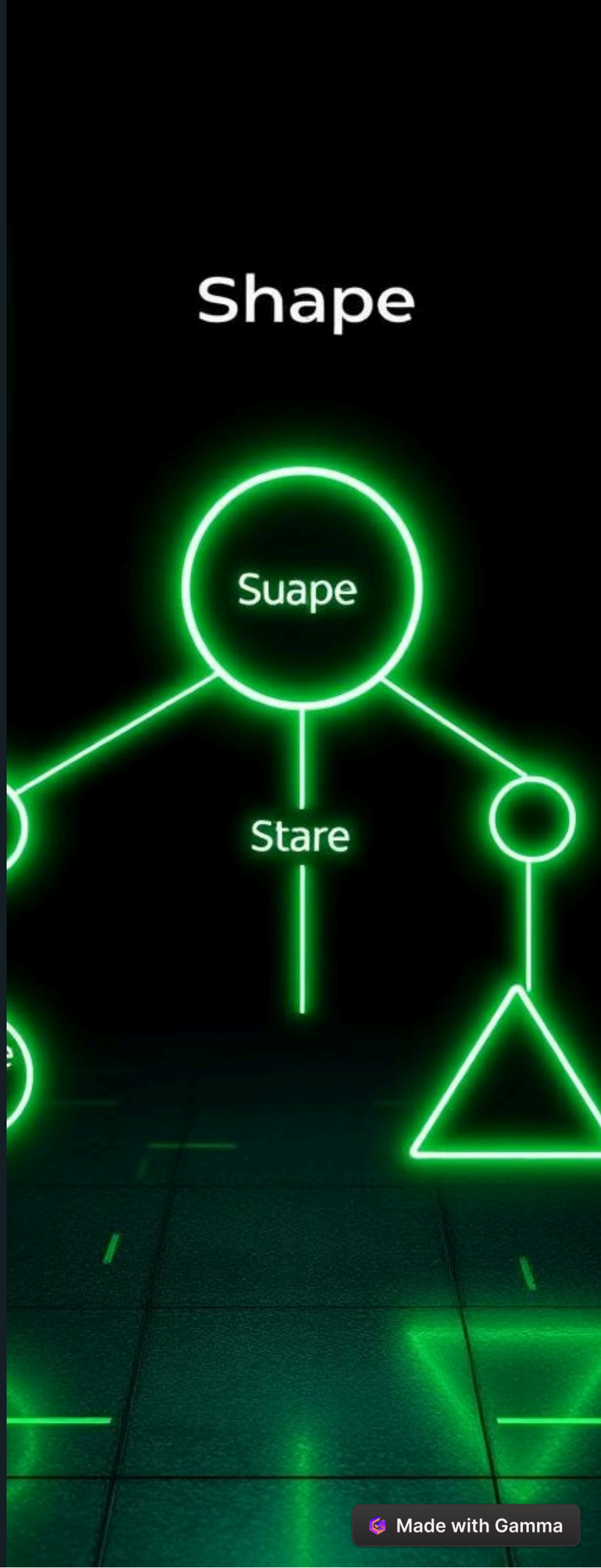
Define a template for subclasses.

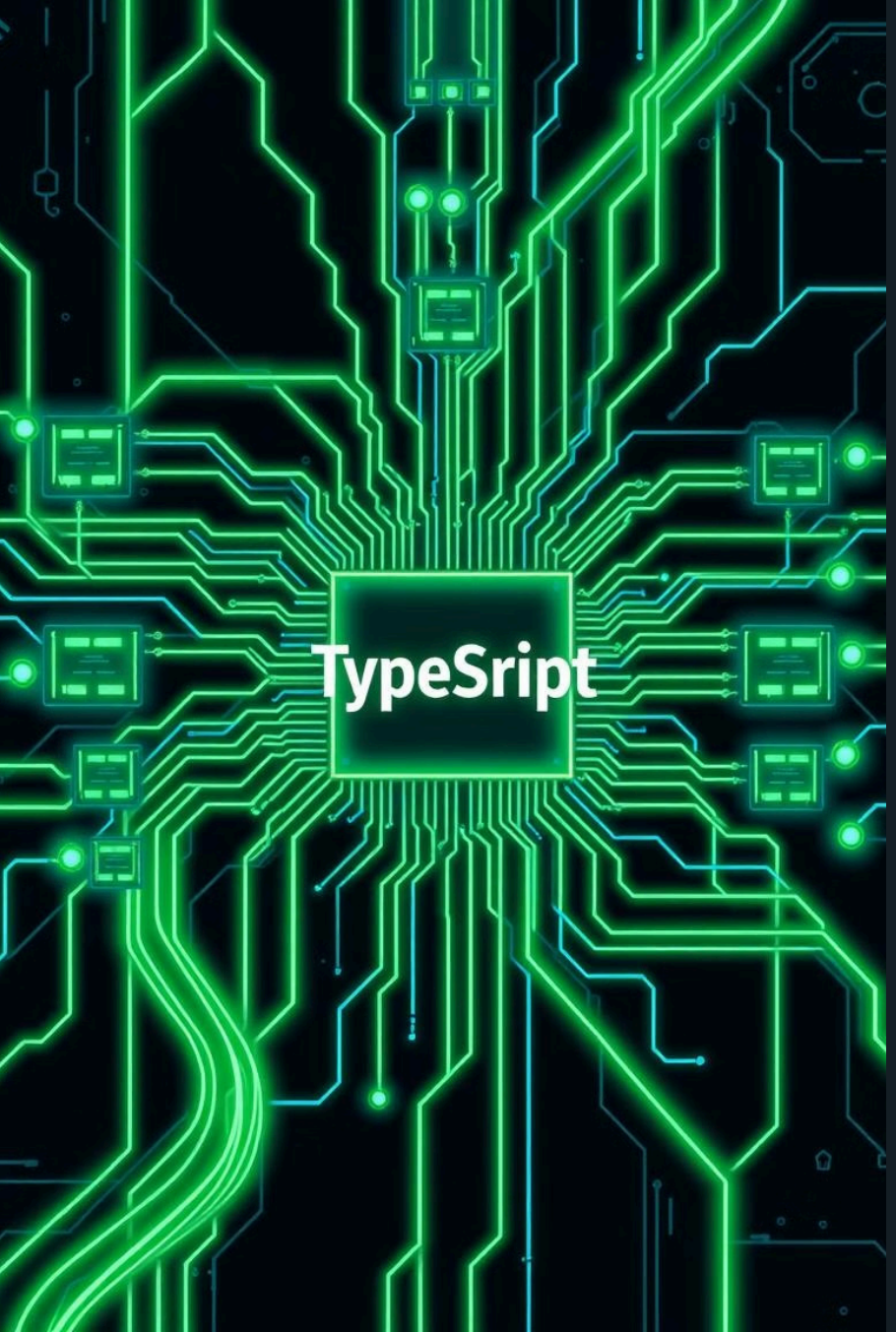
Example: Implementing a Bank Account Class

```
class BankAccount {
  private balance: number;

  constructor(initialBalance: number) {
    this.balance = initialBalance;
  }

  deposit(amount: number): void {
    this.balance += amount;
  }

  withdraw(amount: number): void {
    if (this.balance >= amount) {
      this.balance -= amount;
    } else {
      console.log("Insufficient funds.");
    }
  }

  getBalance(): number {
    return this.balance;
  }
}

const account = new BankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // 1500
account.withdraw(200);
console.log(account.getBalance()); // 1300
```

Example: Implementing a Shape Hierarchy

```
abstract class Shape {
  abstract getArea(): number;
}

class Circle extends Shape {
  constructor(private radius: number) {
    super();
  }

  getArea(): number {
    return Math.PI * this.radius * this.radius;
  }
}

class Square extends Shape {
  constructor(private side: number) {
    super();
  }

  getArea(): number {
    return this.side * this.side;
  }
}

const myCircle = new Circle(5);
console.log(myCircle.getArea()); // 78.54

const mySquare = new Square(4);
console.log(mySquare.getArea()); // 16
```





Conclusion: Leveraging TypeScript Classes for Robust Applications

By understanding the fundamentals of TypeScript classes, we can write more organized, maintainable, and scalable code. TypeScript classes provide a powerful tool for building robust and efficient applications.