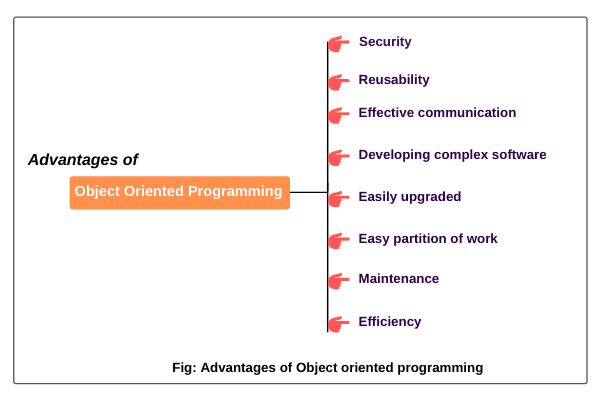
**JavaScript Object-oriented programming**

Developers utilize object-oriented programming to build software based on data. Here is a breakdown of its benefits.

**1. Enables Code Reusability:** It is no longer necessary for programmers to manually develop the same code multiple times because they can reuse code through OOP inheritance.

**2. Increases Productivity in Software Development:** The OOP framework allows programmers to construct programs taking help from existing packages, such as Python, which can save time and boost productivity

**3. Reinforces Security:** Programmers filter out limited data using OOP mechanisms, such as data hiding and abstraction, to keep the application secure. This ensures that only important data can be seen by the users.



**Classes:**

TypeScript supports the definition of classes, which can be used to create objects with properties and methods.

class Person {

constructor(public name: string, public age: number) {}

sayHello() {

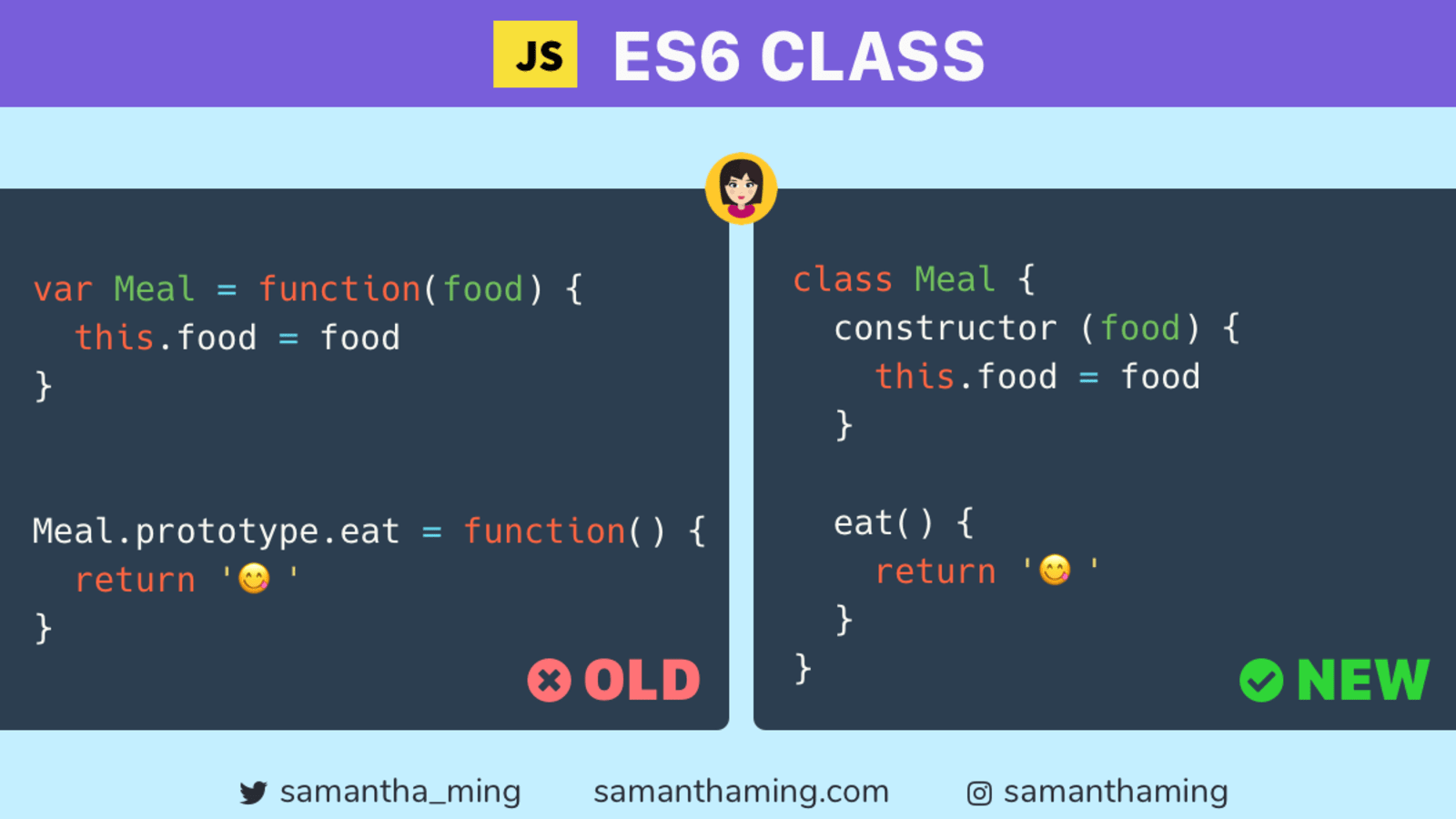
console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

}

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

person.sayHello(); // Output: Hello, my name is Alice and I am 30 years old.



<http://es6-features.org/>

**Inheritance:**

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more. Using class inheritance, a class can inherit all the methods and properties of another class. Inheritance is a useful feature that allows code reusability.

// parent class

class Person {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hello ${this.name}`);

}

}

// inheriting parent class

class Student extends Person {

constructor(name) {

console.log("Creating student class");

// call the super class constructor and pass in the name parameter

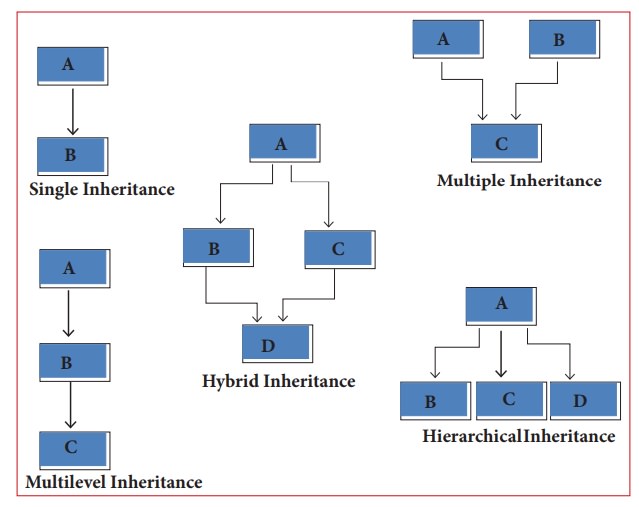
super(name);

}

}

let student1 = new Student('Jack');

student1.greet();



**Encapsulation:**

Encapsulation is the concept of bundling data (attributes or properties) and the methods (functions or behavior) that operate on that data into a single unit called a "class."

It restricts direct access to some of an object's components, providing a controlled interface to interact with the object.

Access modifiers like public, private, and protected control the visibility and accessibility of class members.

**Encapsulation provides several benefits:**

**Modularity**: By encapsulating related data and behavior within a class, you create a self-contained module that can be reused and maintained independently.

**Data Protection:** By making the internal data private (or protected), you prevent unauthorized access and modification from outside the class, ensuring data integrity and security.

**Code Flexibility:** Encapsulation allows you to change the internal implementation of a class without affecting the code that uses the class, as long as the public interface remains unchanged.

TypeScript supports access modifiers like **public**, **private**, and **protected** to control the visibility of class members.

class Car {

private speed: number = 0;

accelerate() {

this.speed += 10;

}

brake() {

this.speed -= 5;

}

getSpeed() {

return this.speed;

}

}

**Abstraction:**

Abstraction is a key concept in Object-Oriented Programming (OOP) that focuses on presenting essential features of an object while hiding unnecessary details. It allows you to represent complex systems or entities in a simplified manner, making it easier to understand and work with them. Abstraction enables you to build models that capture the relevant characteristics of an object, without exposing all the implementation specifics.

In OOP, abstraction is achieved through the use of abstract classes and interfaces. These abstract constructs provide a blueprint for other classes to follow, defining a set of methods and properties without providing their implementation. The concrete subclasses that extend the abstract class or implement the interface are responsible for implementing these abstract elements.

The main benefits of abstraction are:

**Simplified Complexity:** Abstraction allows you to focus on the high-level design and behavior of an object, hiding the intricate details that might not be relevant at that level.

**Code Reusability:** By defining common interfaces through abstract classes or interfaces, you promote code reuse. Concrete classes can inherit from an abstract class or implement an interface, inheriting its structure and behavior.

**Flexibility:** Abstraction enables you to change the implementation details of concrete classes without affecting the overall functionality of the program, as long as the abstract interface remains unchanged.

abstract class Shape {

abstract calculateArea(): number;

}

class Circle extends Shape {

constructor(private radius: number) {

super();

}

calculateArea(): number {

return Math.PI \* this.radius \* this.radius;

}

}

const circle = new Circle(5);

console.log(circle.calculateArea()); // Output: 78.53981633974483

**Polymorphism:**

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. In TypeScript, which is a statically typed superset of JavaScript, polymorphism is achieved through the use of inheritance and interfaces. Here's how polymorphism can be implemented in TypeScript:

**Function overloading:**

class MathOperations {

add(a: number, b: number): number;

add(a: string, b: string): string;

add(a: any, b: any): any {

return a + b;

}

**Function Overriding:**

class Animal {

constructor(public name: string) {}

makeSound() {

console.log("Some generic sound");

}

}

class Dog extends Animal {

makeSound() {

console.log("Woof!");

}

}

class Cat extends Animal {

makeSound() {

console.log("Meow!");

}

}

const dog: Animal = new Dog("Buddy");

const cat: Animal = new Cat("Whiskers");

dog.makeSound(); // Output: Woof!

cat.makeSound(); // Output: Meow!

These are some of the core Object-Oriented Programming features in TypeScript. TypeScript allows you to write more structured and maintainable code by leveraging these features to create reusable and extensible classes and objects.

In JavaScript, there are several ways to create classes, each with its own syntax and features. Here are the primary ways to create classes in JavaScript:

**1. Constructor Functions:**

You can create classes using constructor functions. These functions are typically named with an initial capital letter by convention.

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.sayHello = function () {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

};

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

person.sayHello();

**2. ES5 Prototype-based Classes**

You can define classes using the prototype-based approach, where you attach methods to the prototype of a constructor function.

function Person(name, age) {

this.name = name;

this.age = age;

}

Person.prototype.sayHello = function () {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

};

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

person.sayHello();

**3. ES6 (ES2015) Classes:**

ECMAScript 2015 introduced a class syntax that provides a more concise and familiar way to create classes.

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayHello() {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

}

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

person.sayHello();

**4. Factory Functions:**

You can create classes using factory functions that return objects with methods.

function createPerson(name, age) {

return {

name,

age,

sayHello() {

console.log(`Hello, my name is ${name} and I am ${age} years old.`);

}

};

}

const person = createPerson("Alice", 30);

person.sayHello();

**5. Object.create():**

You can create objects using `Object.create()` to set the prototype of an object explicitly.

const personPrototype = {

sayHello() {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

};

const person = Object.create(personPrototype);

person.name = "Alice";

person.age = 30;

person.sayHello();

**6. Class Expressions:**

You can also use class expressions to create classes, which are similar to class declarations but can be unnamed.

const Person = class {

constructor(name, age) {

this.name = name;

this.age = age;

}

sayHello() {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

};

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

person.sayHello();

Each of these approaches has its own advantages and use cases, and the choice of which one to use depends on your project requirements and coding style preferences. ES6 classes are the most modern and widely adopted way to create classes in JavaScript, but the other methods are still relevant in certain situations, especially when working with legacy code or specific design patterns.