OOPs Concepts

Chapter-4

Aitrich

Object-oriented programming

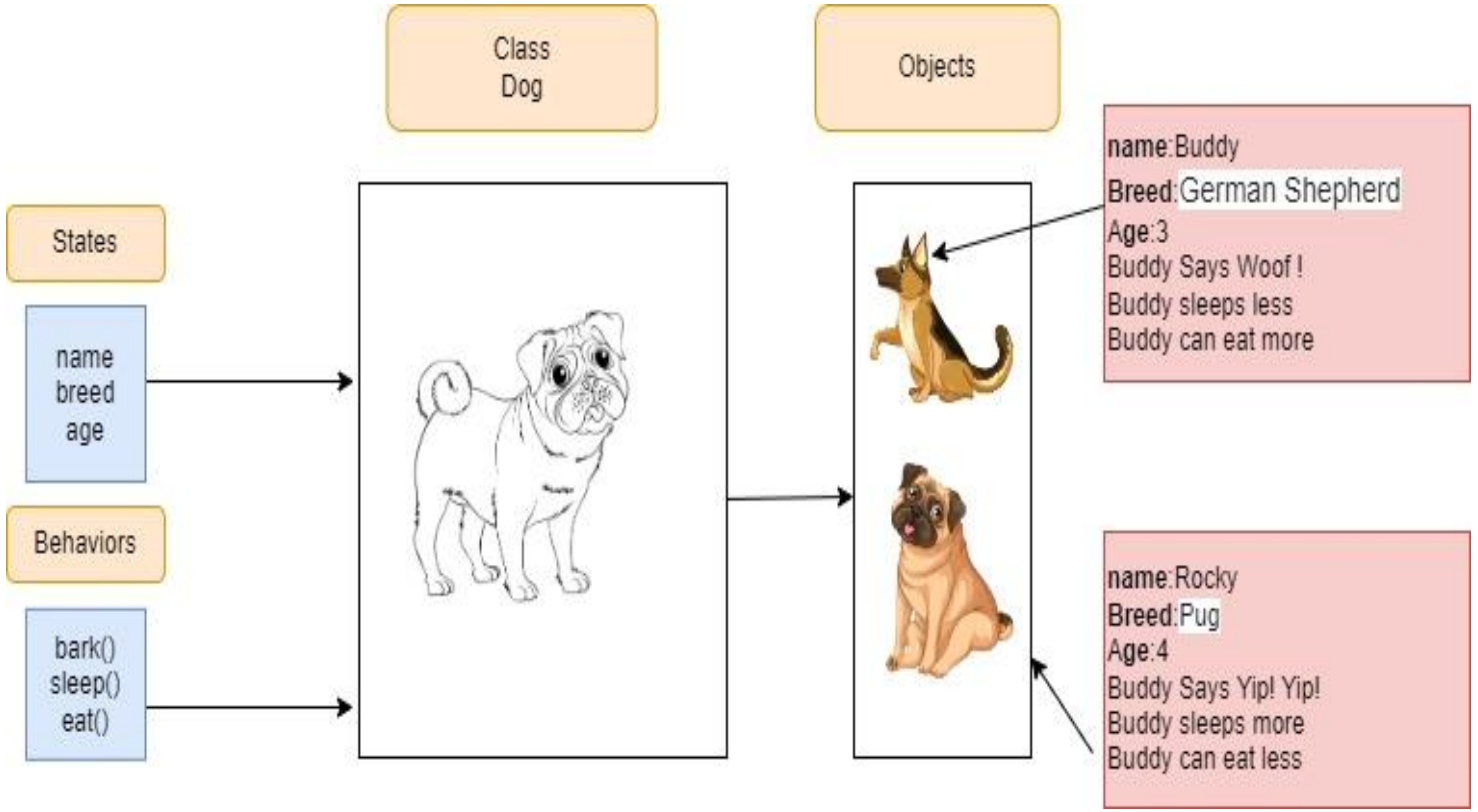
Object-oriented programming is a programming paradigm based on classes and objects rather than functions and logic.

Object-oriented programming in TypeScript differs from Object-oriented programming in JavaScript because, unlike JavaScript, TypeScript has full class support, has access modifiers, and type annotations like most object-oriented programming languages

Class

- Class is a blueprint used to create an instance of an object.
- In typescript a class contains
  - constructor
  - properties
  - methods
- When we create a class we can create multiple objects of the class.
- Class is a logical entity, and object is a physical entity.

Class Example




```
class Dog{
    name!: string;
    breed!: string;
    age!: number;
     display():void{
        console.log(this.name);
        console.log(this.breed);
        console.log(this.age);
     }
}
var dogRef=new Dog();
dogRef.name="buddy";
dogRef.breed="German Shepherd";
dogRef.age=3;
dogRef.display();
```

Constructor

- Constructors are identified with the keyword "constructor".
- A class may contain at least one constructor declaration. If a class has no constructor, a constructor is provided automatically.
- A class can have any number of constructors.
- A constructor is a special function of the class that is automatically invoked when we create an instance of the class in Typescript.
- The constructor method is invoked every time we create an instance from the class using the new operator.
- We can access the current instance using the ‘this‘ inside the constructor.

Constructor

```
class Dog{
    name!: string;
    breed!: string;
    age!: number;
    setData(name:string,breed:string,age:number){
        this.name=name;
        this.breed=breed;
        this.age=age;
    }
     display():void{
        console.log(this.name);
        console.log(this.breed);
        console.log(this.age);
     }
}
var dogRef=new Dog();
dogRef.setData('buddy','German Shepherd',3)
dogRef.display();
```

```
class Dog{
    name!: string;
    breed!: string;
    age!: number;
    constructor(name:string,breed:string,age:number){
        this.name=name;
        this.breed=breed;
        this.age=age;
    }
     display():void{
        console.log(this.name);
        console.log(this.breed);
        console.log(this.age);
     }
}
var dogRef=new Dog('buddy','German Shepherd',3);
dogRef.display();
```

Inheritance

- Inheritance is one of the core concepts of object-oriented programming (OOPs).
- Inheritance is acquiring all the variables and methods from one class to another class.
- It helps to reuse the code .
- The class whose properties and methods are inherited is called the parent class(Super or Base class).
- The class which inherits properties and methods is called the child class(sub-class or Derived class).
- Typescript classes can be extended to create new classes with inheritance, using the keyword extends.
- TypeScript supports only single inheritance and multilevel inheritance

```
class GermanShepherd extends Dog {
  constructor(name: string, age: number) {
    super(name, 'German Shepherd', age);
  }

  bark(): void {
    console.log(`${this.name} barks like a German Shepherd! Woof! Woof!`);
  }
}

class Pug extends Dog {
  constructor(name: string, age: number) {
    super(name, 'Pug', age);
  }

  bark(): void {
    console.log(`${this.name} makes a cute Pug bark! Woof!`);
  }
}
```

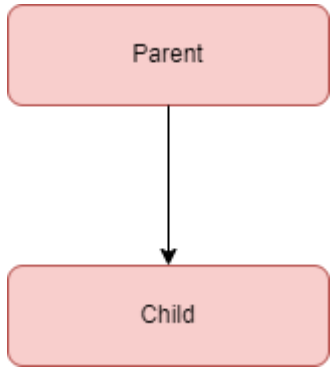
```
class Dog {
  protected name: string;
  protected breed: string;
  protected age: number;

  constructor(name: string, breed: string, age: number) {
    this.name = name;
    this.breed = breed;
    this.age = age;
  }

  eat(): void {
    console.log(`${this.name} is eating.`);
  }

  sleep(): void {
    console.log(`${this.name} is sleeping.`);
  }

  bark(): void {
    console.log(`${this.name} says Woof!`);
  }
}
```

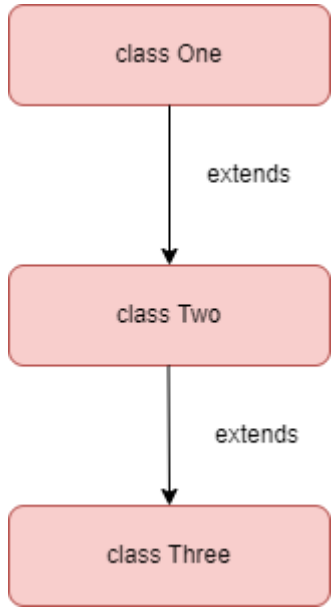
Types Of Inheritance

- Two types of inheritance

  Single Inheritance.

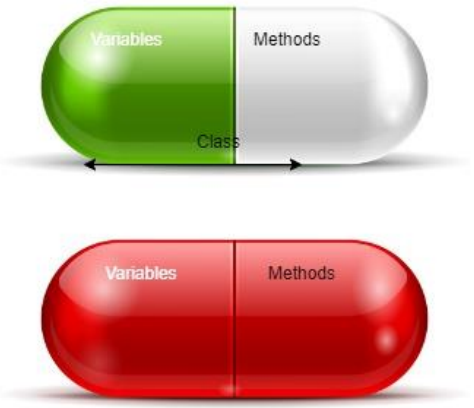
  Multilevel Inheritance.



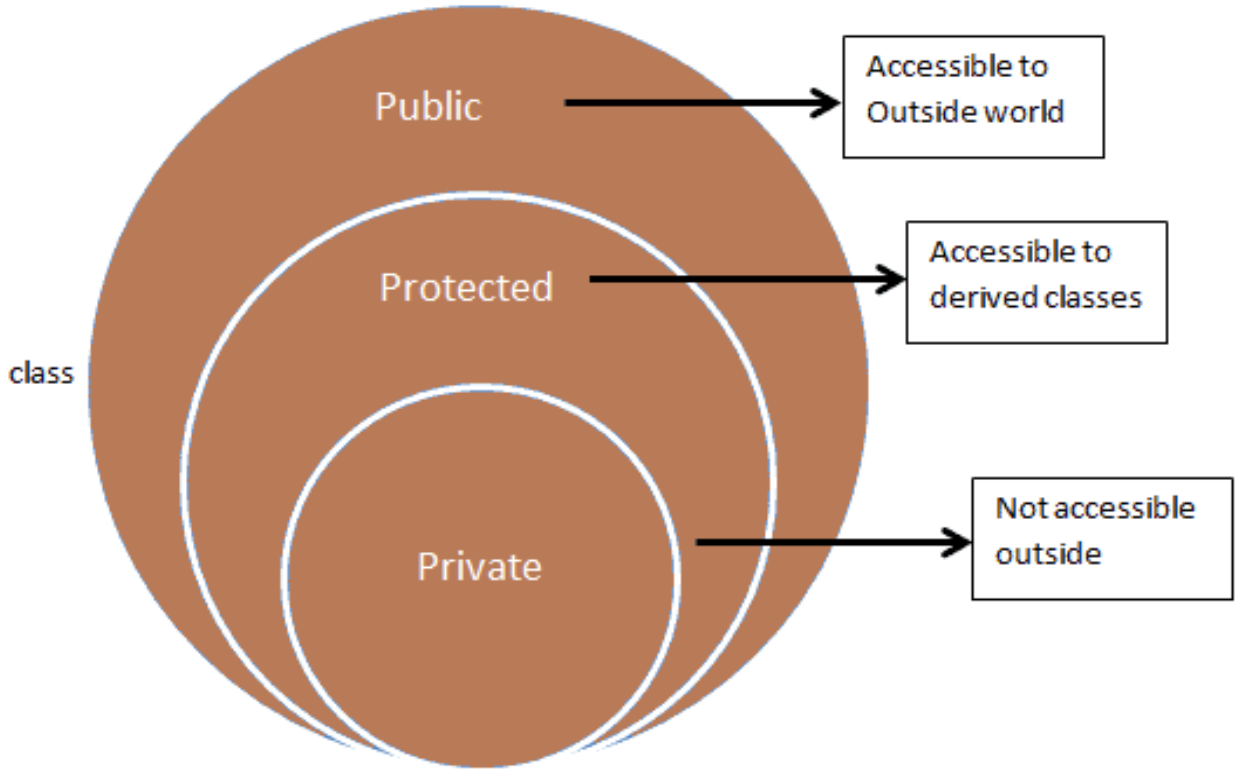


Encapsulation

- Encapsulation in object-oriented programming refers to restricting unauthorized access and mutation of specific properties of an object.
- Bundling the data (properties) and methods that operate on the data within a class, and controlling the access to these members.
- In TypeScript, access modifiers are used to achieve e



- There primary access modifiers in TypeScript:
  - Public
  - private
  - protected



Encapsulation Example

```
class Dog {
    private name: string;
    private age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;

    }
    // Getter for name
    public getName(): string {
        return this.name;
    }
     // Setter for name
    public setName(name: string): void {
        this.name = name;
    }
    // Getter for age
    public getAge(): number {
        return this.age;
    }
 // Setter for age
    public setAge(age: number): void {
        this.age = age;
    }
 // Other methods
    public bark(): void {
        console.log(`${this.name} is barking!`);
    }
}
```

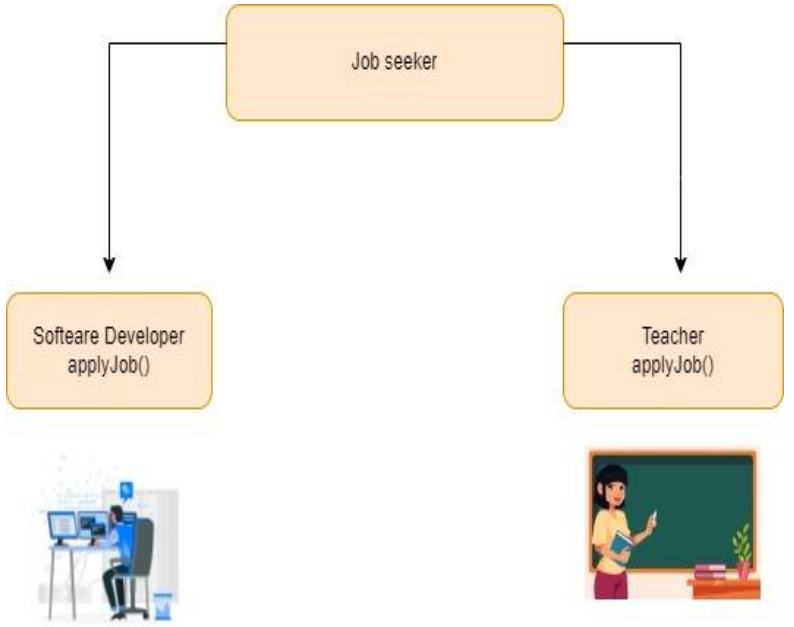
```
// Example usage
const myDog = new Dog('Buddy', 3, 'Golden Retriever');

console.log(`Before: ${myDog.getName()} is ${myDog.getAge()} years old.`);
myDog.setAge(4);
console.log(`After: ${myDog.getName()} is ${myDog.getAge()} years old.`);
myDog.bark();
```

Polymorphism

Classes have the same methods but different implementations.

Polymorphism is the ability to create a class that has more than one form.

To achieve polymorphism, inherit from a base class, then override methods and write implementation code in them



Polymorphism Example

```
class JobSeeker {
    constructor(public name: string, public profile: string) {}

    applyForJob(): void {
        console.log(`${this.name} is applying for a job with a ${this.profile} profile.`);
    }
}

class DeveloperProfile extends JobSeeker {
    constructor(name: string, public programmingLanguage: string) {
        super(name, 'Developer');
    }

    // Overriding applyForJob method
    applyForJob(): void {
        console.log(`${this.name} is applying for a developer position using ${this.programmingLanguage}.`);
    }
}

class TeacherProfile extends JobSeeker {
    constructor(name: string, public teachingExperience: number) {
        super(name, 'Teacher');
    }

    // Overriding applyForJob method
    applyForJob(): void {
        console.log(`${this.name} is applying for a teacher position with ${this.teachingExperience} years of experience.`);
    }
}
```

```
const developerJobSeeker = new DeveloperProfile('Alice', 'JavaScript');
const teacherJobSeeker = new TeacherProfile('Bob', 5);
developerJobSeeker.applyForJob();
teacherJobSeeker.applyForJob();
```

Output:-

```
"Alice is applying for a developer position using JavaScript."

"Bob is applying for a teacher position with 5 years of experience."
```

Data Abstraction

- Abstraction is another one of the simpler concepts when it comes to the four principles of OOP.

- It is the process of hiding the internal complexity of a class while only requiring the absolute necessary data to function correctly.

Data Abstraction Example

```typescript
abstract class Job {
    constructor(public title: string, public description: string) {}

    // Abstract method for applying to a job
    abstract apply(): void;
}

class SoftwareEngineerJob extends Job {
    constructor(public title: string, public description: string, public programmingLanguage: string) {
        super(title, description);
    }

    // Implementing the abstract apply method
    apply(): void {
        console.log(`Applying for Software Engineer position using ${this.programmingLanguage}.`);
    }
}
const softwareEngineerJob = new SoftwareEngineerJob('Software Engineer', 'Develop software applications', 'Typescript');
softwareEngineerJob.apply();
```

Output:-

```
"Applying for Software Engineer position using Typescript."
```

Interface

- One of TypeScript's core principles is that type-checking focuses on the shape that values have. This is sometimes called "duck typing" or "structural subtyping".
- In TypeScript, interfaces fill the role of naming these types and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.
- In other words, an interface defines the syntax that any entity must adhere to.
- Interfaces define properties, methods, and events, which are the members of the interface.
- Interfaces contain only the declaration of the members.
- It is the responsibility of the deriving class to define the members.

Interface Example

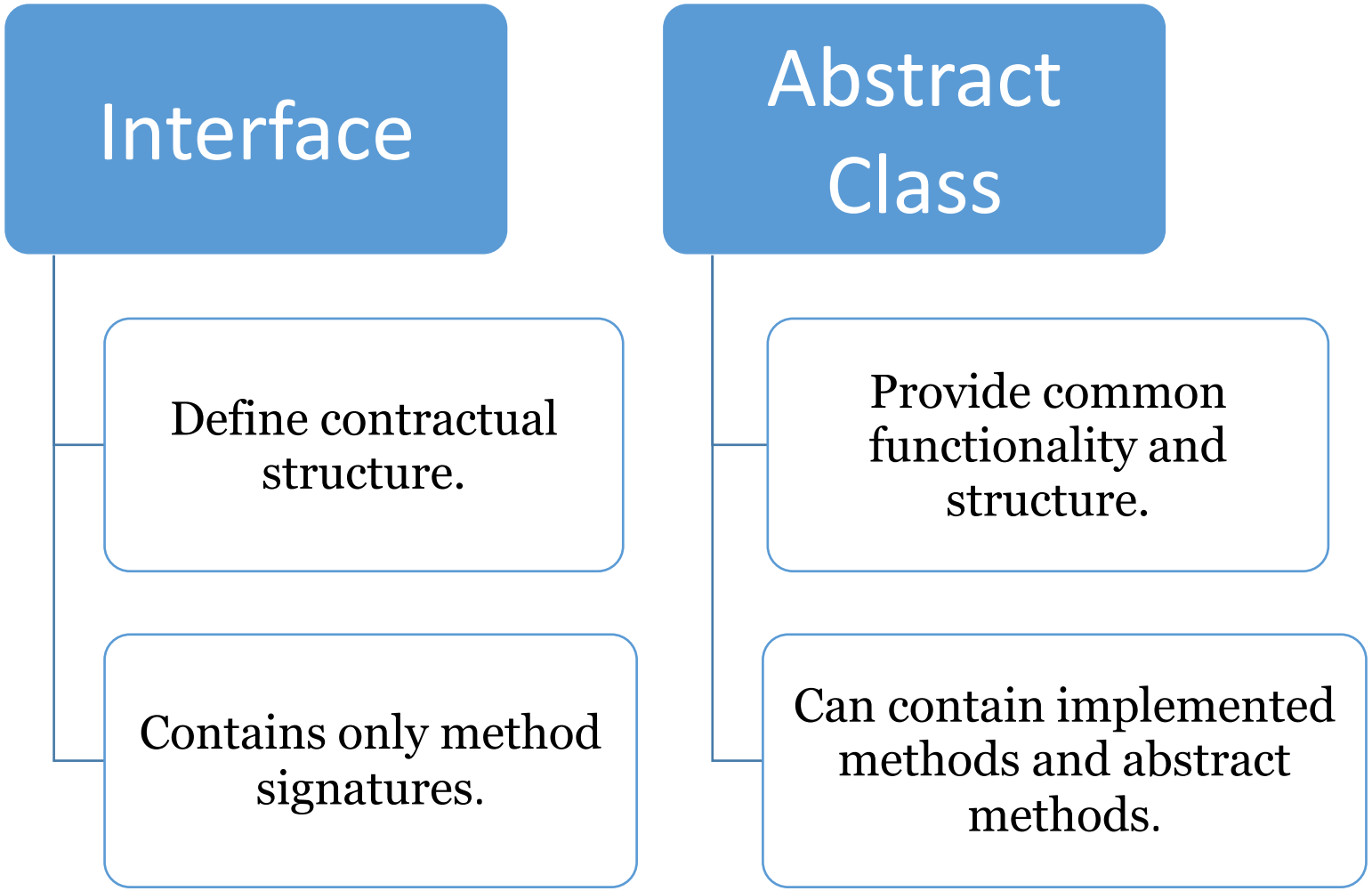
```typescript
// Define the Job interface
interface Job {
  title: string;
  description: string;
  salary: number;
  location: string;
// Abstract method for performing the job
  performJob(): void;
}
```

Output:-

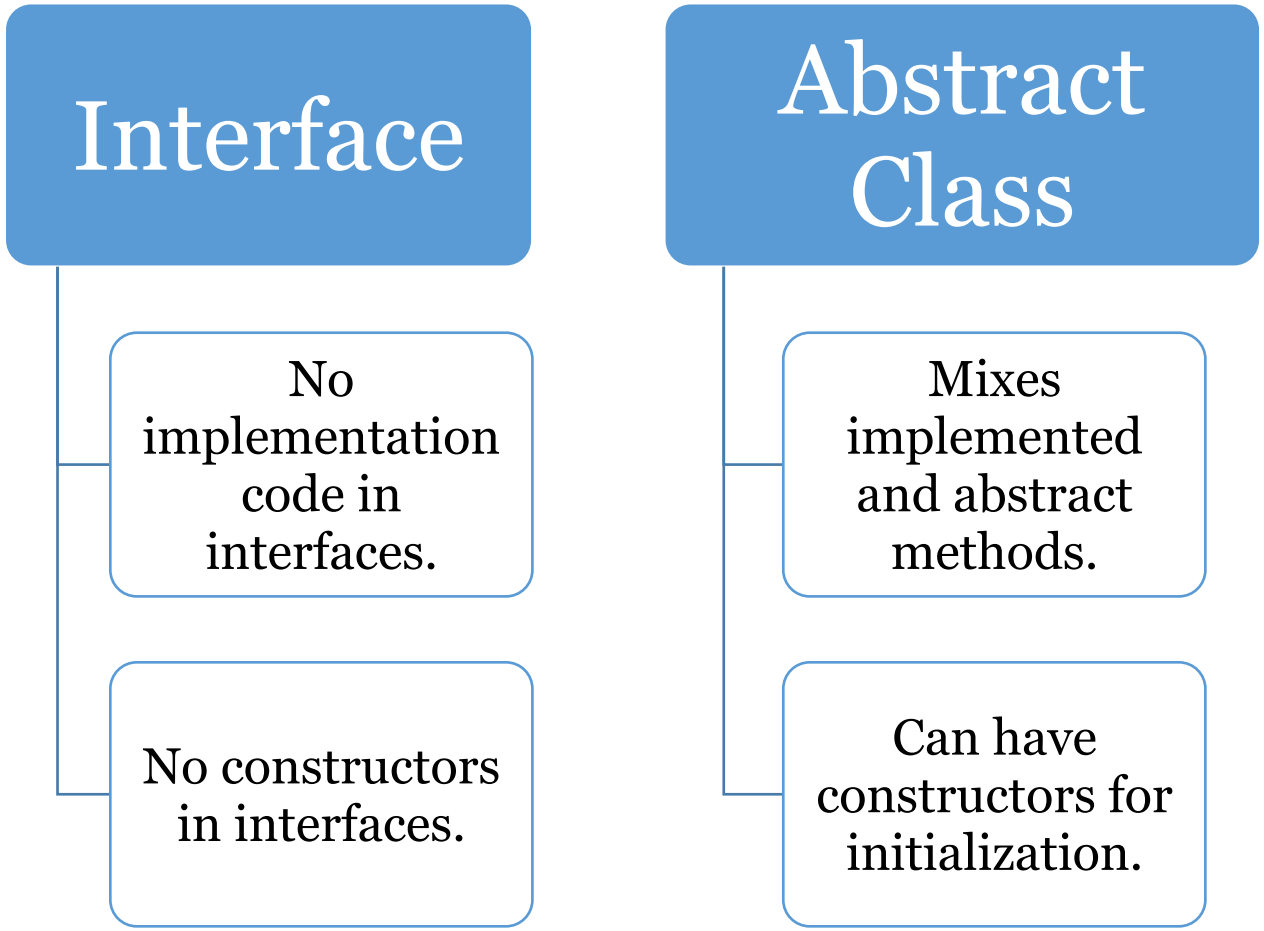
```
[LOG]: SoftwareEngineer: {
  "title": "Software Engineer",
  "description": "Develops software applications",
  "salary": 80000,
  "location": "San Francisco"
}

[LOG]: "As a Software Engineer, I am coding and solving complex problems."
```

```typescript
// Example implementation of a specific job type
class SoftwareEngineer implements Job {
  title: string;
  description: string;
  salary: number;
  location: string;

  constructor(title: string, description: string, salary: number, location: string) {
    this.title = title;
    this.description = description;
    this.salary = salary;
    this.location = location;
  }

  // Implementation of the abstract method
  performJob(): void {
    console.log(`As a ${this.title}, I am coding and solving complex problems.`);
  }
}
// Example usage
const softwareEngineerJob: Job = new SoftwareEngineer(
  "Software Engineer",
  "Develops software applications",
  80000,
  "San Francisco"
);
console.log(softwareEngineerJob);
softwareEngineerJob.performJob();
```

Interface & Abstract class



Interface & Abstract class




THANK
YOU