Abstract Classes and Interfaces

Abstract Classes:

Definition:

An abstract class is like a blueprint for other classes. It gives a structure and some functionality, but you can't make an object directly from it. It can have both outlined plans (methods) and some real details (actual code).

Key Points:

Constructors Allowed: Abstract classes can have a starting setup called a constructor.

Mix of Ideas: It allows a mix of ideas - some just plans (abstract methods) and some ready-to-use (concrete methods).

One Buddy Only: A class can only follow the plan of one abstract class.

Example in Java:

```
abstract class Animal {
    abstract void makeSound(); // Just a plan
    void sleep() {
        System.out.println("Zzzz");
    }
}
```

Interfaces:

Definition:

An interface is like a contract. It says, "If you want to be my friend (implement me), you must do these things (have these methods)." It's all about what to do, not how to do it.

Key Points:

No Constructors: No initial setup allowed here.

Only Plans: It's all about plans - just saying what to do, not actually doing it.

Friends with Many: A class can be friends with many interfaces, not just one.

Example in C#:

```
interface IShape {
    void Draw(); // Just a plan
    void Move(); // Another plan
}
```

Differences:

Constructors:

Abstract classes can have a starting setup.

Interfaces don't allow any initial setup.

Plans and Actions:

Abstract classes can have a mix of plans and actions.

Interfaces only have plans.

Number of Friends:

Abstract classes are like having one best friend - you follow their plans.

Interfaces are like having many friends, each with their own set of plans.

Interfaces in JavaScript:

JavaScript is a bit different. It doesn't have a clear way of saying, "This is my plan, and you have to follow it." But we can still make things work.

How it Works in JavaScript:

1.Using Objects:

We can make an object with plans (methods) and ask others to follow them.

```
const animalInterface = {
    makeSound: function() {
        throw new Error('makeSound must be implemented');
    }
};
```

2. Using Classes:

We can use class-style code to define plans and implement them.

```
class AnimalInterface {
    makeSound() {
        throw new Error('makeSound must be implemented');
    }
}


class Dog extends AnimalInterface {
    // Implementing makeSound
    makeSound() {
        console.log('Woof!');
    }
}
```

JavaScript isn't strict about following plans like some other languages, but these approaches help us create similar structures.