interfaces are meant to define contracts with fixed values that all implementing classes can use.

The reason breaks down to:

1. `public` - Because interfaces define public contracts
2. `static` - Because interfaces can't have instance state
3. `final` - Because interfaces shouldn't allow modification of these shared values

This maintains the interface's role as a pure behavior contract while allowing shared constants.

Static Method vs. Static Block in Java

No, static methods and static blocks are **not** the same, though they both relate to the `static` keyword in Java. Here's the key difference:

Static Method

- A **static method** is a function that belongs to the class rather than any object instance
- You call it using the class name (e.g., `ClassName.methodName()`)
- Used to perform operations that don't require instance data

```java
class MathUtils {

    public static int add(int a, int b) {

        return a + b;

    }

}
// Called as: MathUtils.add(5, 3)
```

Static Block

- A **static block** (or static initialization block) is a code block that runs when the class is first loaded
- Used to initialize static variables or perform one-time setup
- There's no way to call it directly - it executes automatically

```
class Database {
    static Connection conn;
    static {// This runs when class loads
        try {
            conn = DriverManager.getConnection(...);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Key Differences

Feature	**Static Method**	**Static Block**
Purpose	Perform reusable operations	One-time class initialization
Execution	Called explicitly when needed	Runs automatically when class loads

Syntax	Has method signature (name, params)	Just `static { ... }` code block
Invocation	Can be called multiple times	Runs only once
Return	Can return values	Cannot return anything

When to Use Which:

- Use **static methods** for utility functions you need to call repeatedly
- Use **static blocks** for one-time setup of static resources (like database connections)

Complete Guide to Abstract Class in Java

An **abstract class** in Java is a class that cannot be instantiated (you cannot create objects of it directly). It serves as a **blueprint** for other classes and can contain both **abstract (unimplemented) methods** and **concrete (implemented) methods**.

Key Features of Abstract Class

1. Cannot Be Instantiated

- You **cannot** create an object of an abstract class.
- It must be **extended** by a subclass, which provides implementations for its abstract methods.

```
abstract class Animal { }
```

```java
Animal a = new Animal(); // ❌ Error: Cannot instantiate abstract class
```

---

2. Can Have Abstract Methods (No Body)

- Abstract methods **do not have an implementation** and must be overridden by subclasses.
- They are declared using the `abstract` keyword.

```java
abstract class Animal {
    abstract void makeSound(); // No body, must be implemented by subclass
}


class Dog extends Animal {
    void makeSound() { // ✅ Must override abstract method
        System.out.println("Bark!");
    }
}
```

---

3. Can Have Concrete Methods (With Implementation)

- Unlike interfaces (before Java 8), abstract classes can have **fully implemented methods**.

```java
abstract class Animal {
    void breathe() { // ✅ Concrete method
        System.out.println("Breathing...");
```

```
    }
}
```

---

4. Can Have Constructors (Unlike Interfaces)

- Abstract classes can have **constructors**, which are called when a subclass is instantiated.

```
abstract class Animal {
    Animal() { System.out.println("Animal constructor called"); }}
class Dog extends Animal {
    Dog() {
        super(); calls the constructor
    }
}
```

---

5. Can Have Instance Variables (Stateful)

- Unlike interfaces (which can only have `public static final` constants), abstract classes can have **any type of fields**.

```
abstract class Animal {
    String name; // ✅ Instance variable
    int age;     // ✅ Another field
}
```

---

6. Supports Access Modifiers (private, protected, etc.)

- Methods and fields can have **any access modifier** (`public`, `private`, `protected`, default).

```
abstract class Animal {
    private String secretName; // ✅ Private field
    protected void eat() {      // ✅ Protected method
        System.out.println("Eating...");
    }
}
```

---

7. Can Have Static Methods (Like Regular Classes)

- Abstract classes can define **static methods**, which can be called without an object.

```
abstract class Animal {
    static void info() {
        System.out.println("This is an Animal class");
    }
}

Animal.info(); // ✅ Called without instantiation
```

---

8. Can Have a Static Block (For Initialization)

- Like regular classes, abstract classes can have a **static block** for one-time initialization.

```
abstract class Animal {
    static {
        System.out.println("Animal class loaded!");
    }
}
```

When to Use Abstract Class vs. Interface?

Feature	Abstract Class	Interface (Pre-Java 8)	Interface (Java 8+)
Instantiation	❌ Cannot be instantiated	❌ Cannot be instantiated	❌ Cannot be instantiated
Methods	✅ Abstract + Concrete	❌ Only abstract	✅ Abstract + Default + Static
Fields	✅ Any variables	❌ Only `public static final`	❌ Only `public static final`
Constructors	✅ Yes	❌ No	❌ No

Multiple Inheritance	❌ Extends only one class	✅ Implements multiple interfaces	✅ Implements multiple interfaces
Access Modifiers	✅ Any (`private`, `protected`, etc.)	❌ All methods are `public`	❌ All methods are `public`

When to Use Abstract Class?

✔ When you want to **share code** between related classes.

✔ When you need **constructors** or **non-static fields**.

✔ When you need **partial implementation** (some methods implemented, some left abstract).

When to Use Interface?

✔ When you need **multiple inheritance** of type.

✔ When defining a **contract** (what a class should do, not how).

✔ When you want **loose coupling** (unrelated classes can implement the same interface).

Final Summary

- **Abstract classes** provide **partial implementation** and **state management**.
- **Interfaces** (before Java 8) were purely abstract, but now support **default & static methods**.
- **Abstract class** is best for **code reuse**, while **interface** is best for **defining behavior contracts**. Would you like a practical example comparing both? 😊

Interfaces vs. Abstract Classes: Key Similarities and Differences

While interfaces and abstract classes share some similarities in Java (especially since Java 8), they are **not the same**. Let me break down their relationship and clarify when to use each.

Core Similarities

1. **Cannot Be Instantiated Directly**
   - Both cannot be created with `new`:

```
new MyInterface();      // ❌ Compile error
```

   - `new MyAbstractClass();  // ❌ Compile error`
2. **Can Contain Abstract Methods**
   - Both can declare methods that must be implemented by subclasses:

```
interface Drivable { void drive(); } // Abstract in interface
```

   - `abstract class Vehicle { abstract void move(); } // Abstract in class`
3. **Support Inheritance**
   - Classes can `extend` an abstract class or `implement` an interface.
4. **Can Have Default/Concrete Methods (Java 8+)**
   - Interfaces can now have `default` methods:

```
interface Drivable {
    default void startEngine() { System.out.println("Engine started!"); }
```

   - `}`

- Abstract classes have always supported concrete methods.

Key Differences

Feature	Interface	Abstract Class
Instantiation	❌ Never	❌ Never
Method Types	Abstract + `default` + `static` (Java 8+)	Abstract + Concrete + `static`
Fields (Variables)	Only `public static final` (constants)	Any (`private`, `protected`, etc.)
Constructors	❌ No	✅ Yes
Multiple Inheritance	✅ A class can implement many interfaces	❌ Only one abstract class extension
Access Modifiers	All methods are `public` by default	Can be `private`, `protected`, etc.
Purpose	Defines **what** (contract/behavior)	Defines **what + partial how** (shared logic)

When to Use Which?

Use an Interface When:

✔ You need **multiple inheritance** (a class can implement many interfaces).

✔ You want to define a **contract** (what a class must do, not how).

✔ You need **loose coupling** (unrelated classes can implement the same interface).

✔ You want to **extend functionality** without breaking existing code (using `default` methods).

Use an Abstract Class When:

✔ You need **code reuse** (shared method implementations).

✔ You require **constructors** or **non-static fields**.

✔ You need **partial implementation** (some methods implemented, some left abstract).

✔ You want **controlled access** (`private`/`protected` methods/fields).

Example: Interface vs. Abstract Class

Interface Example (Focus on Behavior)

```
interface Flyable {

    void fly(); // Abstract method

    default void takeOff() { System.out.println("Taking off..."); } // Default
method

}
```

```java
class Bird implements Flyable {

    @Override

    public void fly() { System.out.println("Bird is flying!"); }
}
```

Abstract Class Example (Shared Logic + State)

```java
abstract class Animal {

    String name; // Instance variable

    Animal(String name) { this.name = name; } // Constructor

    abstract void makeSound(); // Abstract method

    void breathe() { System.out.println(name + " is breathing..."); } //
Concrete method

}

class Dog extends Animal {

    Dog(String name) { super(name); }

    @Override

    void makeSound() { System.out.println(name + " says: Woof!"); }
}
```

Java 8+ Blurred the Lines, But Core Differences Remain

- **Interfaces** gained `default` and `static` methods, making them more like abstract classes.
- **But** interfaces still cannot have **constructors**, **instance variables**, or **private methods** (except `private` methods in Java 9+).

- **Abstract classes** remain better for **stateful hierarchies**, while **interfaces** excel at **behavior contracts**.

Final Verdict

- **Need multiple inheritance + behavior definitions? → Interface**
- **Need code reuse + state management? → Abstract Class**

Would you like a real-world analogy to better understand the difference? 😊

Functional Interface in Java

A **Functional Interface** is an interface that has exactly one abstract method, but it can have multiple default or static methods. Functional interfaces are meant to be used primarily with **lambda expressions** or method references in Java.

Key Characteristics of Functional Interfaces:

1. **Single Abstract Method**: A functional interface must have exactly one abstract method. It can have multiple default or static methods.

2. **@FunctionalInterface Annotation**: While not required, it's a good practice to use the `@FunctionalInterface` annotation to indicate that an interface is intended to be a functional interface. This annotation helps to prevent you from accidentally adding more abstract methods, as the compiler will give an error if there is more than one abstract method.

Example of a Functional Interface:

```
@FunctionalInterface
public interface MyFunctionalInterface {
                               // Single abstract method
    void myMethod();

    // Can have default methods
    default void defaultMethod() {
        System.out.println("This is a default method");
    }

    // Can have static methods
```

```java
    static void staticMethod() {
        System.out.println("This is a static method");
    }
}
```

Common Examples of Functional Interfaces in Java:

- `Runnable`
- `Callable`
- `Comparator`
- `Consumer`
- `Function`
- `Predicate`
- `Supplier`

Lambda Expressions in Java

A **Lambda Expression** is a concise way to represent an instance of a functional interface using an expression. Lambdas allow you to treat functionality as a method argument or pass a block of code around.

Lambda expressions help to eliminate the boilerplate code, especially when working with functional interfaces.

Syntax of a Lambda Expression:

```
(parameters) -> expression
```

- **Parameters**: You can pass parameters to the lambda. If there is only one parameter, you can omit the parentheses.
- **Expression**: The body of the lambda expression, which can either be a single expression or a block of code.

Basic Example of Lambda Expression:

```java
@FunctionalInterface
public interface MyFunctionalInterface {
    void myMethod(String name);
}

public class LambdaExample {
    public static void main(String[] args) {
        // Using lambda expression
```

```
        MyFunctionalInterface myFunc = (name) ->
System.out.println("Hello, " + name);

        // Call the method using lambda
        myFunc.myMethod("John");  // Output: Hello, John
    }
}
```

- **Lambda Expression Breakdown**:
    - **(name)**: The parameter passed to the method.
    - **->**: The lambda operator.
    - **`System.out.println("Hello, " + name);`**: The body of the lambda expression, which is executed when `myMethod` is called.

In this example, instead of creating a class that implements `MyFunctionalInterface` and then overriding `myMethod()`, we use a lambda to define the implementation inline.

Using Lambda with Functional Interfaces

You can pass a lambda expression to any method that expects a functional interface, which makes the code more readable and concise.

Example: Using Lambda with `Consumer` Functional Interface:

```
import java.util.function.Consumer;

public class LambdaWithFunctionalInterface {
    public static void main(String[] args) {
                        // Consumer functional interface with lambda

        Consumer<String> greet = (name) ->
System.out.println("Hello, " + name);

        // Pass lambda to method
        greet.accept("Alice");  // Output: Hello, Alice
    }
}
```

- **`Consumer<String>`** is a functional interface that has a single abstract method `accept(T t)`.

- The lambda expression `(name) -> System.out.println("Hello, " + name)` provides the implementation for the `accept` method.

Example: Using Lambda with `Predicate` Functional Interface:

```
import java.util.function.Predicate;
public class LambdaWithPredicate {
    public static void main(String[] args) {
        // Predicate functional interface with lambda
        Predicate<String> isNotEmpty = (str) -> !str.isEmpty();

        // Pass lambda to method
        System.out.println(isNotEmpty.test("Hello"));  // Output:
true
        System.out.println(isNotEmpty.test(""));       // Output:
false
    }
}
```

- **`Predicate<T>`** is a functional interface used to evaluate a condition. Its method `test(T t)` returns a boolean.

- The lambda expression `(str) -> !str.isEmpty()` checks whether the string is not empty.

Benefits of Functional Interfaces and Lambdas

1. **Concise Code**: Lambdas provide a more concise and readable way to represent instances of functional interfaces.

2. **Improved Maintainability**: You don't need to write boilerplate code (like creating anonymous classes) for simple implementations.

3. **Enhanced Flexibility**: Functional interfaces can be used in many places like Java Streams, event handling, and callbacks.

4. **Parallel Programming**: Lambdas make it easier to write parallel or concurrent code using the Streams API, such as `map`, `filter`, and `reduce`.

Lambda Expression Variations

No Parameters:

```
() -> System.out.println("Hello World!");
```

1.

One Parameter (optional parentheses for single parameter):

```
name -> System.out.println("Hello " + name);
```

2.

Multiple Parameters:

```
(a, b) -> a + b;
```

3.

Lambda with a Block of Code: If the body of the lambda has more than one statement, you need to wrap it in curly braces `{}` and use a `return` statement if needed.

java
CopyEdit

```java
(a, b) -> {
    int sum = a + b;
    return sum;
}
```

4.

Conclusion

- **Functional Interface**: A functional interface in Java has exactly one abstract method and can be used with lambda expressions or method references.
- **Lambda Expression**: A lambda expression provides a clear and concise way to express instances of functional interfaces.
- **Usage**: Lambdas and functional interfaces are heavily used in Java's **Streams API**, for event handling, and for many other places where you need behavior passed as

an argument.

By combining both, you can write more concise and readable code, which is a key feature introduced in Java

Functional Interface:

java
CopyEdit
```
@FunctionalInterface
public interface Greeting {
    void greet(String name);
}
```

Lambda Expression:

java
CopyEdit
```
public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression implementing the 'greet' method
        Greeting greeting = (name) -> System.out.println("Hello, " +
name);
        greeting.greet("Alice");  // Output: Hello, Alice
    }
}
```

Explanation:

- `Greeting` is a functional interface with one abstract method: `void greet(String name)`.
- The lambda `(name) -> System.out.println("Hello, " + name)` provides the implementation of the `greet` method.

---

2. Functional Interface with Return Type

In this example, we will create a functional interface with a return type, and use a lambda expression to return a value.

Functional Interface:

```
@FunctionalInterface
```

```
public interface Adder {
    int add(int a, int b);
}
```

Lambda Expression:

```
public class LambdaExample {
    public static void main(String[] args) {
        // Lambda expression implementing the 'add' method
        Adder adder = (a, b) -> a + b;
        System.out.println(adder.add(5, 3));  // Output: 8
    }
}
```

Explanation:

- `Adder` is a functional interface with one abstract method `int add(int a, int b)`.
- The lambda `(a, b) -> a + b` implements the `add` method, adding two integers and returning the result.