Lambda Expressions in Java

Example: Using Lambda Expressions with a Functional Interface

Suppose you have a functional interface `Greeting` with a single abstract method `sayHello()`.

```
@FunctionalInterface
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        // Traditional way using an anonymous class
        Greeting traditionalGreeting = new Greeting() {
            @Override
            public void sayHello() {
                System.out.println("Hello, World!");
            }
        };
        traditionalGreeting.sayHello(); // Output: Hello, Wor
ld!

        // Using a Lambda Expression
        Greeting lambdaGreeting = () -> System.out.println("H
ello, World!");
        lambdaGreeting.sayHello(); // Output: Hello, World!
    }
}
```

Explanation:

1. **Functional Interface**:

- `Greeting` is a functional interface because it has only one abstract method, `sayHello()`.

2. **Traditional Approach**:
   - In the traditional approach, an anonymous class is used to implement the `Greeting` interface. This involves more boilerplate code.
3. **Lambda Expression**:
   - The Lambda Expression `() -> System.out.println("Hello, World!")` is a more concise way to implement the `Greeting` interface.
   - `()` represents the empty parameter list of the `sayHello()` method.
   - `>` is the lambda arrow operator.
   - `System.out.println("Hello, World!")` is the implementation of the `sayHello()` method.

Key Points:

- **Lambda Expression**: A way to implement a functional interface in a more concise form.
- **Functional Interface**: An interface with only one abstract method, which makes it eligible for use with lambda expressions.
- **Benefits**: Reduces boilerplate code, making the code more readable and maintainable.