फंक्शनल इंटरफेसेज इन जावा

`java.util.function` पैकेज □□□□ में एक ऐसा संग्रह प्रदान करता है जो फंक्शनल इंटरफेसों का समर्थन करने के लिए डिजाइन किया गया है, जो □□□□ 8 में फंक्शनल प्रोग्रामिंग के साथ पेश किया गया था। इन इंटरफेसों में प्रत्येक एक एकल एब्स्ट्रैक्ट मेथड होता है, जिससे वे लैम्ब्डा एक्सप्रेशन और मेथड रेफरेंस के साथ संगत होते हैं। इस जवाब में, इस पैकेज में सबसे आम फंक्शनल इंटरफेसों—`Function<T, R>`, `Predicate<T>`, `Consumer<T>`, और `Supplier<T>`—का उपयोग करने के बारे में समझाया गया है, साथ ही साथ व्यावहारिक उदाहरण भी दिए गए हैं।

फंक्शनल इंटरफेस क्या हैं?

एक फंक्शनल इंटरफेस एक ऐसा इंटरफेस होता है जिसमें एकल एब्स्ट्रैक्ट मेथड होता है। `java.util.function` पैकेज आम कार्यों के लिए पूर्व-निर्धारित फंक्शनल इंटरफेस प्रदान करता है, इसलिए आपको खुद के इंटरफेस बनाने की आवश्यकता नहीं होती। इन इंटरफेसों का उपयोग लैम्ब्डा एक्सप्रेशन, मेथड रेफरेंस, और स्ट्रीम □□□ के साथ किया जाता है, जिससे संक्षिप्त और अभिव्यक्तिपूर्ण कोड लिखने में मदद मिलती है।

यहाँ मुख्य इंटरफेसों का उपयोग करने का तरीका है:

---

1. `Function<T, R>`: इनपुट को आउटपुट में परिवर्तित करना

`Function<T, R>` इंटरफेस एक ऐसा फंक्शन का प्रतिनिधित्व करता है जो एक इनपुट `T` लेता है और एक आउटपुट `R` उत्पन्न करता है। इसका एब्स्ट्रैक्ट मेथड `apply` है।

उदाहरण: एक स्ट्रिंग की लंबाई प्राप्त करना

```java
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();
        System.out.println(stringLength.apply("Hello")); // Outputs: 5
    }
}
```

- □ **विवरण**: लैम्ब्डा एक्सप्रेशन `s -> s.length()` एक `Function` को परिभाषित करता है जो एक `String` (`T`) लेता है और एक `Integer` (`R`) वापस देता है। `apply` मेथड इस तर्क को कार्यान्वित करता है।

---

2. `Predicate<T>`: एक शर्त का परीक्षण करना

`Predicate<T>` इंटरफेस एक ऐसा बूलियन-वैल्यू फंक्शन का प्रतिनिधित्व करता है जो एक इनपुट `T` लेता है। इसका एब्स्ट्रैक्ट मेथड `test` है।

उदाहरण: एक संख्या की जांच करें कि वह सम है

```
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // Outputs: true
        System.out.println(isEven.test(5)); // Outputs: false
    }
}
```

 विवरण: लैम्ब्डा `n -> n % 2 == 0` एक `Predicate` को परिभाषित करता है जो `true` वापस देता है अगर इनपुट सम है। `test` मेथड इस शर्त का मूल्यांकन करता है।

---

3. `Consumer<T>`: एक क्रिया का कार्यान्वयन करना

`Consumer<T>` इंटरफेस एक ऐसा ऑपरेशन का प्रतिनिधित्व करता है जो एक इनपुट `T` लेता है और कोई परिणाम नहीं देता। इसका एब्स्ट्रैक्ट मेथड `accept` है।

उदाहरण: एक स्ट्रिंग को प्रिंट करना

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

public class Main {
    public static void main(String[] args) {
        Consumer<String> printer = s -> System.out.println(s);
        printer.accept("Hello, World!"); // Outputs: Hello, World!
    }
}
```

 विवरण: लैम्ब्डा `s -> System.out.println(s)` एक `Consumer` को परिभाषित करता है जो अपने इनपुट को प्रिंट करता है। `accept` मेथड क्रिया को कार्यान्वित करता है।

---

4. `Supplier<T>`: एक परिणाम का उत्पादन करना

`Supplier<T>` इंटरफेस एक ऐसा परिणाम का उत्पादक है जो कोई इनपुट नहीं लेता और एक `T` प्रकार का मान वापस देता है। इसका एब्स्ट्रैक्ट मेथड `get` है।

उदाहरण: एक रैंडम संख्या उत्पन्न करना

```
import java.util.function.Supplier;
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        Supplier<Integer> randomInt = () -> new Random().nextInt(100);
        System.out.println(randomInt.get()); // Outputs a random integer between 0 and 99
    }
}
```

 विवरण: लैम्ब्डा `() -> new Random().nextInt(100)` एक `Supplier` को परिभाषित करता है जो एक रैंडम इंटीजर उत्पन्न करता है। `get` मेथड मान को प्राप्त करता है।

स्ट्रीम के साथ फंक्शनल इंटरफेस का उपयोग

इन इंटरफेस □□□□ स्ट्रीम □□□ में चमकते हैं, जहां वे संक्षिप्त डेटा प्रोसेसिंग को सक्षम बनाते हैं। यहाँ एक उदाहरण है जो एक स्ट्रिंग की सूची को फिल्टर, परिवर्तित, और प्रिंट करता है:

उदाहरण: एक स्ट्रिंग की सूची को प्रोसेस करना

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("a", "bb", "ccc", "dddd");

        Predicate<String> longerThanTwo = s -> s.length() > 2;       //         2
        Function<String, String> toUpperCase = s -> s.toUpperCase(); //
```

```
        Consumer<String> printer = s -> System.out.println(s);       //

        strings.stream()
                .filter(longerThanTwo)    // "ccc"    "dddd"
                .map(toUpperCase)         // "CCC"    "DDDD"
                .forEach(printer);        // Outputs: CCC, DDDD (  -        )
    }
}
```

 विवरण:

- `filter` एक `Predicate` का उपयोग करता है लंबाई से अधिक 2 स्ट्रिंग रखने के लिए।
- `map` एक `Function` का उपयोग करता है स्ट्रिंग को अपरकेस में परिवर्तित करने के लिए।
- `forEach` एक `Consumer` का उपयोग करता है प्रत्येक परिणाम को प्रिंट करने के लिए।

मेथड रेफरेंस का उपयोग आप इसे मेथड रेफरेंस के साथ और भी संक्षिप्त बना सकते हैं:

```
strings.stream()
        .filter(s -> s.length() > 2)
        .map(String::toUpperCase)      // Function
        .forEach(System.out::println); // Consumer
```

फंक्शनल इंटरफेस का संयोजन

कुछ इंटरफेस जटिल ऑपरेशनों के लिए संयोजन को सक्षम बनाते हैं: - **फंक्शन संयोजन**: `andThen` या `compose` का उपयोग करें। `java Function<String, Integer> toLength = s -> s.length();  Function<Integer, String> toString = i -> "Length is " + i;  Function<String, String> combined = toLength.andThen(toString);  System.out.println(combi // Outputs: Length is 5 - □□□□□□□□□□ **संयोजन**: `and`, `or`, या `negate` का उपयोग करें। `java Predicate<String> isLong = s -> s.length() > 5; Predicate<String> startsWithA = s -> s.startsWith("A"); Predicate<String> isLongAndStartsWithA = isLong.and(startsWithA); System.out.println(isLongAndStartsWithA.test("Avocado")); // Outputs: true`

सारांश

यहाँ इन इंटरफेसों का उपयोग करने का समय और तरीका है: - `Function<T, R>`: एक इनपुट को एक आउटपुट में परिवर्तित करें (उदाहरण के लिए, `apply`). - `Predicate<T>`: एक शर्त का परीक्षण करें (उदाहरण के लिए, `test`). - `Consumer<T>`: एक इनपुट पर एक क्रिया का कार्यान्वयन करें

(उदाहरण के लिए, `accept`). - `Supplier<T>`: कोई इनपुट के बिना एक मान उत्पन्न करें (उदाहरण के लिए, `get`). - **स्ट्रीम के साथ**: उन्हें शक्तिशाली डेटा प्रोसेसिंग के लिए संयोजित करें। - **संयोजन**: उन्हें जटिल तर्क के लिए चेन या संयोजित करें।

आप इन इंटरफेसों को लैम्ब्डा एक्सप्रेशन (उदाहरण के लिए, `s -> s.length()`) या मेथड रेफरेंस (उदाहरण के लिए, `String::toUpperCase`) का उपयोग करके लागू कर सकते हैं। वे □□□□ में एक फंक्शनल प्रोग्रामिंग शैली को सक्षम बनाते हैं, जिससे आपका कोड, विशेष रूप से स्ट्रीम □□□ के साथ, अधिक संक्षिप्त, पढ़ने योग्य और पुन: उपयोग योग्य हो जाता है।