Functional Interfaces in Java

This document provides an overview of several key functional interfaces in Java, including **Function**, **Predicate**, **Consumer**, and **Supplier**. Each interface is defined with its abstract methods, default methods, and static methods where applicable. Examples are provided to illustrate how these interfaces can be utilized in Java programming.

Function

The **Function** interface represents a function that takes an argument of type **T** and produces a result of type **R**.

Abstract Method:

```
R apply(T t);
```

Default Methods:

- **andThen**: Returns a composed function that first applies this function to its input, and then applies the **after** function to the result.

```
<V> Function<T, V> andThen(Function<? super R, ? extends V> after);
```

- **compose**: Returns a composed function that first applies the **before** function to its input, and then applies this function to the result.

```
<V> Function<V, R> compose(Function<? super V, ? extends T> before);
```

Static Method:

- **identity**: Returns a function that always returns its input argument.

```
static <T> Function<T, T> identity();
```

Example:

```
Function<Integer, String> intToString = Object::toString;
String result = intToString.apply(5); // result is "5"
```

Predicate

The **Predicate** interface represents a single argument function that returns a boolean value.

Abstract Method:

```
boolean test(T t);
```

Default Methods:

- **and**: Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

```
Predicate<T> and(Predicate<? super T> other);
```

- **or**: Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

```
Predicate<T> or(Predicate<? super T> other);
```

- **negate**: Returns a predicate that represents the logical negation of this predicate.

```
Predicate<T> negate();
```

Static Method:

- **isEqual**: Returns a predicate that tests if two arguments are equal.

```
static <T> Predicate<T> isEqual(Object targetRef);
```

Example:

```
Predicate<String> isNotEmpty = str -> !str.isEmpty();
boolean result = isNotEmpty.test("Hello"); // result is true
```

Consumer

The **Consumer** interface represents an operation that accepts a single input argument and returns no result.

Abstract Method:

```
void accept(T t);
```

Default Method:

- **andThen**: Returns a composed **Consumer** that performs, in sequence, this operation followed by the **after** operation.

```
Consumer<T> andThen(Consumer<? super T> after);
```

Example:

```
Consumer<String> print = System.out::println;
print.accept("Hello, World!"); // prints "Hello, World!"
```

Supplier

The **Supplier** interface represents a supplier of results. It does not take any arguments and returns a result.

Abstract Method:

```
T get();
```

Example:

```
Supplier<String> stringSupplier = () -> "Hello, Supplier!";
String result = stringSupplier.get(); // result is "Hello, Supplier!"
```

Conclusion

Understanding these functional interfaces is crucial for effective programming in Java, especially when working with streams and lambda expressions. Each interface serves a unique purpose and can be combined to create powerful and expressive code.