`Consumers` are functional interfaces that take in a parameter and do not produce anything.

Below are some of the functional interfaces which can be categorized as Consumers.

`Consumer<T>`	Represents an operation that accepts a single (reference type) input argument and returns no result	`void accept(T t)`
`DoubleConsumer`	Accepts a single double-value argument and returns no result	`void accept(double value)`
`IntConsumer`	Accepts a single int-value argument and returns no result	`void accept(int value)`
`LongConsumer`	Accepts a single long-value argument and returns no result	`void accept(long value)`
`BiConsumer<T, U>`	Represents an operation that accepts two (reference type) input arguments and returns no result	`void accept(T t, U u)`
`ObjDoubleConsumer<T>`	Accepts an object-value and a double-value argument, and returns no result	`void accept(T t, double value)`
`ObjIntConsumer<T>`	Accepts an object-value and an int-value argument, and returns no result	`void accept(T t, int value)`
`ObjLongConsumer<T>`	Accepts an object-value and a long-value argument, and returns no result	`void accept(T t, long value)`

`Consumer<T>`#

This interface takes a parameter of type `T` and does not return anything.

A consumer can be used in all contexts where an object needs to be consumed,i.e. taken as input, and some operation is performed on the object without returning any result.

Below is the list of methods in the `Consumer` interface. `Consumer<T>` has an abstract method `accept()` and a default method called `andThen()`, which is used for chaining.

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
void	**accept**(T t) Performs this operation on the given argument.
default Consumer<T>	**andThen**(Consumer<? super T> after) Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

In the below example, we will crate a `Consumer` which prints a value.

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

public class ConsumerDemo {

public static void main(String[] args) {

        Consumer<String> stringConsumer = s -> System.out.println(s);
        stringConsumer.accept("Hello World.");

        Consumer<Integer> intConsumer = i -> System.out.println("Integer value = " + i);
        intConsumer.accept(5);
    }
}
```

Run | Save | Reset

The `andThen()` method, which is a default method in the `Consumer` interface is used for chaining. Here is the syntax of this method.

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

The `andThen()` method returns a composed `Consumer` that performs this operation followed by the **after** operation. In the below example, we will create two consumers, and we will chain them together using the `andThen()` method.

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

public class ConsumerDemo {

    public static void main(String[] args) {

        Consumer<String> consumer1 = (arg) -> System.out.println(arg + "My name is Jane.");

        Consumer<String> consumer2 = (arg) -> System.out.println(arg + "I am from Canada.");

        consumer1.andThen(consumer2).accept("Hello. ");

    }
}
```

Run | Save | Reset

`BiConsumer<T,U>`#

This interface takes two parameters and returns nothing.

- T - the type of the first argument to the operation
- U - the type of the second argument to the operation.

This interface has the same methods as present in the `Consumer<T>` interface.

```
import java.util.function.BiConsumer;

public class BiConsumerDemo {

    public static void main(String[] args) {

      BiConsumer<String, String> greet = (s1, s2) -> System.out.println(s1 + s2);
      greet.accept("Hello", "World");

    }
}
```

Run | Save | Reset