CitiusTech

Lambda Built-in Functional Interfaces

Session 2

Version 1.0

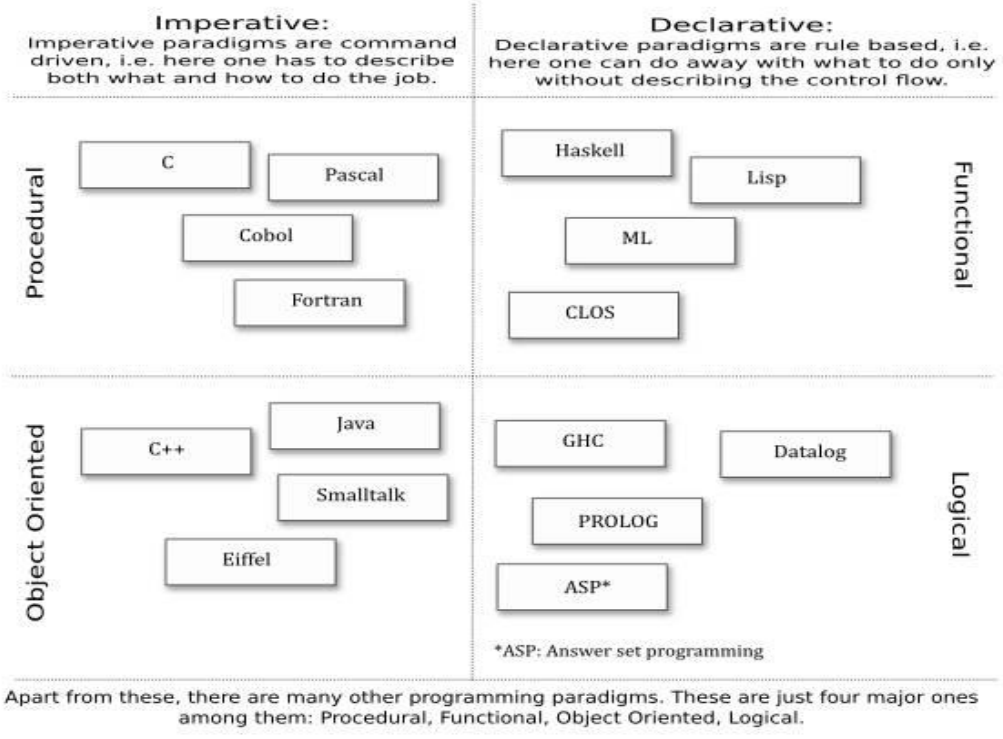
January, 2019

Agenda

- **Functional Interfaces in Java 8**
- Types of Interfaces
- Primitive Versions of Functional Interfaces
- Binary Versions of Functional Interfaces
- BinaryOperator

Functional Interfaces in Java 8 (1/3)

- **What Is Functional Programming in Java?**
  - Functional programming emphasizes on declarative aspects of programming where business logic is composed of pure functions, an idea that somewhat contrasts the essence of object-oriented methodology



Ref: http://prikid.biz/what-is-a-template-in-java.html

Functional Interfaces in Java 8 (2/3)

- The **java.util.function** has numerous built-in functional interfaces.
- **Built-in Functional Interfaces**:
  - Declares a single abstract method
  - Can have any number of default or static methods
  - Are useful for creating lambda expressions
- Figure shows some of the key functional interfaces from the **java.util.function** package and their abstract method.

Functional Interfaces	Method
Predicate<T>	Boolean test (T t)
Consumer<T>	void accept (T T)
Function<T, R>	R apply (T t)
Supplier<T>	T get()

Functional Interfaces in Java 8 (3/3)

- Table discuss four important built-in interfaces in the **java.util.function** package: Predicate, Consumer, Function, and Supplier.

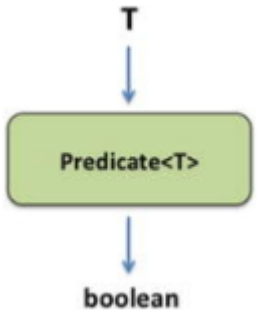
Functional Interfaces	Description	Common Use
Predicate<T>	▪ Checks a condition and returns a boolean value as result	▪ In filter() method in java.util.stream. Stream which is used to remove elements in the stream that don't match the given condition (i.e., predicate) as argument.
Consumer<T>	▪ Operation that takes an argument but returns nothing	▪ In forEach() method in collections and in java.util.stream. This method is used for traversing all the elements in the collection or stream.
Function<T>	▪ Functions that take an argument and return a result	▪ In map() method in java.util.stream. Stream to transform or operate on the passed value and return a result.
Supplier<T>	▪ Operation that returns a value to the caller	▪ In generate() method in java.util.stream. Stream to create an infinite stream of elements.

Agenda

- Functional Interfaces in Java 8
- **Types of Interfaces**
- Primitive Versions of Functional Interfaces
- Binary Versions of Functional Interfaces
- BinaryOperator

Predicate Interface

- We often need to use functions that check a condition and return a boolean value.
- In this case, we can use the **Predicate** functional interface.
- Figure shows the abstract named **test()** that takes an argument and returns **true** or **false**.

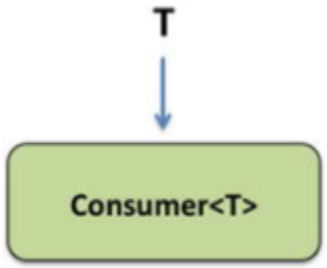



```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
   // other methods elided
}
```

- The **Predicate** functional interface also defines default methods namely, **and()** and **or()** that takes a **Predicate** and returns a **Predicate**.
- It also has a **negate()** method whose behavior is similar to the **!** Operator.

Consumer Interface (1/2)

- There are many methods that take one argument, perform some operations based on the argument but do not return anything to their callers–they are **consumer** methods.
- Figure shows the declaration of abstract method named **accept()** defined in the **Consumer** interface.




```
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
      // the default andThen method elided
}
```

- The **accept()** method "consumes" an object and returns nothing (**void**).
- The following code snippet demonstrates the use of the **Consumer** interface:

```
Consumer<String> printUpperCase = str -> System.out.println(str.toUpperCase());

printUpperCase.accept("hello");
// prints: HELLO
```

- In this code, the lambda expression takes the given string, converts to upper case, and prints it to the console. We are passing the actual argument "hello" to the accept() method.

Consumer Interface (2/2)

- Consumer interface also has a default method named **andThen()**; it allows chaining calls to **Consumer** objects.
- Consumer **andThen()** returns a composed Consumer that performs, in sequence, for the current operation followed by the after operation.
- The syntax of the andThen() method are as follows:

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

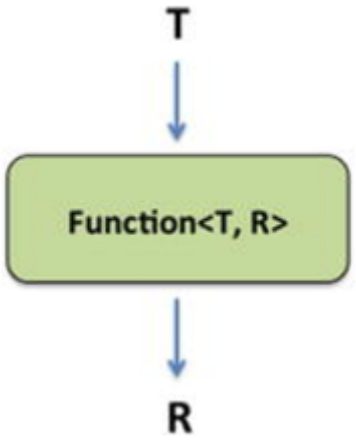
- Following code snippet shows the use of andThen():

```
Consumer<String> c = (x) -> System.out.println(x.toLowerCase());
c.andThen(c).accept("java8");

//prints:
java8
java8
```

Function Interface

- The **Function** interface defines a single abstract method named **apply()** that takes an argument of generic type **T** and returns an object of generic type **R**.
- Figure shows the abstract method named **apply()** in the **Function** interface.




```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);
   // other methods elided
}
```

-
```
Function<String, Integer> strLength = str -> str.length();
System.out.println(strLength.apply("supercalifragilisticexpialidocious"));
// prints: 34
```

- This code takes a string and returns its length, as a result of making the call to **apply()** method.
- Advantages of Function interface:
  - In all scenarios where an object of a particular type is the input, an operation is performed on it and object of another type is returned as output

Function Interface: Helper Methods

- The Function interface contains the following default and static methods:

```
default <V> Function<T, V> andThen(Function<? super R,? extends V> after) default <V>
Function<V, R> compose(Function<? super V,? extends T> before) static <T> Function<T, T>
identity()
```

- **andThen()** creates a Function that calls the current function and specified function after to get the result.
- **compose()** creates a Function that calls the specified function then current function and returns the result.
- **identify()** creates a function that returns its argument.

Function Interface: Specialization

- There are six specialization of the Function<T, R> interface:
  - IntFunction<R>
  - LongFunction<R>
  - DoubleFunction<R>
  - ToIntFunction<T>
  - ToLongFunction<T>
  - ToDoubleFunction<T>
- **IntFunction<R>**, **LongFunction<R>**, and **DoubleFunction<R>** take an int, a long, and a double as an argument, respectively, and their return value is in type R.
- **ToIntFunction<T>**, **ToLongFunction<T>**, and **ToDoubleFunction<T>** take an argument of type T and return an int, a long, and a double, respectively.

Function Interface: IntFunction

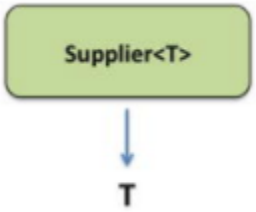
- **IntFunction** represents a function that accepts an int-valued argument and produces a result.
- This is the int-consuming primitive specialization for Function.
- Following code snippet shows how to use IntFunction:

```
IntFunction<String> i = (x)->Integer.toString(x);

System.out.println(i.apply(3).length());
```

Supplier Interface (1/2)

- There are method that does not take any input but returns some output, they are supplier methods.
- Figure shows the abstract method in the **Supplier** interface.




```
@FunctionalInterface
public interface Supplier<T> {
   T get();
   // no other methods in this interface
}
```

- Following code snippet returns a value without taking anything as input:

```
Supplier<String> currentDateTime =  ()  ->  LocalDateTime.now().toString();

System.out.println(currentDateTime.get());
```

- The lambda expression does not take any input but returns the current date/time as a **String** format.

Supplier Interface (2/2)

- Following code snippet makes use of constructor references:

```
Supplier<String> newString = () -> new String();
System.out.println(newString.get());
```

- This lambda expression returns a new **String** object.

Agenda

- Functional Interfaces in Java 8
- Types of Interfaces
- **Primitive Versions of Functional Interfaces**
- Binary Versions of Functional Interfaces
- BinaryOperator

Primitive Versions of Functional Interfaces (1/6)

- The built-in interfaces **Predicate**, **Consumer**, **Function**, and **Supplier** operate on reference type objects.
- For primitive types there are specializations available for **int**, **long** and **double** types for these functional interfaces.
- For example,
  - Consider **Predicate** that operates on objects of type T, i.e., it is **Predicate**<T>.
  - The specializations for int, long, and double for Predicate are **IntPredicate**, **LongPredicate**, and **DoublePredicate** respectively.
- When you try to use primitive types with these functional interfaces, it results in implicit autoboxing and unboxing.
- For example, an **int** value gets converted to an **Integer** object and vice versa.

Primitive Versions of Predicate Interface (2/6)

- Table shows the primitive versions of Predicate interface.

Functional Interface	Abstract Method	Description
IntPredicate	▪ boolean test(int value)	▪ Evaluates the condition passed as int and returns a boolean value as result
LongPredicate	▪ boolean test(long value)	▪ Evaluates the condition passed as long and returns a boolean value as result
DoublePredicate	▪ boolean test(double value)	▪ Evaluates the condition passed as double and returns a boolean value as result

Primitive Versions of Function Interface (3/6)

- Table shows the primitive versions of Function interface.

Functional Interface	Abstract Method	Description
IntFunction<R>	▪ R apply(int value)	▪ Operates on the passed int argument and returns value of generic type R
LongFunction<R>	▪ R apply(long value)	▪ Operates on the passed long argument and returns value of generic type R
DoubleFunction<R>	▪ R apply(double value)	▪ Operates on the passed double argument and returns value of generic type R
ToIntFunction<T>	▪ int applyAsInt(T value)	▪ Operates on the passed generic type argument T and returns an int value
ToLongFunction<T>	▪ long applyAsLong(T value)	▪ Operates on the passed generic type argument T and returns a long value

Primitive Versions of Function Interface (4/6)

- Table shows the primitive versions of Function interface.

Functional Interface	Abstract Method	Description
ToDoubleFunction<T>	▪ double applyAsDouble(T value)	▪ Operates on the passed generic type argument T and returns an double value
IntToLongFunction	▪ long applyAsLong(int value)	▪ Operates on the passed int type argument and returns a long value
IntToDoubleFunction	▪ double applyAsDouble(int value)	▪ Operates on the passed int type argument and returns a double value
LongToIntFunction	▪ int applyAsInt(long value)	▪ Operates on the passed long type argument and returns an int value
LongToDoubleFunction	▪ double applyAsLong(long value)	▪ Operates on the passed long type argument and returns a doublevalue
DoubleToIntFunction	▪ int applyAsInt(double value)	▪ Operates on the passed double type argument and returns an intvalue
DoubleToLongFunction	▪ long applyAsLong(double value)	▪ Operates on the passed double type argument and returns a longvalue

Primitive Versions of Consumer Interface (5/6)

- Table shows the primitive versions of Consumer interface.

Functional Interface	Abstract Method	Description
IntConsumer	▪ void accept(int value)	▪ Operates on the given int argument and returns nothing
LongConsumer	▪ void accept(long value)	▪ Operates on the given long argument and returns nothing
DoubleConsumer	▪ void accept(double value)	▪ Operates on the given double argument and returns nothing
ObjIntConsumer<T>	▪ void accept(T t, int value)	▪ Operates on the given generic type argument T and int arguments and returns nothing
ObjLongConsumer<T>	▪ void accept(T t, long value)	▪ Operates on the given generic type argument T and long arguments and returns nothing
ObjDoubleConsumer<T>	▪ void accept(T t, double value)	▪ Operates on the given generic type argument T and double arguments and returns nothing

Primitive Versions of Supplier Interface (6/6)

- Table shows the primitive versions of Supplier interface.

Functional Interface	Abstract Method	Description
BooleanSupplier	▪ boolean getAsBoolean()	▪ Takes no arguments and returns a boolean value
IntSupplier	▪ int getAsInt()	▪ Takes no arguments and returns an int value
LongSupplier	▪ long getAsLong()	▪ Takes no arguments and returns a long value
DoubleSupplier	▪ double getAsDouble()	▪ Takes no arguments and returns a double value

Agenda

- Functional Interfaces in Java 8
- Types of Interfaces
- Primitive Versions of Functional Interfaces
- **Binary Versions of Functional Interfaces**
- BinaryOperator

Binary Versions of Functional Interfaces

- The functional interfaces namely, Predicate, Consumer, and Function have one abstract method in them.
- However, these interfaces also have binary versions.
- The prefix “Bi” indicates the version that takes “two” arguments.
- There are **BiFunction**, **BIPredicate**, **BiConsumer** for Function, Predicate and **BiConsumer** for Consumer that takes two arguments.

Since the abstract method in **Supplier** does not take any argument, there is no equivalent **BiSupplier** available.

- Table shows the binary versions of functional interfaces.

Functional Interface	Abstract Method	Description
BiPredicate<T, U>	▪ boolean test(T t, U u)	▪ Checks if the arguments match the condition and returns a boolean value as result
BiConsumer<T, U>	▪ void accept(T t, U u)	▪ Operation that consumes two arguments but returns nothing
BiFunction<T, U, R)	▪ R apply(T t, U u)	▪ Function that takes two argument and returns a result

BiFunction and BiPredicate Interface

- Following code snippet demonstrates the use of BIFunction interface:

```
BiFunction<String, String, String> concatStr = (x, y) -> x + y;
System.out.println(concatStr.apply("hello ", "world"));
// prints: hello world
```

- Here, both the arguments are of same type, that is **String**. However, they can be of different types.
- Following code snippet demonstrates the use of **BiPredicate** interface:

```
BiPredicate<Integer, Integer> bi = (x, y) -> x > y;
System.out.println(bi.test(2, 3));
// prints: false
```

- **BiPredicate** represents a predicate which is a boolean-valued function of two arguments.

BiConsumer andThen

- BiConsumer andThen returns a composed BiConsumer that performs, in sequence, this operation followed by the after operation.
- **Syntax**:

```
default BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after)
```

- Following code snippet shows how to use **andThen**:

```
BiConsumer<String, String> biConsumer = (x, y) ->
   {
          System.out.println(x);
          System.out.println(y);
   };
biConsumer.andThen(biConsumer).accept("Java 8", " Lambdas");
// prints: Java 8
           Lambdas
           Java 8
           Lambdas
```

Agenda

- Functional Interfaces in Java 8
- Types of Interfaces
- Primitive Versions of Functional Interfaces
- Binary Versions of Functional Interfaces
- **BinaryOperator**

BinaryOperator (1/2)

- **BinaryOperator** represents an operation upon two operands of the same type, producing a result of the same type.
- This is a specialization of **BiFunction** for the case where the operands and the result are all of the same type.
- This is a functional interface whose functional method is:
  - BiFunction.apply(Object, Object)
- Table shows the static methods of BinaryOperator.

Method	Description
static <T> BinaryOperator <T> maxBy(Comparator<? Super T> comparator)	▪ Returns a BinaryOperator which returns the greater of two elements according to the specified Comparator.
static <T> BinaryOperator <T> minBy(Comparator<? Super T> comparator)	▪ Returns a BinaryOperator which returns the lesser of two elements according to the specified Comparator.

BinaryOperator (2/2)

- Following code snippet shows how to use BinaryOperator:

```
BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;

System.out.println(adder.apply(3, 4));

//prints: 7
```

- Following code snippet shows how to use **maxBy()** method:

```
BinaryOperator<Integer> bi = BinaryOperator.maxBy(Comparator.reverseOrder());
System.out.println(bi.apply(2, 3));
//prints: 2
```

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email ct-univerct@citiustech.com

www.citiustech.com