Functional Interface, Lambda Expression, and Functional Programming

Functional Interface and Lambda expression are introduced in Java 8 to support Functional Programming.

In general, a functional interface has a single functionality to exhibit.

- The interface only contains 1 abstract method.
- In Java 7 and earlier versions, an interface can only have abstract methods.
- In Java 8 an interface may contain non-abstract methods
  - static and default methods are fully implemented.
    - Remark: A class that implements an interface does not inherit static and default methods in the interface
  - public methods inherited from the class `Object`.
- In Java 9 an interface may have private methods

Example:

```
public class Trade
{
   int quantity;
   String status; // "NEW", "COMPLETED", "CANCELLED"
   ...

   public int getQuantity()
   {  return quantity;
   }

   public String getStatus()
   {  return status;
   }

   ...  // other methods
}
```

Consider a functional interface `Predicate` that has a method `test`.

```
// java.util.function.Predicate
@FunctionalInterface
public interface Predicate<T>
{
   public boolean test(T t);
}
```

You can have a utility method that select trades based on a given criterion.

```
public class MyUtil
{
   public static List<Trade> filterTrades(
               List<Trade> trades, Predicate<Trade> tester)
   {
      List<Trade> list = new ArrayList();

      for (Trade t : trades)
         if (tester.test(t))
            list.add(t);

      return list;
   }
}
```

In the program that uses the `filterTrades` method, you need to provide a `Predicate<Trade>` object.

You can write your code using anonymous class.

```
List<Trade> list = new ArrayList();
...  // statements to fill up list

Predicate myPred = new Predicate<Trade>()
                   {
                      @Override
                      public boolean test(Trade t)
                      {
                         return t.getStatus().equals("NEW");
                      }
                   };

List<Trade> newTrades = MyUtil.filterTrades(list, myPred);
```

Lambda expression is introduced in Java 8.

Lambda expression is used to define an implementation of a Functional interface.

Basic syntax of Lambda expression

```
(arg1, arg2) -> single statement;

(Type arg1, Type arg2) -> { multiple statements };
```

The number of arguments depends on the abstract method defined in the interface.

Arguments are enclosed in parentheses and separated by commas.
The arguments correspond to the inputs required by the (abstract) method of the Functional interface.
The expression or statement block corresponds to the body of the method.

The data type of the arguments can be explicitly declared or it can be inferred from the context (i.e. the compiler will determine the target data type)

When there is a single argument, if its type is inferred, it is not mandatory to use parentheses, e.g. `(a) -> statement;` is the same as `a -> statement;`

Now, let's go back to the above example.
The codes can be rewritten using Lambda expression:

```
List<Trade> list = new ArrayList();
...

// Lambda expression
Predicate<Trade> myPred = t -> t.getStatus().equals("NEW");

List<Trade> newTrades = MyUtil.filterTrades(list, myPred);

// Alternative coding style using anonymous object
List<Trade> newTrades = MyUtil.filterTrades(list,
                          t -> t.getStatus().equals("NEW"));
```

Some Functional Interface declared in `java.util.function`

Interface name	Method	Description
`Function``<T,R>`	`R` **`apply`**`(T t)`	A function that takes an argument of type `T` and returns a result of type `R`. Apply a function to an input value.
`BiFunction``<T,U,R>`	`R` **`apply`**`(T t, U u)`	A function that takes 2 arguments of types `T` and `U`, and returns a result of type `R`.
`Predicate``<T>`	`boolean` **`test`**`(T t)`	A predicate is a Boolean-valued function that takes an argument and returns true or false. Test the predicate with an input value.
`BiPredicate``<T,U>`	`boolean` **`test`**`(T t, U u)`	A predicate with 2 arguments.
`Consumer``<T>`	`void` **`accept`**`(T t)`	An operation that takes an argument, operates on it to produce some side effects, and returns no result. The consumer accepts an input item.
`BiConsumer``<T,U>`	`void` **`accept`**`(T t, U u)`	An operation that takes 2 arguments, operates on them to produce some side effects, and returns no result.
`Supplier``<T>`	`T` **`get`**`()`	Represents a supplier that returns a value of type `T`. Get an item from supplier.
`UnaryOperator``<T>`	`T` **`apply`**`(T t)`	Inherits from `Function<T, T>`
`BinaryOperator``<T>`	`T` **`apply`**`(T t1, T t2)`	Inherits from `BiFunction<T, T, T>`

Example: convert centigrade to Fahrenheit

```
// Function<T, R> R apply(T t)
// T : Double
// R : Double
// argument t is called x in this example
Function<Double, Double>
   centigradeToFahrenheit = x -> (x * 9 / 5) + 32.0;

double degreeC = 36.9;
double degreeF = centigradeToFahrenheit.apply(degreeC);
```

Example: function to calculate the aggregated quantity of all trades

```
// Function<T, R> R apply(T t)
// T : List<Trade>
// R : Integer
// argument t is called trades in this example
Function<List<Trade>, Integer> aggregatedQty =
   trades -> {
      int total = 0;
      for (Trade t : trades)
         total += t.getQuantity();
      return total;
   };


List<Trade> list = new ArrayList();
...  // statements to fill up list

int totalQty = aggregatedQty.apply(list);
```

Target Typing

Consider the lambda expression

```
(x, y) -> x + y;
```

The process of inferring the data type of a lambda expression from the context is known as target typing.

```
@FunctionInterface
public interface Adder {
   double add(double n1, double n2);
}

@FunctionInterface
public interface Joiner {
   String join(String s1, String s2);
}

Adder adder = (x, y) -> x + y;
// x, y and return value are inferred to type double
// operator '+' represents the addition operation

Joiner joiner = (x, y) -> x + y;
// x, y and return value are inferred to type String
// operator '+' represents string concatenation
```

Functional interfaces are used in 2 contexts:

- Library designers that implement the APIs (e.g. Collection and Stream API)
- Library users that use the APIs

```
// FunctionUtil.java
// import statements are omitted for brevity

public class FunctionUtil {
   // Apply an action on each item in a list
   public static <T> void forEach(List<T> list,
                                  Consumer<? super T> action)
   {
      for(T item : list)
         action.accept(item);
   }

   // Apply a filter to a list, returned the filtered list items
   public static <T> List<T> filter(List<T> list,
                                    Predicate<? super T> predicate)
   {
      List<T> filteredList = new ArrayList();
      for(T item : list)
         if (predicate.test(item))
            filteredList.add(item);

      return filteredList;
   }

   // Map each item of type T in a list to a value of type R
   public static <T, R> List<R> map(List<T> list,
                                    Function<? super T, R> mapper)
   {
      List<R> mappedList = new ArrayList();
      for(T item : list)
         mappedList.add(mapper.apply(item));

      return mappedList;
   }

   // Apply an action on each item of type T in input list.
   // Transform the item to type R, and aggregate/save results
   // in a List<R>.
   public static <T, R> List<R> transform(List<T> list,
                           BiConsumer<List<R>, ? super T> action)
   {
      List<R> resultList = new ArrayList();
      for (T item : list)
         action.accept(resultList, item);

      return resultList;
   }
}
```

```java
// Person.java
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public enum Gender {
    MALE, FEMALE
}

public class Person {
   private String name;
   private LocalDate dob;  // date of birth
   private Gender gender;
   private double income;

   public Person(String name, LocalDate dob, Gender gender,
                 double salary)
   {
      this.name = name;
      this.dob = dob;
      this.gender = gender;
      this.income = salary;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public LocalDate getDob() {
      return dob;
   }

   public void setDob(LocalDate dob) {
      this.dob = dob;
   }


   public Gender getGender() {
      return gender;
   }

   public void setGender(Gender gender) {
      this.gender = gender;
   }
```

```
    public boolean isMale()
    {
       return gender == Male;
    }

    public double getIncome()
    {
       return income;
    }

    @Override
    public String toString() {
       return name + " " + ", " + gender + ", " + dob + ", "
              + income;
    }

    // A utility method to create a List<Person>.
    // For illustration purpose only.
    public static List<Person> persons() {
       ArrayList<Person> list = new ArrayList();

       list.add(new Person("John", LocalDate.of(1975, 1, 20),
                           MALE, 1000.0);

       list.add(new Person("Wally", LocalDate.of(1965, 9, 12),
                           MALE, 1500.0);

       list.add(new Person("Donna", LocalDate.of(1970, 9, 12),
                           FEMALE, 2000.0);
       return list;
    }
}
```

```
// FunctionUtilTest.java
import java.util.List;

public class FunctionUtilTest {
   public static void main(String[] args) {
      List<Person> list = Person.persons();

      // Use forEach() method to print each person in the list
      System.out.println("Original list of persons:");
      FunctionUtil.forEach(list, p -> System.out.println(p));

      // Filter only males
      List<Person> maleList = FunctionUtil.filter(list,
                              p -> p.getGender() == MALE);

      System.out.println("\nMales only:");
      FunctionUtil.forEach(maleList,
                           p -> System.out.println(p));

      // Map each person to his/her year of birth
      List<Integer> dobYearList = FunctionUtil.map(list,
                                  p -> p.getDob().getYear());

      System.out.println(
                    "\nPersons mapped to year of their birth:");
      FunctionUtil.forEach(dobYearList,
                           year -> System.out.println(year));

      // Apply an action to each person in the list
      // Add one year to each male's dob
      FunctionUtil.forEach(maleList,
                          p -> p.setDob(p.getDob().plusYears(1)));

      System.out.println(
                     "\nMales only after adding 1 year to DOB:");

      FunctionUtil.forEach(maleList, p -> System.out.println(p));
   }
}

// Remark: the method forEach is defined in ArrayList<E> and
// Vector<E>. Vector is synchronized, i.e. thread-safe.

// Method forEach is also defined in interface Iterable<E>.
// interface List<E> extends Collection<E> and Iterable<E>.
// Any class that implements the List<T> interface also
// supports the forEach method.
```

Outputs of the program:

Original list of persons:

```
John, MALE, 1975-01-20, 1000.0
Wally, MALE, 1965-09-12, 1500.0
Donna, FEMALE, 1970-09-12, 2000.0
```

Males only:

```
John, MALE, 1975-01-20, 1000.0
Wally, MALE, 1965-09-12, 1500.0
```

Persons mapped to year of their birth:

```
1975
1965
1970
```

Males only after adding 1 year to DOB:

```
John, MALE, 1976-01-20, 1000.0
Wally, MALE, 1966-09-12, 1500.0
```

Example: Find the top10 most popular video

```
// VideoRec.java
public class VideoRec
{
    private final long timestamp;
    private final String vid;
    private final String client;

    public VideoRec(long t, String v, String c)
    {
        timestamp = t;
        vid = v;
        client = c;
    }

    public long getTimestamp()
    {
        return timestamp;
    }

    public String getVid()
    {
        return vid;
    }

    public String getClient()
    {
        return client;
    }

    @Override
    public String toString()
    {
        return timestamp + "," + vid + "," + client;
    }
}
```

```java
// Pair.java
public class Pair<S, T>
{
    private S first;
    private T second;

    public Pair(S n1, T n2)
    {
        first = n1;
        second = n2;
    }

    public S getFirst()
    {
        return first;
    }

    public T getSecond()
    {
        return second;
    }

    public void setFirst(S e)
    {
        first = e;
    }

    public void setSecond(T e)
    {
        second = e;
    }

    @Override
    public String toString()
    {
        return "(" + first + ", " + second + ")";
    }
}
```

```
// Version 1 : Conventional imperative programming

String fname = "videoData.txt";
ArrayList<VideoRec> list = readDataFile(fname);

// Find the top 10 most popular videos in the log
list.sort((r1, r2)-> r1.getVid().compareTo(r2.getVid()));
// list.sort(comparing(VideoRec::getVid));

ArrayList<Pair<String, Integer>> viewCountList = new ArrayList();

int i = 0;
while (i < list.size())
{
   String curVid = list.get(i).getVid();
   int j = i + 1;
   while (j < list.size() && list.get(j).getVid().equals(curVid))
      j++;

   viewCountList.add(new Pair(curVid, j-i));
   i = j;
}

viewCountList.sort((a, b) -> b.getSecond() - a.getSecond());

int end = (viewCountList.size() >= 10) ? 10 : viewCountList.size();

System.out.println("Top 10 most popular videos:");

for (Pair<String, Integer> p : viewCountList.subList(0, end))
   System.out.println(p);
```

```
private static ArrayList<VideoRec> readDataFile(String fname)
{
   // Read in the VideoRec from data file
   ArrayList<VideoRec> list = new ArrayList();

   try (Scanner sc = new Scanner(new File(fname)))
   {
      while (sc.hasNextLine())
      {
         String line = sc.nextLine();
         String[] token = line.split(",");
         list.add(new VideoRec(Long.parseLong(token[0]),
                                              token[1], token[2]));
      }
   }
   catch(FileNotFoundException e)
   {  }
   return list;
}
```

```
// Version 2 : Functional programming using FunctionUtil class.

String fname = "videoData.txt";
ArrayList<VideoRec> list = readDataFile(fname);

list.sort((r1, r2)-> r1.getVid().compareTo(r2.getVid()));

BiConsumer<List<Pair<String, Integer>>, VideoRec> action =
   (result, v) -> {
      if (result.isEmpty())
         result.add(new Pair(v.getVid(), 1));
      else
      {
         Pair<String, Integer> item = result.get(result.size()-1);

         if (item.getFirst().equals(v.getVid()))
            item.setSecond(item.getSecond() + 1);
         else
            result.add(new Pair(v.getVid(), 1));
      }
  };

List<Pair<String, Integer>>
             viewCountList = FunctionUtil.transform(list, action);

viewCountList.sort((a, b) -> b.getSecond() - a.getSecond());

int end = (viewCountList.size() >= 10) ? 10 : viewCountList.size();

System.out.println("Top 10 most popular videos:");

for (Pair<String, Integer> p : viewCountList.subList(0, end))
   System.out.println(p);
```

Method References

A lambda expression represents an anonymous function that is treated as an instance of a functional interface.

A method reference is a shorthand to create a lambda expression using an existing method.

If a lambda expression contains a body that is an expression using a method call, you can use a method reference in place of that lambda expression.

Types of method references (: : “4 dots”)

Syntax	Description
`TypeName::staticMethod`	A method reference to a static method of a class, an interface, or an enum
`objectRef::instanceMethod`	A method reference to an instance of the specified object
`ClassName::instanceMethod`	A method reference to an instance method of an arbitrary object of the specified class
`TypeName.super::instanceMethod`	A method reference to an instance method of the supertype of a particular object
`ClassName::new`	A constructor reference to the constructor of the specified class
`ArrayTypeName::new`	An array constructor reference to the constructor of the specified array type

Example statements in `findTop10Video()`

```
List<Pair<String, Integer>> top10 = findTop10Video(list);

for (Pair<String, Integer> p : top10)
    System.out.println(p);

// Replace the above for-loop using the forEach method

// top10.forEach(p -> System.out.println(p));

// top10.forEach(System.out::println);
```

More Examples

```
ToIntFunction<String> lenFunction = str -> str.length();

Supplier<Item> func1 = () -> new Item();

Function<String, Item> func2 = str -> new Item(str);

BiFunction<String, Double, Item>
            func3 = (name, price) -> new Item(name, price);
```

The above lambda expressions can be rewritten using method reference.

```
ToIntFunction<String> lenFunction = String::length;

Supplier<Item> func1 = Item::new;

Function<String, Item> func2 = Item::new;

BiFunction<String, Double, Item> func3 = Item::new;
```

Revisit the `Comparator<T>` interface

Modifier and Type	Method and Description
int	**compare**(**T** o1, **T** o2) Compares its two arguments for order.
static <T,U extends **Comparable**<? super U>> **Comparator**<T>	**comparing**(**Function**<? super T,? extends U> keyExtractor) Accepts a function that extracts a **Comparable** sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T,U> **Comparator**<T>	**comparing**(**Function**<? super T,? extends U> keyExtractor, **Comparator**<? super U> keyComparator) Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified **Comparator**.
static <T> **Comparator**<T>	**comparingDouble**(**ToDoubleFunction**<? super T> keyExtractor) Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> **Comparator**<T>	**comparingInt**(**ToIntFunction**<? super T> keyExtractor) Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> **Comparator**<T>	**comparingLong**(**ToLongFunction**<? super T> keyExtractor) Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key.
boolean	**equals**(**Object** obj) Indicates whether some other object is "equal to" this comparator.
static <T extends **Comparable**<? super T>> **Comparator**<T>	**naturalOrder**() Returns a comparator that compares **Comparable** objects in natural order.
static <T> **Comparator**<T>	**nullsFirst**(**Comparator**<? super T> comparator) Returns a null-friendly comparator that considers null to be less than non-null.
static <T> **Comparator**<T>	**nullsLast**(**Comparator**<? super T> comparator) Returns a null-friendly comparator that considers null to be greater than non-null.
default **Comparator**<**T**>	**reversed**() Returns a comparator that imposes the reverse ordering of this comparator.
static <T extends **Comparable**<? super T>> **Comparator**<T>	**reverseOrder**() Returns a comparator that imposes the reverse of the *natural ordering*.
default **Comparator**<**T**>	**thenComparing**(**Comparator**<? super **T**> other) Returns a lexicographic-order comparator with another comparator.
default <U extends **Comparable**<? super U>> **Comparator**<**T**>	**thenComparing**(**Function**<? super **T**,? extends U> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
default <U> **Comparator**<**T**>	**thenComparing**(**Function**<? super **T**,? extends U> keyExtractor, **Comparator**<? super U> keyComparator) Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.

default **Comparator**<**T**>	**thenComparingDouble**(**ToDoubleFunction**<? super **T**> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a double sort key.
default **Comparator**<**T**>	**thenComparingInt**(**ToIntFunction**<? super **T**> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a int sort key.
default **Comparator**<**T**>	**thenComparingLong**(**ToLongFunction**<? super **T**> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a long sort key.

Very often, we want to compare 2 objects based on selected data field(s).

```
class Student
{
   private String name;
   private int sid;
   private String major;

   public Student(String n, int s, String m)
   {
      name = n;
      sid = s;
      major = m;
   }

   public String getName()
   {
      return name;
   }

   public int getSid()
   {
      return sid;
   }

   public String getMajor()
   {
      return major;
   }

   ...  // other methods
}
```

```
ArrayList<Student> list = new ArrayList();
...  // codes to initialize the contents of list


// Comparator that compares Student by major
// and then by name
Comparator cmp = new Comparator<Student>() {
          int compare(Student s1, Student s2)
          {
             int r = s1.getMajor().compareTo(s2.getMajor());
             if (r != 0)
                return r;

             return s1.getName().compareTo(s2.getName());
          }
       };

list.sort(cmp);
```

Methods **comparing** and **thenComparing** in `Comparator` interface

```
static <T, U extends Comparable<? super U>> Comparator<T>
  comparing(Function<? super T, ? extends U> keyExtractor)

default <U extends Comparable<? super U>> Comparator<T>
  thenComparing(Function<? super T, ? extends U> keyExtractor)


// keyExtractor extracts a value of type U (U is Comparable)
// from an object of type T (or super class of T)

// The static method comparing returns a Comparator that
// compares objects of type T based on a data field of type U
// extracted from the objects to be compared.

// The default method thenComparing is applied to an implicit
// Comparator object c1 (returned by static method comparing)
// to create a new Comparator object c2.
```

```
// Comparator to compare Student by major and then by name.

list.sort(comparing(Student::getMajor)
          .thenComparing(Student::getName));
```

`java.util.Optional<T>` in Java 8

An `Optional<T>` is used to represent a value is present or absent.

It can help to avoid runtime `NullPointerException`, and supports us in developing clean and neat Java APIs or applications.

Example

```
// Find student name by sid
static String findName(List<Student> list, int sid)
{
   for (Student s : list)
      if (s.getSid() == sid)
         return s.getName();

   return null; // sid not found
}

static void testFn(List<Student> list)
{
   String result = findName(list, 1234);

   // possible NullPointerException
   System.out.println(result);

   /* To avoid NullPointerException

   if (result != null)  // null checking
      System.out.println(result);
   else
      // do something else
   */

}
```

Code design using `Optional<T>`

```
static Optional<String> findName(List<Student> list, int sid)
{
   for (Student s : list)
      if (s.getSid() == sid)
         return Optional.of(s.getName());

   return Optional.empty(); // sid not found

   // Optional.of(value) : create an Optional with the given
   //                      non-null value

   // Optional.empty() : create an empty Optional instance
}

static void testFn(List<Student> list)
{
   Optional<String> result = findName(list, 1234);

   System.out.println(result);

   // If sid is not found, output:
   //    Optional.empty

   // If sid is found, output:
   //    Optional[name]

}
```

Methods in the class `Optional<T>`

Modifier and Type	Method and Description
static <T> **Optional**<T>	**empty**() Returns an empty Optional instance.
boolean	**equals**(**Object** obj) Indicates whether some other object is "equal to" this Optional.
Optional<**T**>	**filter**(**Predicate**<? super **T**> predicate) If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.
<U> **Optional**<U>	**flatMap**(**Function**<? super **T**,**Optional**<U>> mapper) If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
T	**get**() If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
int	**hashCode**() Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
void	**ifPresent**(**Consumer**<? super **T**> consumer) If a value is present, invoke the specified consumer with the value, otherwise do nothing.
boolean	**isPresent**() Return true if there is a value present, otherwise false.
<U> **Optional**<U>	**map**(**Function**<? super **T**,? extends U> mapper) If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result.
static <T> **Optional**<T>	**of**(T value) Returns an Optional with the specified present non-null value.
static <T> **Optional**<T>	**ofNullable**(T value) Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
T	**orElse**(**T** other) Return the value if present, otherwise return other.
T	**orElseGet**(**Supplier**<? extends **T**> other) Return the value if present, otherwise invoke other and return the result of that invocation.
<X extends **Throwable**> **T**	**orElseThrow**(**Supplier**<? extends X> exceptionSupplier) Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.
String	**toString**() Returns a non-empty string representation of this Optional suitable for debugging.

Refined example to illustrate the uses of `isPresent()`, `get()`, `orElse()`, and `map()`

```
static void testFn(List<Student> list)
{
   Optional<String> result = findName(list, 1234);

   // We want to modify the output format.

   // If sid is not found, output:
   //    Not Found

   // If sid is found, output:
   //    name

   if (result.isPresent())  // if Optional has a value
      System.out.println(result.get()); // get the value
   else
      System.out.println("Not Found");

   // Alternative implementation using orElse()
   System.out.println(result.orElse("Not Found"));

   // If sid is found, output name in upper case:
   //    NAME
   System.out.println(result.map(String::toUpperCase)
                            .orElse("Not Found"));

}
```