

# Java 8 Lambda Expression: Functional Interface

**“Code with Passion!”**



# Topics

- Java 8 provided Functional Interface definitions
  - > Simply typed functional interfaces
  - > Generically typed functional interfaces
- Usage example of *Predicate*
- Usage example of *Function*
- Composition

# **Java 8 Provided Functional Interfaces**

# Functional Interfaces defined in Java 8

- Java 8 comes with a set of **commonly used functional interfaces** in *java.util.function* package - so you don't have to define your own anymore
- There are two kinds of Java 8 provided functional interfaces
  - > Simply typed
  - > Generically typed

# Simply Typed Functional Interfaces

# Simply Typed Functional Interfaces

- Use simply typed functional interfaces when the types of arguments and return type can be pre-determined
- Simply typed functional interfaces
  - > IntPredicate int -> boolean test()
  - > LongPredicate long -> boolean test()
  - > LongUnaryOperator long ->long applyAsLong()
  - > DoubleBinaryOperator (double, double) -> double applyAsDouble()
  - > ...

```
// Implementation of the functional interface is provided  
LongUnaryOperator operator = x -> x * 10;
```

```
// You are going to call the method of the functional interface  
Long resultLong = operator.applyAsLong(20L);
```

# IntPredicate int->boolean test()

- Represents a predicate (boolean-valued function) of one int-valued argument
  - > This is the int-consuming primitive type specialization of Predicate

```
// Definition
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);
}
```

```
// Usage
IntPredicate predicate = x -> x > 10;
boolean resultBoolean = predicate.test(5);
```

# Lab:

**Exercise 1: Java 8 Provided  
Functional Interfaces - Simply Typed**  
**[1621\\_javase8\\_lambda\\_fi.zip](#)**



# Generically Typed Functional Interfaces

# Generically Types Functional Interfaces

- Supplier<T>              () -> T              get()
- Consumer<T>              T -> void            accept()
- BiConsumer<T,U>         (T,U) -> void        accept()
- Function<T,R>            T -> R                apply()
- BiFunction<T,U,R>       (T,U) -> R            apply()
- Predicate<T>             T -> boolean        test()
- BiPredicate<T,U>        (T,U) -> boolean    test()
- UnaryOperator<T>        T -> T                apply()
- BinaryOperator<T>       (T,T) -> T            apply()

# Supplier<T>      ()->T      get()

- Represents a supplier of results
  - > This is a functional interface whose functional method is get()
  - > <T> the type of result supplied by this supplier

```
// Definition
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

```
// Usage examples
Supplier<String> supplier1 = () -> "String1";
Supplier<Integer> supplier2 = () -> "String1".length();
```

# **Consumer<T>    T->void    accept()**

- Represents an operation that accepts a single input argument and returns no result
  - > This is a functional interface whose functional method is `accept(Object)`
  - > `<T>` the type of the input to the operation
- Variation: `BiConsumer<T,U>    (T,U) -> void    accept()`

```
// Definition
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
// Usage examples
Consumer<String> consumer1 = x -> System.out.println(x);
Consumer<Integer> consumer2 = y -> System.out.println(y*y);
```

# Function<T,R>

T->R

apply()

- Represents a function that accepts one argument and produces a result
  - > The lambda expression is the body of the *apply()* method
  - > <T> the type of the input to the function
  - > <R> the type of the result of the function
- Variation: BiFunction<T,U,R> (T,U) -> R apply()

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

```
// Usage examples
Function<String, String> function1 = x -> x.toUpperCase();
Function<String, Integer> function2 = x -> x.length();
```

# Predicate<T> T->boolean test()

- Represents a predicate (boolean-valued function) of one argument.
  - > <T> the type of the input to the predicate
- BiPredicate<T,U> (T,U) -> boolean test()

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

```
// Usage examples
Predicate<Double> predicate1 = x -> x > 10;
Predicate<String> predicate2 = s -> s.length() > 10;
```

# UnaryOperator<T> T->T apply()

- UnaryOperator is a java 8 functional interface that extends Function
- UnaryOperator is used to work on a single operand. It returns the same type as an operand
- BinaryOperator<T> (T,T) -> T apply()

```
// Definition
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

```
// Usage example
UnaryOperator<String> unaryOperator1 = x -> x.toUpperCase();
```

# Lab:

**Exercise 2: Java 8 Provided  
Functional Interfaces - Generically Typed  
[1621\\_javase8\\_lambda\\_fi.zip](#)**



# Usage Example of Predicate

# Example Usage of Predicate

- Example scenario
  - > From a list, find all items that meet a test criteria - use predicate to filter out unqualified items
- There are several options to write this code (in the order of least desirable to most desirable option)
  - > #1: Embed test code in a for loop (In Java 7) - least desirable
  - > #2: Use predicate with specific type (Java 8)
  - > #3: Use predicate with generic type (Java 8)
  - > #4: Use a stream (Java 8) - most desirable

# #1: Embed test code in a for loop (Java 7)

- Find all people who has name “Jon”

```
public static List<Person> findPeopleByName(List<Person> people, String name) {  
    List<Person> result = new ArrayList<Person>();  
    for (Person p : people) {  
        if (p.getName().equals(name)) { result.add(p); }  
    }  
    return result;  
}
```

(Bad) Every time you need to perform a search using a new test criteria, you have to write new code. Here, we had to write 3 different code.

- Find all people whose age is greater than 10

```
public static List<Person> findPeopleByAge(List<Person> people, int age){ // Code ...  
}
```

- Find all people who has name “Jon” and whose age is greater than 10

```
public static List<Person> findPeopleByNameAndAge(List<Person> people, String name, int age){ // ...  
}
```

## #2 Use Predicate for specific type (Java 8)

- Find all people who has name “Jon”

```
public static List<Person> finePeople(List<Person> people, Predicate<Person> aPredicate) {
```

```
    List<Person> result = new ArrayList<Person>();  
    for (Person p : people) {  
        if (aPredicate.test(p)) { result.add(p);}  
    }  
    return result;  
}
```

```
peopleResult = finePeople(people, person -> person.getName().equals("Jon"));
```

(Good) Every time you need to perform a search with a new test criteria, you just write new predicate.

(Bad) The code works only with Person type and cannot be used other types

- Find all people whose age is greater than 10

```
peopleResult = finePeople(people, person -> person.getAge() > 10);
```

- Find all people who has name “Jon” and whose age is greater than 10

```
peopleResult = finePeople(people, person -> person.getName().equals("Jon") &&  
                           person.getAge() > 10);
```

# #3 Use Predicate for Generic type (Java 8)

- Find people using test criteria

```
public static <T> List<T> find(List<T> myList, Predicate<T> aPredicate) {  
    List<T> result = new ArrayList<T>();  
    for (T item : myList) {  
        if (aPredicate.test(item)) {result.add(item);} }  
    return result;  
}
```

(Good) The same code can be used with different types - Person and Fruit

(Bad) The code still uses for loop

```
List<Person> peopleResult = find(people, person -> person.getName().equals("Jon"));  
peopleResult = find(people, person -> person.getAge() > 10);  
peopleResult = find(people, person -> person.getName().startsWith("J") && person.getAge() > 10);
```

- Find fruits using test criteria

```
List<Fruit> fruitResult = find(fruits, fruit -> fruit.getName().equals("Apple"));  
fruitResult = find(fruits, fruit -> fruit.getQuantity() > 10);  
fruitResult = find(fruits, fruit -> fruit.getName().startsWith("J") && fruit.getQuantity() > 10);
```

## #4 Use Stream's filter (Java 8)

- <We are going to cover Streams in another presentation in detail. It is mentioned here for the sake of completeness>

```
Stream<Person> resultPeople = people.stream().filter(person -> person.getName().equals("Jon"));  
resultPeople.forEach(person -> System.out.println(person.getName()));
```

```
Stream<Fruit> resultFruits = fruits.stream().filter(fruit -> fruit.getName().equals("Apple"));  
resultFruits.forEach(fruit -> System.out.println(fruit.getName()));
```

(Good) The code is simple and fluent

# Lab:

**Exercise 3: Usage Example of Predicate  
1621\_javase8\_lambda\_fi.zip**



# Usage Example of Function

# Example Usage of Function

- Example scenario
    - > From a list, convert each item using conversion logic - use Function to perform conversion of each item
  - There are several options to write this code (in the order of least desirable to most desirable)
    - > #1: Embed conversion code in a for loop (In Java 7) - least desirable
    - > #2: Use Function with specific type (Java 8)
    - > #3: Use Function with generic type (Java 8)
    - > #4: Use a stream (Java 8) - most desirable

# #1: Embed conversion code in a for loop (Java 7)

- Convert name of each person to uppercase

```
public static void convertPeopleUppercase(List<Person> people) {  
    for (Person p : people) {  
        p.setName(p.getName().toUpperCase());  
    }  
}
```

(Bad) Every time you need to perform a new conversion, you have to write new code.

- Convert name of each person to lowercase

```
public static void convertPeopleLowercase(List<Person> people) {  
    for (Person p : people) {  
        p.setName(p.getName().toLowerCase());  
    }  
}
```

- Convert name of each person to camelcase

```
public static void convertPeopleCamelcase(List<Person> people) {  
}
```

## #2: Use Function for specific type (Java 8)

- Convert name of each person to uppercase

```
public static void convertPeople(List<Person> people, Function<Person, Person>aFunction) {  
    for (Person p : people) {  
        aFunction.apply(p);  
    }  
}
```

(Good) Every time you need to perform a new conversion, you just write new Function.

(Bad) The code works only with Person type and cannot be used other types

```
Function<Person, Person> aFunction1 =  
    person -> {person.setName(person.getName().toUpperCase()); return person;};  
convertPeople(people, aFunction1);
```

- Convert name of each person to lowercase

```
Function<Person, Person> aFunction2 =  
    person -> {person.setName(person.getName().toLowerCase()); return person;};  
convertPeople(people, aFunction2);
```

- Convert name of each person to camelcase

# #3: Use Function for Generic type (Java 8)

- Convert name of each person to uppercase

```
public static <T,R> void convert(List<T> myList, Function<T, R> aFunction) {
```

```
    for (T t : myList) {  
        aFunction.apply(t);  
    }  
}
```

(Good) The same code can be used  
with different types - Person and Fruit

(Bad) The code still uses for loop

```
Function<Person, Person> aFunction1 =
```

```
    person -> {person.setName(person.getName().toUpperCase()); return person;};  
convert(people, aFunction1);
```

- Convert name of each person to lowercase

```
Function<Person, Person> aFunction2 =
```

```
    person -> {person.setName(person.getName().toLowerCase()); return person;};  
convert(people, aFunction2);
```

- Convert name of each fruit to camelCase

```
Function<Fruit, Fruit> aFunction3 = ...
```

## #4 Use Stream's map (Java 8)

- <We are going to cover Streams in another presentation in detail. It is mentioned here for the sake of completeness>

```
// Use stream
Stream<Person> resultPeople =
    people
        .stream()
        .map(person -> {person.setName(person.getName().toUpperCase()); return person;});
resultPeople.forEach(person -> System.out.print(person.getName() + " "));
```

(Good) The code is simple and fluent

# Lab:

**Exercise 4: Usage Example of Function  
1621\_javase8\_lambda\_fi.zip**



# Composition of Lambda Expressions

# Composition of Lambda Expressions

- Composition allows applying lambda expressions one after another
- There are two methods:
  - > *Function compose(Function before)* - The before function is applied first and then the calling function
  - > *Function andThen(Function after)* - The after function is applied after the calling function

# andThen and compose

```
// Functions without composition
```

```
Function<Person, Address> personToAddressFunction = (person) -> person.getAddress();
Function<Address, String> addressToCountryFunction = (address) -> address.getCountry();
Address address = personToAddressFunction.apply(new Person("Sang", new Address("Korea")));
String country = addressToCountryFunction.apply(address);
```

```
// Functions with "andThen" composition
```

```
Function<Person, String> personToCountryFunction1 =
    personToAddressFunction.andThen(addressToCountryFunction);
country = personToCountryFunction1.apply(new Person("Jon", new Address("USA")));
```

```
// Functions with "compose" composition
```

```
Function<Person, String> personToCountryFunction2 =
    addressToCountryFunction.compose(personToAddressFunction);
country = personToCountryFunction2.apply(new Person("Jon", new Address("China")));
```

# Lab:

**Exercise 5: Composition**  
**1621\_javase8\_lambda\_fi.zip**



# Code with Passion!

