Professional Programming in Java

Session: 15

Effective Programming with Lambda

Objectives

- Explain lambdas
- Identify the built-in functional interfaces
- Explain code refactoring for readability using lambdas
- Describe debugging of lambda

Lambda Usage [1-2]

- Lambda expressions
  - Introduced in Java 8
  - Unnamed blocks of code
  - Facilitate functional programming

Lambda Usage [2-2]

- Types of lambda expressions are:

Object lambda

Inline lambda

- Inferred type lambdas
  - Allow passing parameters to the expression body

- Code for creating different types of lambdas and using them:

Code Snippet

```
package lambdaexpressiondemo;

import java.util.Arrays;

@FunctionalInterface
interface FunctionalA {
   int doWork(int a, int b);
}
public class LambdaExpressionDemo {
  public static void main(String[] args) {
     /*Lambda 1: Using basic lambda */
       FunctionalA functionalA1 = (int num1, int num2) ->
       num1 +num2;
     System.out.println("5+5= " + functionalA1.doWork(5,
     5));
```

Lambda Types [2-5]

```
/*Lambda 2: Using lambda with inferred types */
FunctionalA functionalA2 = (num1, num2) -> num1 +
num2;
System.out.println("5+10= " +
functionalA2.doWork(5, 10));
/*Lambda 3: Using lambda with expression body
containing return statement */
FunctionalA functionalA3 = (num1, num2) -> {
return num1 + num2;
};
System.out.println("5+15= " +
functionalA3.doWork(5, 15));
/*Lambda 4: Using lambda with expression body
containing multiple statements */
```

Lambda Types [3-5]

```
FunctionalA functionalA4 = (num1, num2) -> {
int sum = num1 + num2;
int result = sum * 10;
return result;
};

System.out.println("(5+10)*10= " +
functionalA4.doWork(5,
10));
/*Lambda 5: Passing lambda as method parameter to
Arrays.sort() method*/
String[] words = new String[]{"Hi", "Hello",
"HelloWorld",
"H"};
System.out.println("Original array= " +
Arrays.toString(words));
Arrays.sort(words,
```

Lambda Types [4-5]

```
    (first, second) -> Integer.compare(first.length(),
    second.length()));
    System.out.println("Sorted array by length using
    lambda= "+ Arrays.toString(words));
  }
}
```

Following is the output of the code:

```
Output - LambdaExpressionDemo (run)
run:
5+5= 10
5+10= 15
5+15= 20
(5+10)*10= 150
Original array= [Hi, Hello, HelloWorld, H]
Sorted array by length using lambda= [H, Hi, Hello, HelloWorld]
BUILD SUCCESSFUL (total time: 2 seconds)
```

Lambda Types [5-5]

- The lambdas used in the code are:
  - **Lambda 1**: Uses basic lambda syntax to assign the value of a lambda expression to a variable of type `FunctionalA`.
  - **Lambda 2**: Performs the same function as Lambda 1 but without specifying the parameter types.
  - **Lambda 3**: Performs the same function as Lambda 1 but with an explicit return statement enclosed within curly braces ({}).
  - **Lambda 4**: Uses multiple statements in the lambda expression body before returning the final value to the caller.
  - **Lambda 5**: Passes a lambda as the second parameter to the `Arrays.sort(T[] a, Comparator<? super T> c)` method that sorts the specified array of objects according to the order induced by the comparator.

Built-in Functional Interfaces [1-4]

- Following table lists functional interfaces:

Interface	Abstract Method	Description
`Predicate<T>`	`boolean test(T t)`	Represents an operation that checks a condition and returns a `boolean` value as result
`Consumer<T>`	`void accept(T t)`	Represents an operation that takes an argument but returns nothing
`Function<T, R>`	`R apply(T t)`	Represents an operation that takes an argument and returns some result to the caller
`Supplier<T>`	`T get()`	Represents an operation that does not take any argument but returns a value to the caller

Built-in Functional Interfaces [2-4]

- Code shows use of common built-in functional interfaces.

Code Snippet

```
package lambdaexpressiondemo;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;

static void testPredicate() {
   Predicate<String> result = arg -> (arg.equals("Hello
   Lambda"));
   String testStr = "Hello Lambda";
   System.out.println(result.test(testStr));
}
```

Built-in Functional Interfaces [3-4]

```
public class FunctionalInterfacesDemo {
  static void testConsumer() {
     Consumer<String> result = str ->
     System.out.println(str.toUpperCase());
     result.accept("hello lambda");
}

   static void testFunction() {
     Function<String, Integer> result = str ->
     str.length();
     System.out.println(result.apply("Hello Lambda"));
   }
   static void testProducer() {
     Supplier result = () -> {return "Hello lambda
     from supplier";};
     System.out.println(result.get());
   }
```

Built-in Functional Interfaces [4-4]

```
public static void main(String[] args) {
   testPredicate();
   testConsumer();
   testFunction();
   testProducer();
}}
```

Following is the output of the code:

Output - FunctionalInterfacesDemo (run)

```
run:
true
HELLO LAMBDA
12
Hello lambda from supplier
BUILD SUCCESSFUL (total time: 1 second)
```

Primitive Versions of Functional Interfaces [1-3]

Predicate<T>, Consumer<T>, Function<T, R>, and Supplier<T> operate on reference type objects.

Primitive values, such as int, long, float, or double cannot be used with them.

Java 8 provides primitive versions for functional interfaces.

Primitive Versions of Functional Interfaces [2-3]

Code for use of the primitive versions of the `Predicate` and `Consumer` functional interfaces.

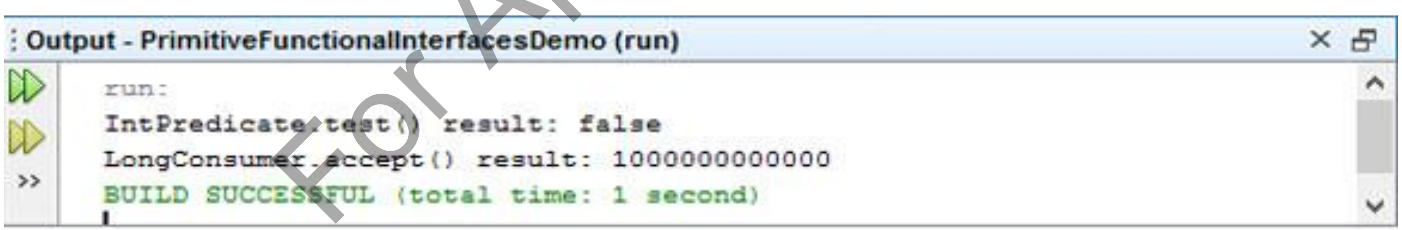
Code Snippet

```
package lambdaexpressiondemo;
import java.util.function.IntPredicate;
import java.util.function.LongConsumer;
public class PrimitiveFunctionalInterfacesDemo {
static void testIntPredicate() {
IntPredicate result = arg -> (arg==10);
System.out.println("IntPredicate.test() result:
"+result.test(11));
}
```

Primitive Versions of Functional Interfaces [3-3]

```
static void testlongConsumer() {
LongConsumer result = val ->
System.out.println("LongConsumer.accept() result:
"+val*val);
result.accept(1000000);
}
public static void main(String[] args) {
testIntPredicate();
testlongConsumer();
}
}
```

Following is the output of the code:



Binary Versions of Functional Interfaces [1-3]

- Abstract methods of the `Predicate`, `Consumer`, and `Function` functional interfaces accept one argument.
- Java 8 provides equivalent binary versions of such functional interfaces that can accept two parameters.
- Binary functional version interfaces are prefixed with Bi.

Binary Versions of Functional Interfaces [2-3]

- The code demonstrates the use of the binary versions of the `Predicate` and `Consumer` functional interfaces.

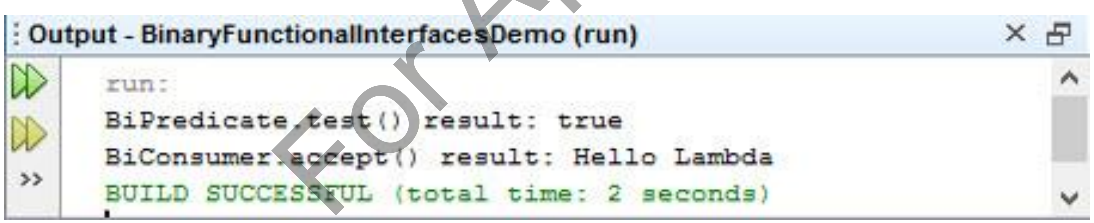
Code Snippet

```
package lambdaexpressiondemo;
import java.util.function.BiPredicate;
import java.util.function.BiConsumer;
public class BinaryFunctionalInterfacesDemo {
static void testBiPredicate() {
BiPredicate<Integer, Integer> result = (arg1, arg2) ->
arg1
< arg2;
System.out.println("BiPredicate.test() result:
"+result.test(5,10));
}
```

Binary Versions of Functional Interfaces [3-3]

```
static void testBiConsumer() {
BiConsumer<String,String> result = (arg1, arg2) ->
System.out.println("BiConsumer.accept() result:
"+arg1+arg2);
result.accept("Hello ", "Lambda");
}
public static void main(String[] args) {
testBiPredicate();
testBiConsumer();
}
}
```

Following is the output of the code:



UnaryOperator Interface [1-2]

- `UnaryOperator`

Present in the `java.util.function` package

Is a specialized version of the `Function` interface.

Can be used on a single operand when the types of the operand and result are the same.

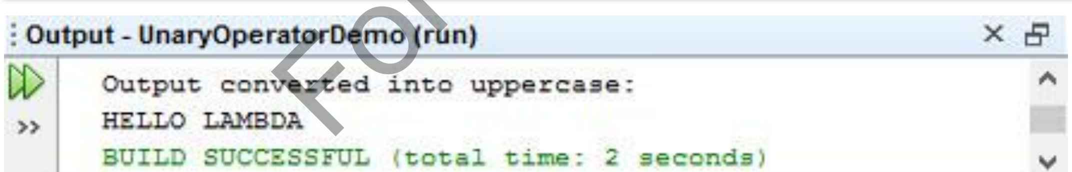
UnaryOperator Interface [2-2]

- The code demonstrates use of `UnaryOperator`.

Code Snippet

```
package lambdaexpressiondemo;
import java.util.function.UnaryOperator;
public class UnaryOperatorDemo {
public static void main(String[] args) {
UnaryOperator<String> result = (x)-> x.toUpperCase();
System.out.println("Output converted into
uppercase:");
System.out.println(result.apply("Hello Lambda"));
}
}
}
```

Following is the output of the code:



Refactoring for Improved Readability [1-3]

- Lambda
  - Is useful for Java programmers for expressing problems in many situations.
  - Its introduction does not break code.
- Existing code can run as it is with new code containing lambdas running alongside.
- Refactoring will be done to remove existing boilerplate code and make the existing code more concise.

Refactoring for Improved Readability [2-3]

- The code demonstrates how to create the different types of lambdas and use them:

Code Snippet

```
package lambdaexpressiondemo;
public class MultiThreadedAnonymousDemo {
public static void main(String[] args) {
Runnable r1 = new Runnable(){
@Override
public void run(){
System.out.println("Hello from anonymous");
}
};
r1.run();
}
}
```

Refactoring for Improved Readability [3-3]

- The code demonstrates the use of lambda to write a `Runnable` instance and run it.

Code Snippet

```
package lambdaexpressiondemo;
public class MultiThreadedLambdaDemo {
public static void main(String[] args) {
Runnable r1 = () -> {
System.out.println("Hello from lambda");
};
r1.run();
}
}
```

Refactoring Comparison Code [1-5]

- `Comparator` Interface:

Enables comparing the elements of a collection that need to be sorted.

Is a functional interface that contains the single int compare(T o1, T o2) method.

- When a collection or array needs to be sorted, a Comparator object is passed to the `Collections.sort()` or `Arrays.sort()` method.

Refactoring Comparison Code [2-5]

- The code applies a `Comparator` using an anonymous inner class to sort a `List` of objects:

Code Snippet

```
. . .
Collections.sort(employeeList, new Comparator<Employee>() {
@Override
public int compare(Employee emp1, Employee emp2) {
   return emp1.getLastName().compareTo(emp2.getLastName());
}
});
System.out.println("=== Sorted Employee by last name in
ascending order ===");
for (Employee emp : employeeList) {
System.out.println(emp.getFirstName() + " " +
emp.getLastName());
}
. . .
```

Refactoring Comparison Code [3-5]

- The code snippet lists the complete code that applies a `Comparator` using a lambda to sort a `List` of `Employee` objects.

Code Snippet

```
package lambdaexpressiondemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
class Employee{
private String firstName;
```

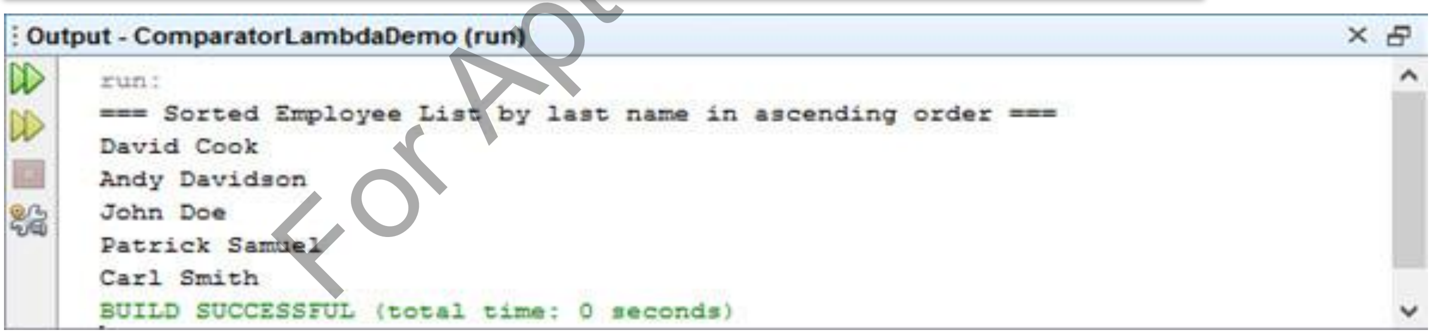
Refactoring Comparison Code [4-5]

```
private String lastName;
public Employee(String firstName, String lastName) {
this.firstName = firstName;
this.lastName = lastName;
}
public String getFirstName() {
return firstName;
}
public String getLastName() {
return lastName;
}
}
public class ComparatorLambdaDemo {
public static void main(String[] args) {
List<Employee> employeeList = new ArrayList<>();
employeeList.add(new Employee("Patrick", "Samuel"));
employeeList.add(new Employee("John", "Doe"));
employeeList.add(new Employee("Andy", "Davidson"));
employeeList.add(new Employee("Carl", "Smith"));
```

Refactoring Comparison Code [5-5]

```
employeeList.add(new Employee("David", "Cook"));
Comparator<Employee> sortedEmployee = (Employee emp1,
Employee emp2) -> emp1.getLastName()
.compareTo(emp2.getLastName());
System.out.println("=== Sorted Employee List by last
name in ascending order
===");
Collections.sort(employeeList,sortedEmployee);
for (Employee emp : employeeList) {
System.out.println(emp.getFirstName() + " " +
emp.getLastName());
}}}
```

Following is the output of the code:



Refactoring Concurrency Code [1-5]

- Callable and Future are extensively used in multithread Java applications to implement asynchronous processing.
- Callable
  - can return a value.
  - is a functional interface in the `java.util.concurrent` package.
- The Callable interface has a single abstract method, `V call()`.
- When a Callable is passed to a thread pool maintained by `ExecutorService`, the pool selects a thread and execute the Callable.
- The `get()` method of Future returns the computation result or block if the computation is not complete.

Refactoring Concurrency Code [2-5]

The code snippet demonstrates use of an anonymous class to run a piece of code in a different thread with `Callable` and `Future`.

Code Snippet

```
. . .
ExecutorService executor = Executors.newFixedThreadPool(5);
Callable callable = new Callable(){
  @Override
  public String call(){
    try{
        Thread.sleep(10);
        return Thread.currentThread().getName();
    }
    catch(InterruptedException ie){
        ie.printStackTrace();
    }
    return Thread.currentThread().getName();
  }
};
Future<String> future = executor.submit(callable);
. . .
```

Refactoring Concurrency Code [3-5]

The code snippet demonstrates the complete code refactored to construct a `Callable` using lambda instead of an anonymous and to submit the `Callable` to an `ExecutorService`.

Code Snippet

```
package lambdaexpressiondemo;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CallableLambdaDemo {
public static void main(String[] args) {
ExecutorService executor =
Executors.newFixedThreadPool(5);
```

Refactoring Concurrency Code [4-5]

```
List<Future<String>> list = new ArrayList<>();
Callable callable = () -> {
try {
Thread.sleep(10);
return Thread.currentThread().getName();
} catch (InterruptedException ie) {
ie.printStackTrace();
}
return Thread.currentThread().getName();
};
for (int i = 0;i < 10; i++) {
Future<String> future = executor.submit(callable);
list.add(future);
}
for (Future<String> future : list) {
try {
System.out.println(future.get());
} catch (InterruptedException | ExecutionException e)
{
```

Refactoring Concurrency Code [5-5]

```
e.printStackTrace();
}
}
executor.shutdown();
}
}
```

Following is the output of the code:

```
Output - CallableDemo (run)
run:
pool-1-thread-1
pool-1-thread-2
pool-1-thread-3
pool-1-thread-4
pool-1-thread-5
pool-1-thread-5
pool-1-thread-2
pool-1-thread-1
pool-1-thread-4
pool-1-thread-3
BUILD SUCCESSFUL (total time: 1 second)
```

Debugging Lambdas [1-5]

The code snippet demonstrates a Java class with lambda that you can debug in NetBeans.

Code Snippet

```
package lambdaexpressiondemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Employee{
  private String firstName;
  private String lastName;
  public Employee(String firstName, String
lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
```

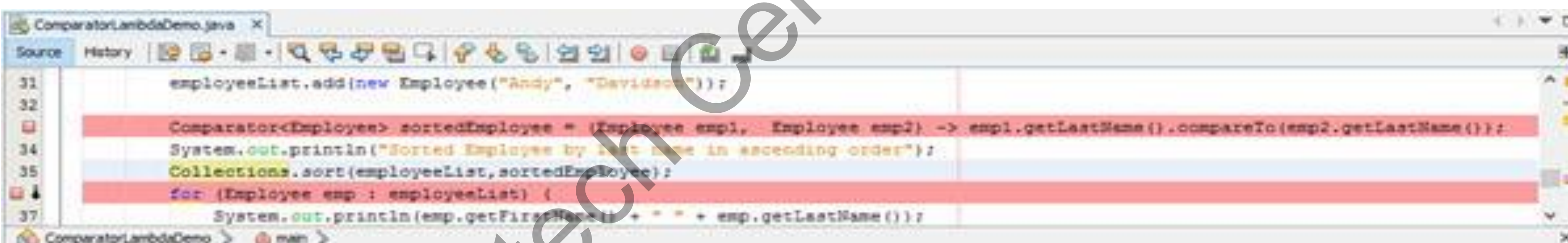
Debugging Lambdas [2-5]

```
public String getFirstName() {
   return firstName;
}
public String getLastName() {
    return lastName;
}
}
public class ComparatorLambdaDemo {
  public static void main(String[] args) {
  List<Employee> employeeList = new ArrayList<>();
  employeeList.add(new Employee("Patrick", "Samuel"));
  employeeList.add(new Employee("John", "Doe"));
  employeeList.add(new Employee("Andy", "Davidson"));
  Comparator<Employee> sortedEmployee = (Employee emp1,
  Employee emp2) ->
  emp1.getLastName().compareTo(emp2.getLastName());
```

Debugging Lambdas [3-5]

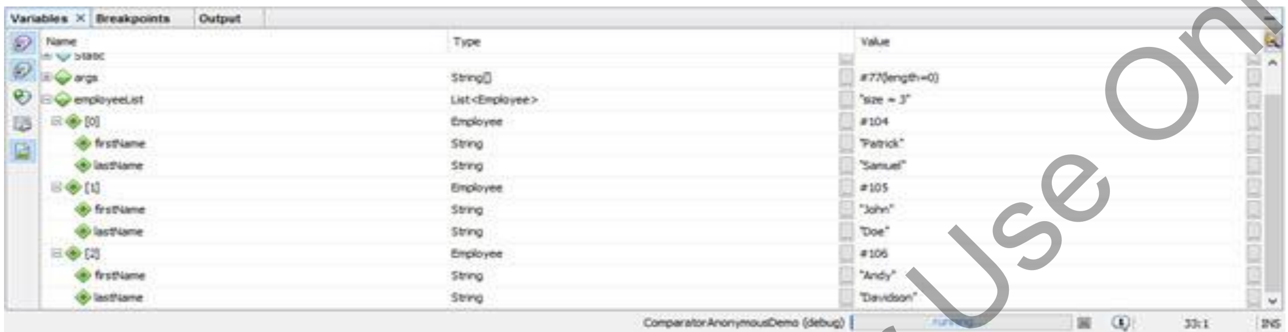
```
     System.out.println("Sorted Employee by last name
     in ascending order");
     Collections.sort(employeeList,sortedEmployee);
     for (Employee emp : employeeList) {
     System.out.println(emp.getFirstName() + " " +
     emp.getLastName());
   }
  }
}
```

- To test the lambda used in the code:
  - Open the `ComparatorLambdaDemo` class in `NetBeans`.
  - In the code editor, double-click the line number of the statement that uses lambda to set a breakpoint.
  - In the code editor, double-click the line number of the statement containing the for loop to set a breakpoint.



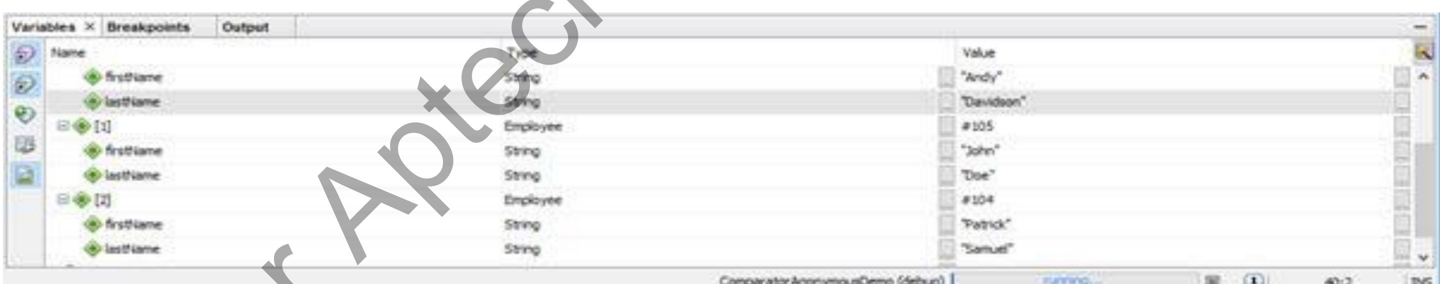
  - Select Debug → Debug Project from the main menu of `NetBeans`. The program execution stops in the first breakpoint.
  - Observe the first name and last name values in the Variables window displayed. At this point, the lambda is yet to perform the sorting.

Debugging Lambdas [5-5]



Employee Values before Sorting

- Select Debug → Continue from the main menu to continue debugging until the debugging thread hits the second breakpoint.
- Check the Variables window to ensure that the lambda has correctly performed the sorting based on the last name.



Employee Values after Sorting

- Select Debug → Finish Debugger Session to stop debugging.

Summary [1-2]

- A lambda expression is an unnamed block of code that facilitates functional programming.
- `java.util.function` package introduced in Java 8 contains a large number of functional interfaces.
- Java 8 provides primitive versions for functional interfaces to operate on primitive values.
- Java 8 also provides equivalent binary versions of some functional interfaces that can accept two parameters.

Summary [2-2]

- `UnaryOperator` interface is used on a single operand when the types of the operand and result are the same.
- Java programmers can use lambdas to express problems in a shorter and more readable way.
- Lambda expressions can be debugged in NetBeans like any piece of Java code by setting breakpoints.