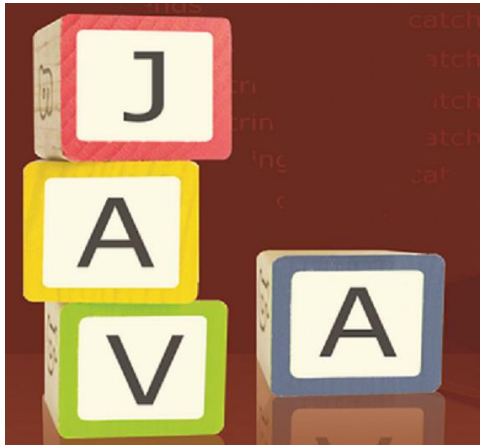


Effective programming with lambda

SESSION 15

Objectives

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

Lambda usage

- Lambda expression:
  - Introduced in Java 8
  - Unnamed blocks of code
  - Facilitate functional programming
- Types of lambda expressions are:
  - Object lambda
  - Inline lambda
- Inferred type lambdas
  - Allow passing parameters to the expression body

Lambda types [1/2]

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

```
12  import java.util.*;
13
14  @FunctionalInterface
15  interface FunctionalA {
16      int doWork(int a, int b);
17  }
18  public class LambdaExpressionDemo {
19      public static void main(String[] args) {
20          /*1: Using basic lambda */
21          FunctionalA a1 = (int n1, int n2) -> n1 + n2;
22          System.out.println("5+5= " + a1.doWork(5, 5));
23
24          /*2: Using lambda with inferred types */
25          FunctionalA a2 = (n1, n2) -> n1 + n2;
26          System.out.println("5+10= " + a2.doWork(5, 10));
27
28          /*3: Using lambda with expression body containing return statement */
29          FunctionalA a3 = (n1, n2) -> {
30              return n1 + n2;
31          };
32          System.out.println("5+15= " + a3.doWork(5, 15));
```

Lambda types [2/2]

```
33
34          /*4: Using lambda with expression body containing multiple statements */
35          FunctionalA a4 = (n1, n2) -> {
36              int sum = n1 + n2;
37              int result = sum * 10;
38              return result;
39          };
40          System.out.println("(5+10)*10= " + a4.doWork(5,10));
41
42          /*Lambda 5: Passing lambda as method parameter to Arrays.sort() method*/
43          String[] w = new String[]{"Hi", "Hello", "HelloWorld", "H"};
44          System.out.println("Original array= " + Arrays.toString(w));
45          Arrays.sort(w, (first, second) -> Integer.compare(first.length(), ↩
    second.length()));
46          System.out.println("Sorted array by length u
    toString(w));
47      }
48  }
```

```
Output - demo_lambda (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: 1 second)
```

Built-in functional interface [1/2]

- 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 interface [2/2]

```
12  import java.util.function.*;
13
14  public class FunctionalInterfacesDemo {
15      public static void main(String[] args) {
16          //test Predicate
17          Predicate<String> r1 = arg -> (arg.equals("Hello Lambda"));
18          String testStr = "Hello Lambda";
19          System.out.println(r1.test(testStr));
20
21          //test Consumer
22          Consumer<String> r2 = str -> System.out.println(str.toUpperCase());
23          r2.accept("hello lambda");
24
25          //test Function
26          Function<String, Integer> r3 = str -> str.length();
27          System.out.println(r3.apply("Hello Lambda"));
28
29          //test Supplier
30          Supplier r4 = () -> { return "Hello lambda  from supplier"; }
31          System.out.println(r4.get());
32      }
33  }
```

```
Output - demo_lambda (run)
run:
true
HELLO LAMBDA
12
Hello lambda  from supplier
BUILD SUCCESSFUL (total time: 0 seconds)
```

Primitive Versions of Functional Interfaces [1/2]

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 [2/2]

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

```
12  import java.util.function.*;
13
14  public class PrimitiveFunctionalInterfacesDemo {
15
16      public static void main(String[] args) {
17          //test IntPredicate
18          IntPredicate r1 = arg -> (arg == 10);
19          System.out.println("IntPredicate.test() result:  " + r1.test(11));
20
21          //test LongConsumer
22          LongConsumer r2 = val -> System.out.println("LongConsumer.accept() ↩
    result:  " + val * val);
23          r2.accept(1000000);
24      }
25  }
```

```
Output - demo_lambda (run)
run:
IntPredicate.test() result:  false
LongConsumer.accept() result:  1000000000000
BUILD SUCCESSFUL (total time: 4 seconds)
```

Binary Versions of Functional Interfaces

- 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.

```
12  import java.util.function.*;
13
14  public class BinaryFunctionalInterfacesDemo {
15
16      public static void main(String[] args) {
17          // test BiPredicate
18          BiPredicate<Integer, Integer> r1 = (arg1, arg2) -> arg1 < arg2;
19          System.out.println("BiPredicate.test() result:  " + r1.test(5, 10));
20
21          //test BiConsumer
22          BiConsumer<String, String> r2 = (arg1, arg2) -> System.out.println("↩
    BiConsumer.accept() result:  " + arg1 + arg2);
23          r2.accept("Hello ", "Lambda");
24      }
25  }
```

```
Output - demo_lambda (run)
run:
BiPredicate.test() result:  true
BiConsumer.accept() result:  Hello Lambda
BUILD SUCCESSFUL (total time: 2 seconds)
```

UnaryOperator Interface

- 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.

```
12  import java.util.function.*;
13
14    public class UnaryOperatorDemo {
15
16        public static void main(String[] args) {
17            UnaryOperator<String> result = (x) -> x.toUpperCase();
18            System.out.println("Output converted into uppercase:");
19            System.out.println(result.apply("Hello Lambda"));
20        }
21    }
```

```
Output - demo_lambda (run)
run:
Output converted into uppercase:
HELLO LAMBDA
BUILD SUCCESSFUL (total time: 0 seconds)
```

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 along side.

- 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:

```
12  public class MultiThreadedAnonymousDemo {
13      public static void main(String[] args) {
14
            Runnable r1 = new Runnable() {
16              @Override
                public void run() {
18                  for (int i = 0; i < 5; i++) {
19                      System.out.println("Hello from anonymous");
20                      try {
                            Thread.sleep(200);
22                      } catch (InterruptedException ex) {
                            ex.printStackTrace();
24                      }
25                  }
26              }
27          };
```

Refactoring for Improved Readability [3/3]

```
28
29          Runnable r2 = () -> {
30              for (int i = 0; i < 5; i++) {
31                  System.out.println("\tHello from lambda");
32                  try {
                        Thread.sleep(120);
                    } catch (Exception e) {
35                  }
36              }
37          };
38
39          Thread t1 = new Thread(r1);
40          Thread t2 = new Thread(r2);
41          t1.start();
42          t2.start();
43      }
44  }
```

```
Output - demo_lambda (run)
run:
Hello from anonymous
        Hello from lambda
        Hello from lambda
Hello from anonymous
        Hello from lambda
        Hello from lambda
Hello from anonymous
        Hello from lambda
Hello from anonymous
Hello from anonymous
BUILD SUCCESSFUL (total time: 2 seconds)
```

Refactoring Comparison Code [1/2]

- 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.
- The code applies a Comparator using an anonymous inner class

```
38    class Employee {
          private String firstName;
          private String lastName;
41
42        public Employee(String firstName, String lastName) {
43            this.firstName = firstName;
44            this.lastName = lastName;
45        }
46        public String getFirstName() {
47            return firstName;
48        }
49        public String getLastName() {
50            return lastName;
51        }
52    }
```

```
17        List<Employee> employeeList = new ArrayList<>();
18        employeeList.add(new Employee("Patrick", "Samuel"));
19        employeeList.add(new Employee("John", "Doe"));
20        employeeList.add(new Employee("Andy", "Davidson"));
21
          Collections.sort(employeeList, new Comparator<Employee>() {
23            @Override
              public int compare(Employee emp1, Employee emp2) {
25                return emp1.getLastName().compareTo(emp2.getLastName());
26            }
27        });
```

Refactoring Comparison Code [2/2]

- The code snippet applies a Comparator by using a lambda

```
14    public class ComparatorLambdaDemo {
15
16        public static void main(String[] args) {
17            List<Employee> employeeList = new ArrayList<>();
18            employeeList.add(new Employee("Patrick", "Samuel"));
19            employeeList.add(new Employee("John", "Doe"));
20            employeeList.add(new Employee("Andy", "Davidson"));
21
22            Comparator<Employee> sorted = (Employee e1, Employee e2) -> e1.↩
      getLastName().compareTo(e2.getLastName());
23
24            System.out.println("Sorted Employee by last name  in ascending order");
25            Collections.sort(employeeList, sorted);
              for (Employee emp : employeeList) {
27                System.out.println(emp.getFirstName() + " " + emp.getLastName());
28            }
29        }
30    }
```

```
Output - demo_lambda (run)
run:
Sorted Employee by last name  in ascending order
Andy Davidson
John Doe
Patrick Samuel
BUILD SUCCESSFUL (total time: 2 seconds)
```

Refactoring Concurrency Code [1/3]

- 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 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 [2/3]

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

```
22      ExecutorService executor = Executors.newFixedThreadPool(5);
23      List<Future<String>> list = new ArrayList<>();
        Callable callable = new Callable() {
25          @Override
            public String call() throws Exception {
27              try {
28                  Thread.sleep(10);
29                  System.out.println("aaa");
30                  return Thread.currentThread().getName();
31              } catch (InterruptedException ie) {
                    ie.printStackTrace();
33              }
34              return Thread.currentThread().getName();
35          }
36      };
37
38      for (int i = 0; i < 10; i++) {
39          Future<String> future = executor.submit(callable);
40          list.add(future);
41      }
```

Refactoring Concurrency Code [3/3]

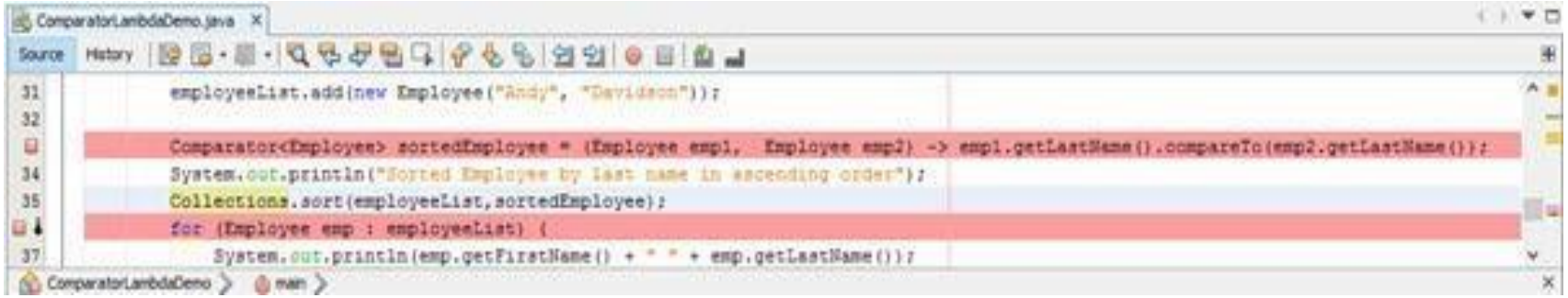
- The code snippet demonstrates using lambda to construct a Callable

```
22      ExecutorService executor = Executors.newFixedThreadPool(5);
23      List<Future<String>> list = new ArrayList<>();
        Callable callable = () -> {
25          try {
26              Thread.sleep(10);
27              return Thread.currentThread().getName();
28          } catch (InterruptedException ie) {
                ie.printStackTrace();
30          }
31          return Thread.currentThread().getName();
32      };
33      for (int i = 0; i < 10; i++) {
34          Future<String> future = executor.submit(callable);
35          list.add(future);
36      }
37      for (int i = 0; i<10; i++) {
38          try {
39              System.out.println(i+ ": "+list.get(i).get());
40          } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
42          }
43      }
44      executor.shutdown();
```

```
Output - demo_lambda (run)
run:
0: pool-1-thread-1
1: pool-1-thread-2
2: pool-1-thread-3
3: pool-1-thread-4
4: pool-1-thread-5
5: pool-1-thread-1
6: pool-1-thread-4
7: pool-1-thread-5
8: pool-1-thread-3
9: pool-1-thread-2
BUILD SUCCESSFUL (to
```

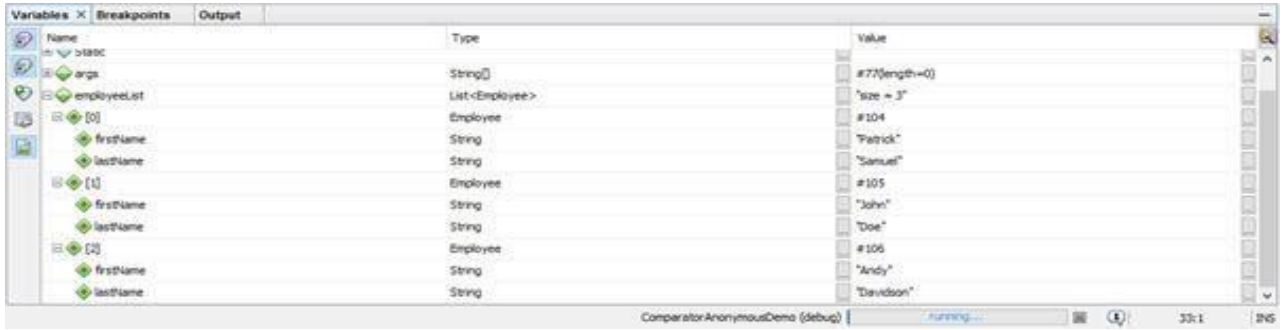
Debugging Lambdas [1/2]

- To test the lambda used in **`ComparatorLambdaDemo`** class :
  - 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.



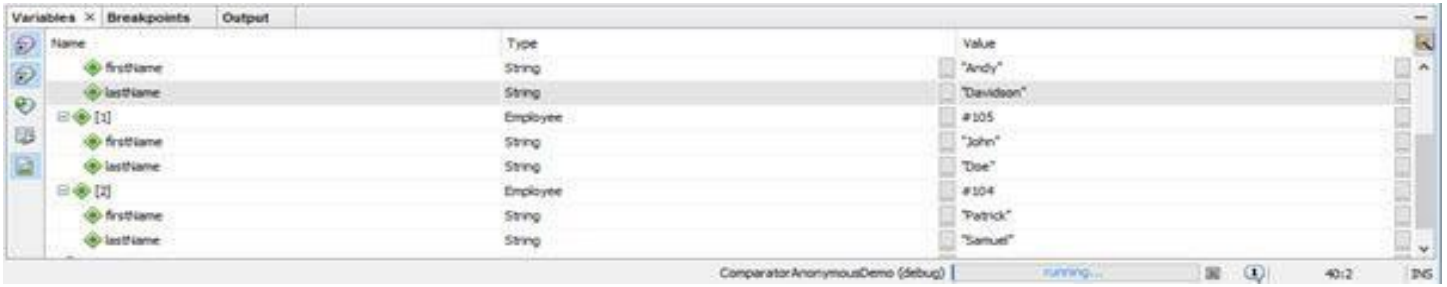
  - Select Debug ⍰ Debug Project from the main menu of NetBeans. The program execution stops in the first breakpoint.
  - Observe the first name and last name values in the Variables window displayed. At this point, the lambda is yet to perform the sorting.

Debugging Lambdas [2/2]



- Employee Values before Sorting

- Select Debug → Continue from 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

- 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.
- 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.