

Professional Programming in Java

Session: 15

**Effective Programming
with Lambda**





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



For Aptech Center Use Only



◆ Lambda expressions

Introduced in Java 8

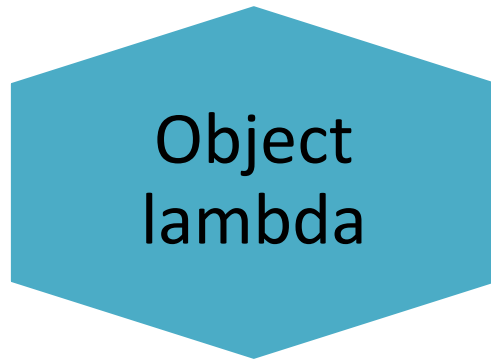
Unnamed blocks of code

Facilitate functional programming

For Aptech Center Use Only



- ◆ Types of lambda expressions are:



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

For Aptech Center Use Only



- ◆ 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 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 */
```

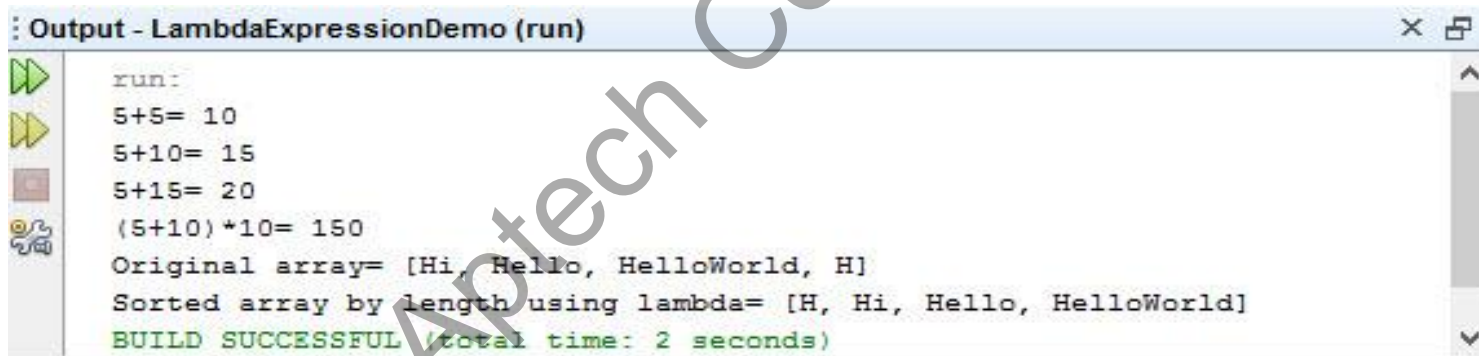


```
FunctionalA functionalA4 = (num1, num2) -> {  
    int sum = num1 + num2;  
    int result = sum * 10;  
    return result;  
};  
  
System.out.println("(5+10)*10= " +  
    functionalA4.doWork(5,  
    10));  
/*Lambda 5: Passing lambda as method parameter to  
Arrays.sort() method*/  
String[] words = new String[]{"Hi", "Hello",  
    "HelloWorld",  
    "H"};  
System.out.println("Original array= " +  
    Arrays.toString(words));  
Arrays.sort(words,
```



```
(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)
```




- ◆ 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.



- ◆ Following table lists functional interfaces:

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



- ◆ 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"));
    }

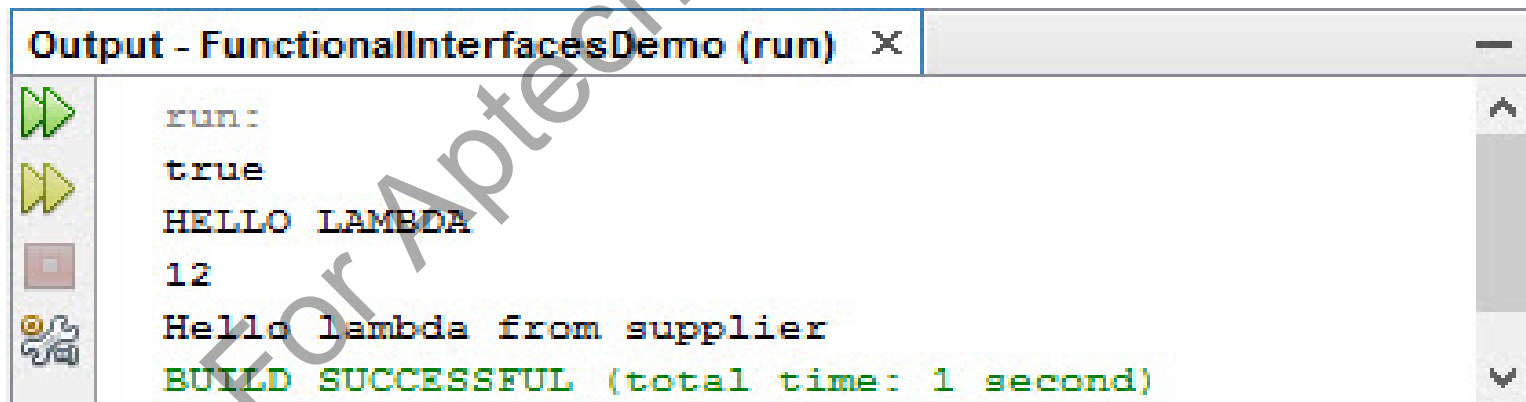
    static void testProducer() {
        Supplier result = () -> {return "Hello lambda
        from supplier";};
        System.out.println(result.get());
    }
}
```

Built-in Functional Interfaces [4-4]



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

Following is the output of the code:



```
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.



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:

A screenshot of an IDE's output window titled "Output - PrimitiveFunctionalInterfacesDemo (run)". The window shows the following text: "run:", "IntPredicate.test() result: false", "LongConsumer.accept() result: 1000000000000", and "BUILD SUCCESSFUL (total time: 1 second)". There are green arrow icons on the left side of the output lines.

```
run:
IntPredicate.test() result: false
LongConsumer.accept() result: 1000000000000
BUILD SUCCESSFUL (total time: 1 second)
```


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.





- ◆ 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:

A screenshot of an IDE's output window titled "Output - BinaryFunctionalInterfacesDemo (run)". The window contains the following text: "run:", "BiPredicate.test() result: true", "BiConsumer.accept() result: Hello Lambda", and "BUILD SUCCESSFUL (total time: 2 seconds)". On the left side of the window, there are three green arrow icons and a double right-pointing arrow icon. On the right side, there are up and down arrow icons for scrolling.

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



◆ 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.



- ◆ 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:

```
: Output - UnaryOperatorDemo (run)
>> Output converted into uppercase:
    HELLO LAMBDA
    BUILD SUCCESSFUL (total time: 2 seconds)
```



◆ 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.





- ◆ 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();
    }
}
```



- ◆ 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();
    }
}
```

For Aptech Center Use Only



◆ 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 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());
}
. . .
```



- ◆ 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;
```

For Aptech Center Use Only

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;
}
}

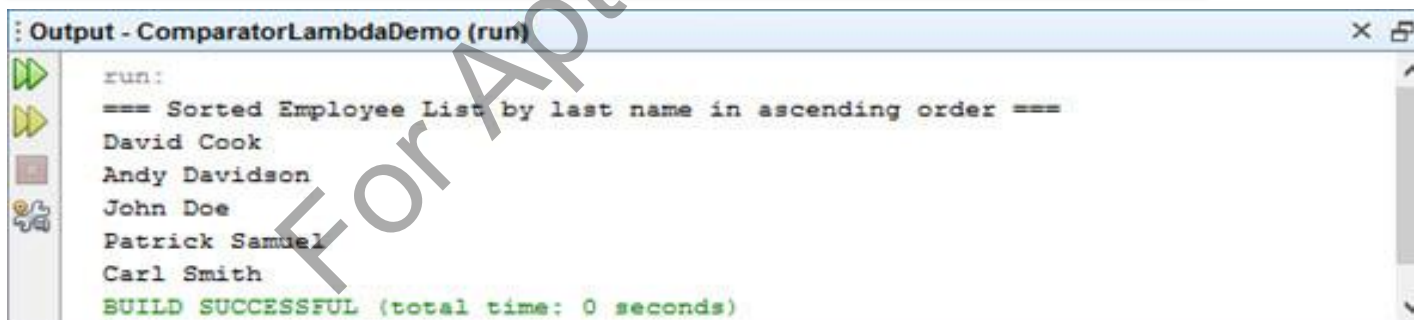
public class ComparatorLambdaDemo {
    public static void main(String[] args) {
        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(new Employee("Patrick", "Samuel"));
        employeeList.add(new Employee("John", "Doe"));
        employeeList.add(new Employee("Andy", "Davidson"));
        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:



```
Output - ComparatorLambdaDemo (run)
run:
=== Sorted Employee List by last name in ascending order ===
David Cook
Andy Davidson
John Doe
Patrick Samuel
Carl Smith
BUILD SUCCESSFUL (total time: 0 seconds)
```



- ◆ 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, `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.



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);
. . .
```



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]



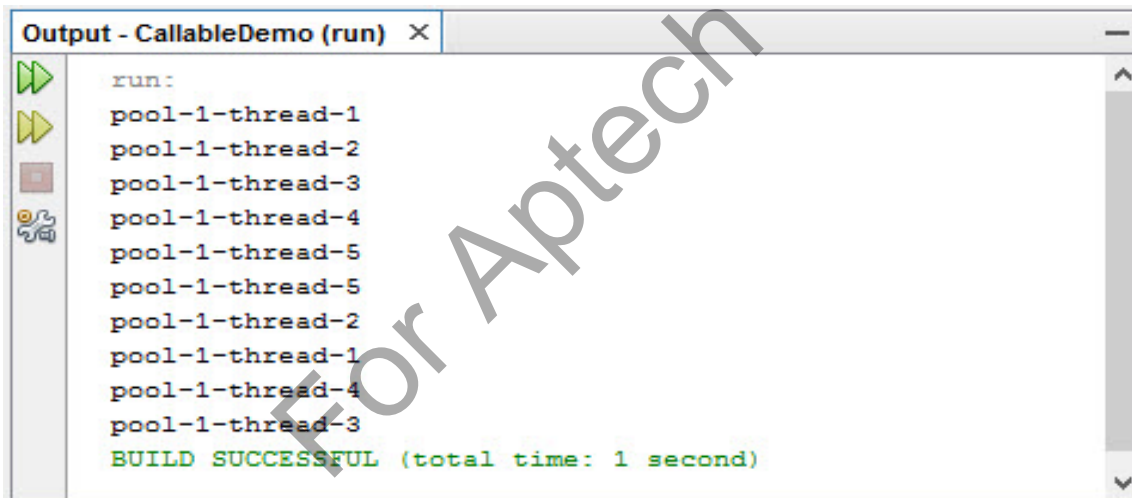
```
List<Future<String>> list = new ArrayList<>();
Callable callable = () -> {
    try {
        Thread.sleep(10);
        return Thread.currentThread().getName();
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
    return Thread.currentThread().getName();
};
for (int i = 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:



```
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)
```



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;
    }
}
```





```
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());
    }
}
```



```
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());  
}  
}  
}
```

For Aptech Center Use Only



◆ 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.



The screenshot shows the 'Variables' window with a tree view on the left and a table of values on the right. The tree view shows a list of employees with their first and last names. The table on the right shows the values of these variables.

Name	Type	Value
args	String[]	#77[length=0]
employeeList	List<Employee>	"size = 3"
[0]	Employee	#104
firstName	String	"Patrick"
lastName	String	"Samuel"
[1]	Employee	#105
firstName	String	"John"
lastName	String	"Doe"
[2]	Employee	#106
firstName	String	"Andy"
lastName	String	"Davidson"

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.

The screenshot shows the 'Variables' window with a tree view on the left and a table of values on the right. The tree view shows a list of employees with their first and last names. The table on the right shows the values of these variables.

Name	Type	Value
firstName	String	"Andy"
lastName	String	"Davidson"
[1]	Employee	#105
firstName	String	"John"
lastName	String	"Doe"
[2]	Employee	#104
firstName	String	"Patrick"
lastName	String	"Samuel"

Employee Values after Sorting

- Select Debug → Finish Debugger Session to stop debugging.



- ◆ 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.



For Aptech Center Use Only