Power Programming in Java

Session: 11

Effective Programming with Lambda

Objectives

- Describe lambda
- Describe local variable syntax for lambda parameters
- Identify the built-in functional interfaces
- Explain code refactoring for readability using lambdas
- Describe debugging of lambda
- Identify use of lambda expressions with user-defined functional interfaces

Lambda Usage

The lambda or simply lambda was introduced in Java 8

A lambda is a block of code, which takes parameters and returns a value like methods

Unlike other methods a lambda expression does not require a name

Lambda expressions are usually used to express instances of functional interface

Lambda Expression versus Anonymous Expression [1-2]

Syntax for using lambda expression having,

- Zero parameters is,

```
() -> body
```

- A single parameter and an expression is,

```
Parameter -> body
```

- To use more than one parameters,

```
(Parameter1, Parameter2) -> body
```

Lambda Expression versus Anonymous Expression [2-2]

Code Snippet

```
Button btn = new Button();
btn.setOnClick(new EventHandler<ButtonClick>() {
@Override
public void handle(ButtonClick event) {
 System.out.println("This is Java FX");
}
});
```

- The `EventHandler<ButtonClick>` interface can be replaced by lambda expression as,

```
Button btn = new Button();
btn.setOnClick(
event -> System.out.println("This is Java FX")
);
```

Lambda Types [1-4]

Based on how lambdas are written, lambdas can be categorized into two types:

- **Object Lambdas**: An object lambda implements a functional interface and stored in a variable to be used later.
- **Inline Lambdas**: Inline lambdas are passed inline as parameters to methods.
- The code creates a functional interface, named Fun_A. The code then uses it several lambdas.

Code Snippet

```
interface Fun_A {
int doWork(int a, int b);
}
/*Lambda 1: Using basic lambda */
Fun_A fun_A1 = (int num1, int num2) -> num1 + num2;
System.out.println("5+5= " + fun_A1.doWork(5, 5));

/*Lambda 2: Using lambda with inferred types */
Fun_A fun_A2 = (num1, num2) -> num1 + num2;
```

Lambda Types [2-4]

```
System.out.println("5+10= " + fun_A2 .doWork(5, 10));
/*Lambda 3: Using lambda with expression body containing
return statement */
Fun_A fun_A3 = (num1, num2) -> {
return num1 + num2;
};
System.out.println("5+11. " + fun_A3.doWork(5, 11.);


/*Lambda 4: Using lambda with expression body containing
multiple statements */
Fun_A fun_A4 = (num1, num2) -> {
int sum = num1 + num2;
int result = sum * 10; return result;
};
System.out.println("(5+10)*10= " + fun_A4.doWork(5, 10));
```

Lambda Types [3-4]

```
/* Lambda 5: Passing lambda as method parameter to
Arrays.sort() method */

String[] word = new String[]{"Hi", "Hey", "How", "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));
```

Lambda Types [4-4]

Capturing Lambda

- A variable defined in the **outer scope** can be used by lambda expressions are called **capturing lambdas**.
- They are capable of capturing different variables such as static, local, or instance variables, the local variable must be final or effectively final.

Code Snippet

```
Supplier<Integer> incrementer(int start) {
return () -> start++;
}
```

Local Variable Syntax for Lambda Parameters

- Lambda expressions are similar to functions.
- It is not ideal to use anonymous functions for simple implementations. Sometimes it may become cumbersome.
- In such situations lambda functions can be used, as they can express single method class instances more concisely.
- `var` is supported from Java 11 onwards.
- `var` helps to define a variable without explicitly specifying the types of parameters such as String.
  - `(var p, var q) -> p.print(q);`
  - p and q are lambda parameters but have been given local variable syntax.
- This makes the code more compact, easier, readable, and consistent.

Built-in Functional Interfaces

- Java 8 onwards includes a large number of built-in functional interfaces as part of `java.util.function` package.

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

Primitive Versions of Functional Interfaces [1-2]

- Functional interfaces `Predicate<T>`, `Consumer<T>`, `Function<T, R>`, and `Supplier<T>` are generic and therefore, operate on reference type objects.
- Primitive values, such as `int`, `long`, `float`, or `double` cannot be used with them.
- Therefore, Java provides primitive versions for such functional interfaces.
- For example, `IntPredicate`, `LongPredicate`, and `DoublePredicate` are primitive versions of the `Predicate` interface. Similarly, `IntConsumer`, `LongConsumer`, and `DoubleConsumer` are primitive versions of the `Consumer` interface.

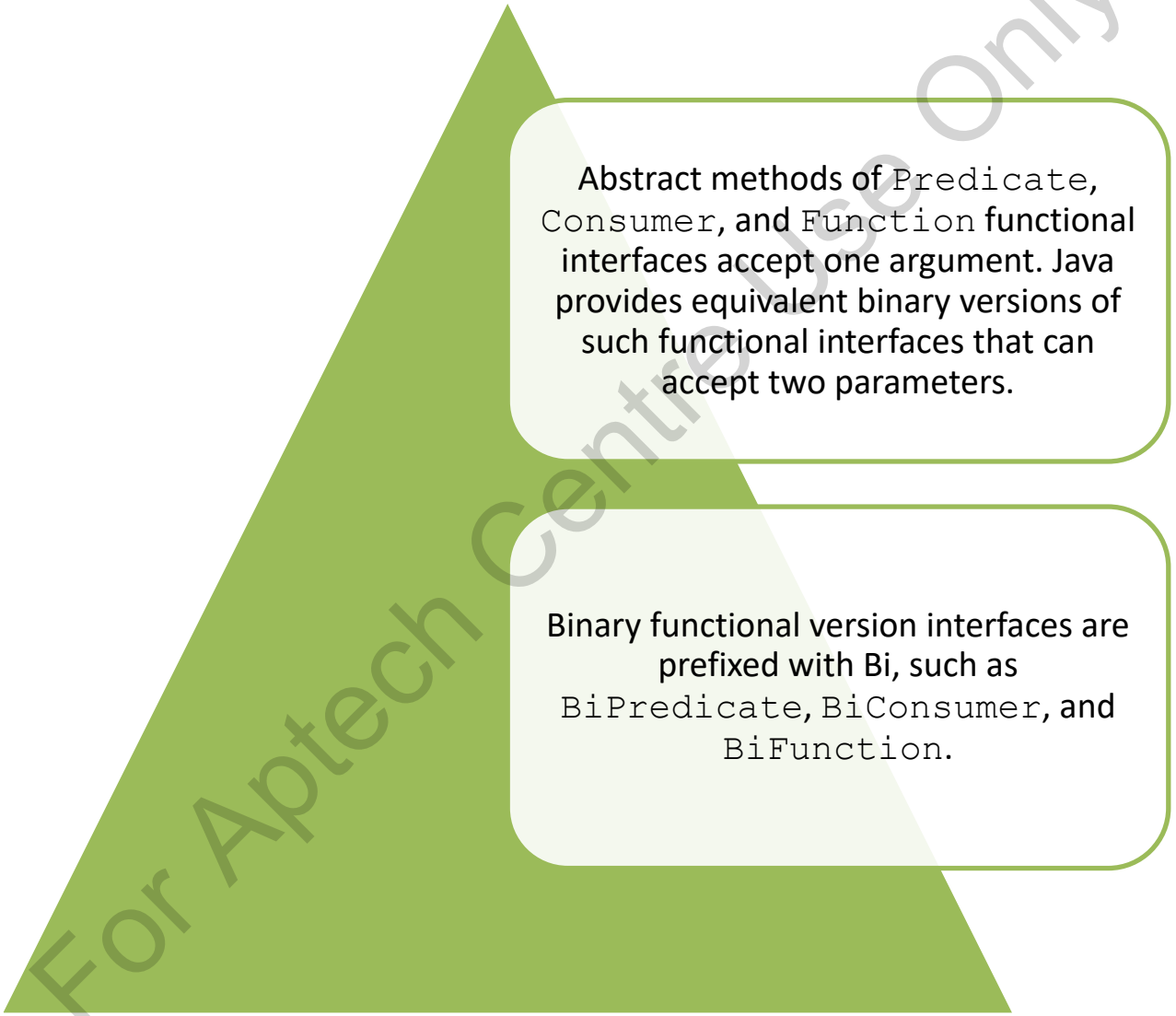
Primitive Versions of Functional Interfaces [2-2]

- Code Snippet shows the use of the primitive versions of the `Predicate` and `Consumer` functional interfaces.

Code Snippet

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

Binary Versions of Functional Interfaces [1-2]



Abstract methods of `Predicate`, `Consumer`, and `Function` functional interfaces accept one argument. Java provides equivalent binary versions of such functional interfaces that can accept two parameters.

Binary functional version interfaces are prefixed with Bi, such as `BiPredicate`, `BiConsumer`, and `BiFunction`.

Binary Versions of Functional Interfaces [2-2]

Code Snippet

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

UnaryOperator Interface

- The `java.util.function` package contains a `UnaryOperator` functional interface that is a specialized version of the `Function` interface. `UnaryOperator` can be used on a single operand when the types of the operand and result are the same.

Code Snippet

```
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"));
    }
}
```

Refactoring for Improved Readability

Lambdas can greatly increase readability of code.

Java programmers can use lambdas to express problems in many situations, in a shorter and more readable way than it was possible before.

The introduction of lambdas does not break code. Existing code can run as it is with new code containing lambdas running alongside.

However, developers might want to refactor their existing code to use the more convenient lambdas.

Typically, refactoring will be done to remove existing boilerplate code and make the existing code more concise.

Refactoring Runnable Code

- In multi-threaded applications, one way to implement a new thread is to create a `Runnable` object and call its `run()` object. Prior to Java 8, it was achieved through an anonymous class.

Code Snippet

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

- By using lambda, the same `Runnable` instance can be written as,

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

Refactoring Comparison Code [1-2]

- The `Comparator` interface enables comparing the elements of a collection that required to be sorted.
- `Comparator` is a functional interface that contains the single `int compare(T o1, T o2)` method.
- When a collection or array is to be sorted, a `Comparator` object is passed to the `Collections.sort()` or `Arrays.sort()` method.

Refactoring Comparison Code [2-2]

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 Concurrency Code

`Callable` and `Future` are extensively used in multi-threaded Java applications to implement asynchronous processing. They are functional interfaces in the `java.util.concurrent` package.

`Callable` is similar to `Runnable`. Both can be used to create a task which can be executed by threads in parallel.

Unlike `Runnable` that cannot return a value, `Callable` can return a value.

In addition, `call()` method of `Callable` can throw checked exception, which is not possible for `run()` method of `Runnable`.

`Callable` interface has a single `call()` abstract method. When a `Callable` is passed to a thread pool maintained by `ExecutorService`, the pool selects a thread and executes `Callable`.

Refactoring Concurrency Code

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);
. . .
```

Debugging Lambdas [1-5]

- Debugging lambda is similar to any other code in Java.

Code Snippet

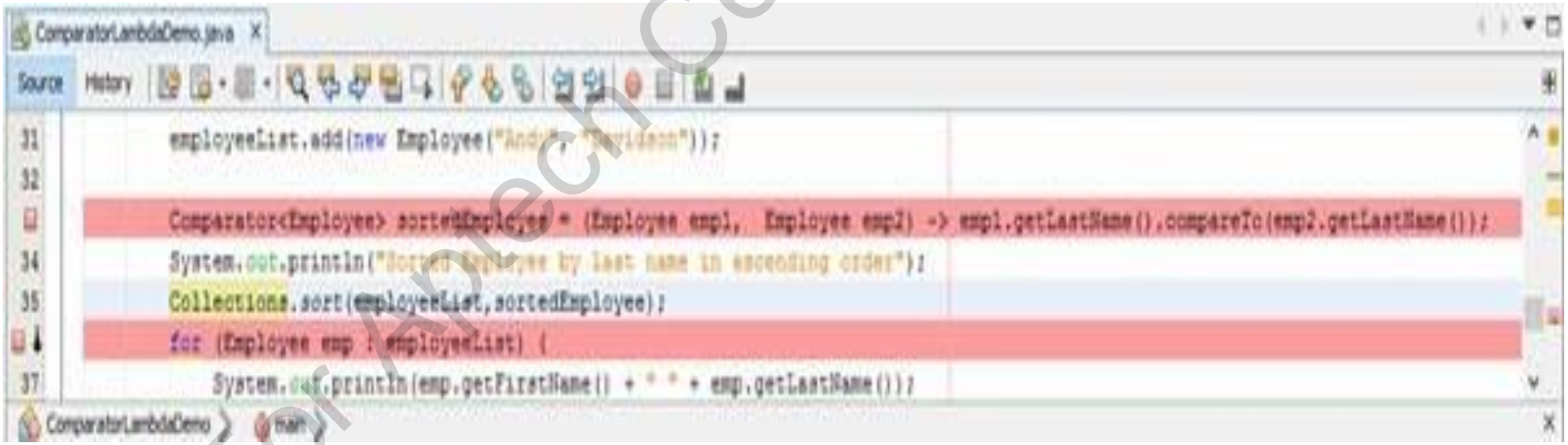
```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
class Employee{
private String firstName;
private String lastName;
public Employee(String firstName, String lastName) {
this.firstName = firstName;
this.lastName = lastName;
}
public String getFirstName() {
return firstName;
}
public String getLastName() {
return lastName;
}
}
```

Debugging Lambdas [2-5]

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

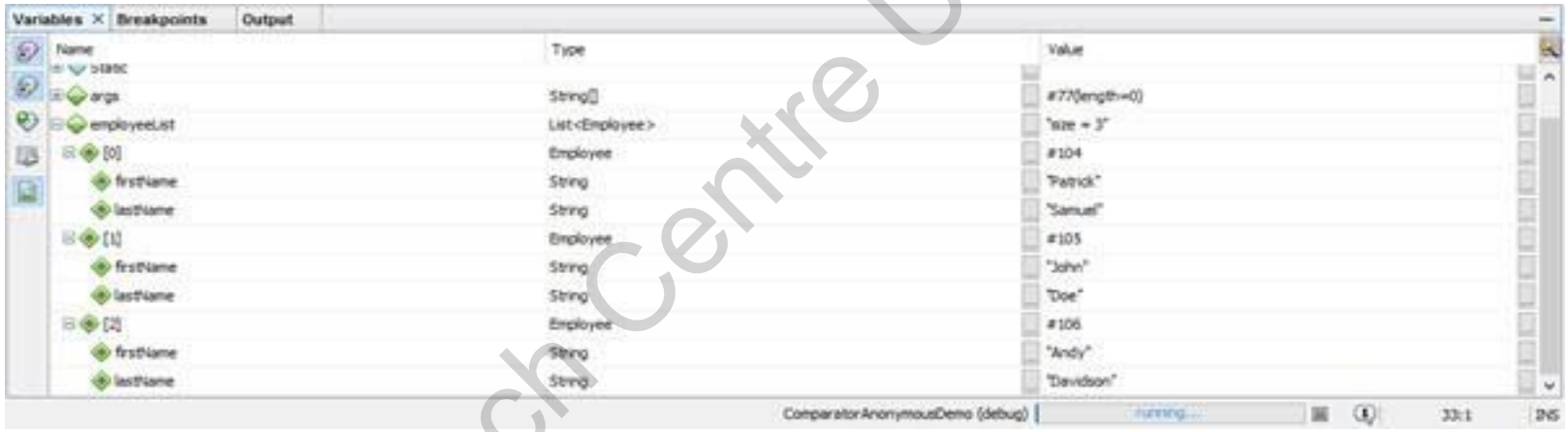
Debugging Lambdas [3-5]

- The code uses a lambda to compare two `Employee` objects based on the `lastName` field and stores it in a `Comparator` variable.
- To test this lambda code in Netbeans:
  - 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.



Debugging Lambdas [4-5]

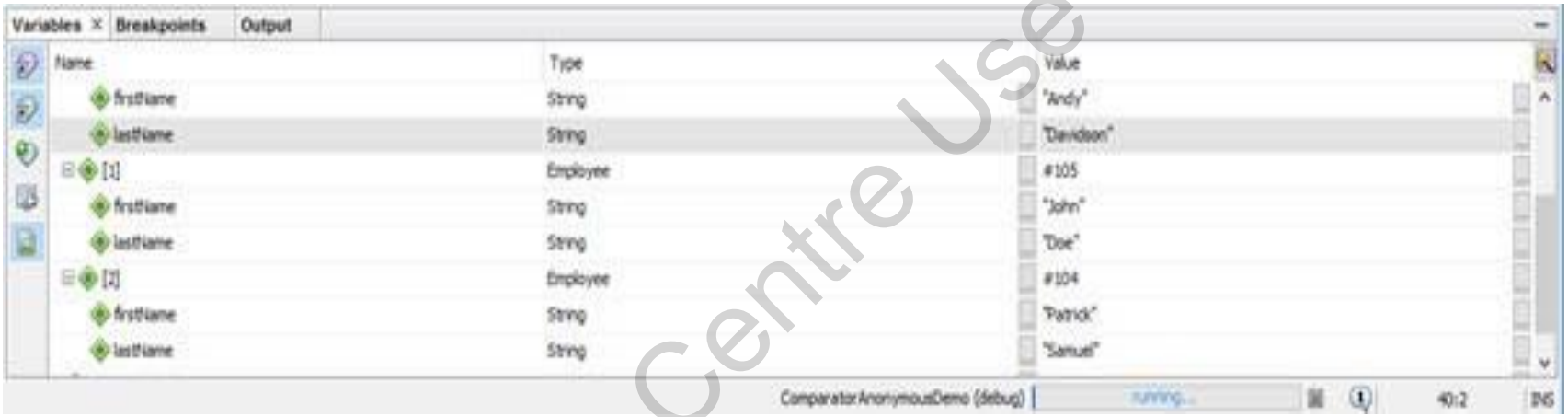
- Select **Debug → Debug Project** from the main menu of NetBeans. The program execution stops in the first breakpoint.
- Observe the first name and last name values in the **Variables** window displayed. At this point, the lambda is yet to perform the sorting.



- Select **Debug → Continue** from the main menu to continue debugging until the debugging thread hits the second breakpoint.

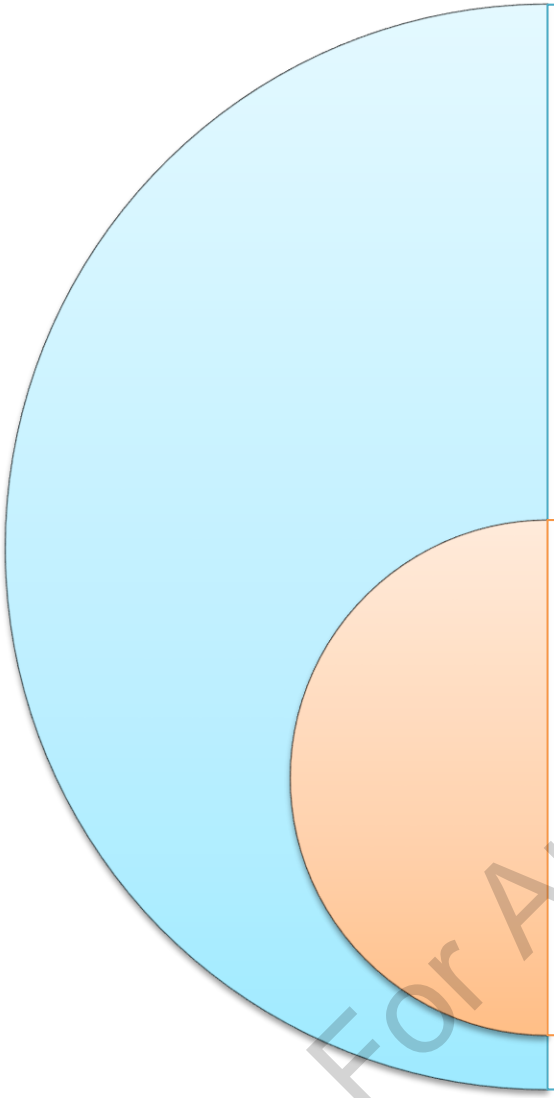
Debugging Lambdas [5-5]

- Check the **Variables** window to ensure that the lambda has correctly performed the sorting based on the last name.



- Select **Debug → Finish Debugger Session** to stop debugging.

Lambda with User-defined Functional Interfaces [1-2]



Lambda expressions can also be used to implement an instance containing functional interfaces.

Lambda expressions provide functionalities such as creating a function that does not belong to any class and passing around a lambda expression in the form of an object and executing it on demand.

Lambda with User-defined Functional Interfaces [2-2]

Code Snippet

```
// User-defined functional interface
interface FuncInterface {
    // An abstract method
    void abstractFun(int x);
    // A non-abstract (or default) method
    default void normalFun() {
    System.out.println("Hello");}}
    public class Test{
    public static void main(String args[])    {
    /* lambda expression to implement user-defined functional
    interface created earlier. This interface, by default,
    implements abstractFun() */
    FuncInterface fobj = (int x)->System.out.println(3*x);
    // This statement calls the lambda expression and prints 18.
    fobj.abstractFun(6);
    }
}
```

Interfaces with Both Default and Static Methods

From Java 8 onwards, there can be an interface with both default and static methods.

Both of these can be implemented directly while declaring the interface.

Therefore, a Java lambda expression can implement interfaces with more than one method, provided the interface only has a single unimplemented (that is, abstract) method.

Summary

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