A Brief Introduction to Lambda Expressions in Java 8

Short-ish Version

November 2017

Contents

1 Introduction

These notes introduce the features of the new lambda expressions which is included in Java Platform Standard Edition 8 (Java SE 8).

Lambda expressions are a new and important feature included in Java SE 8. They provide a way to represent one method interface using an expression. Lambda expressions also improve the `Collection` libraries making it easier to iterate through, filter, and extract data from a `Collection`. In addition, new concurrency features improve performance in multicore environments.

An introduction to *anonymous inner functions* is provided, followed by a discussion of functional interfaces and the new lambda syntax. Then, examples of common usage patterns before and after lambda expressions are discussed.

In addition, some of the common functional interfaces, `Predicate` and `Function`, provided in the package `java.util.function` are described. We finish with a review of how the Java collection has been updated with lambda expressions.

To follow these notes you will need to download code from the repository as some of the sections refer to that code. You will find the sources under the `Lambdas` subdirectory.

2 Background

In this section we consider how Java code can be improved with the inclusion of lambda expressions.

2.1 Anonymous Inner Classes — AIC

In Java, anonymous inner classes provide a way to implement classes that may occur only once in an application. For example, in a standard Swing or JavaFX application

a number of event handlers are required for keyboard and mouse events. Rather than writing a separate event-handling class for each event, you can write something like this.

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
  @Override public void actionPerformed(ActionEvent ae){
    System.out.println("Click Detected by Anon Class");
  }
});
```

Otherwise, a separate class that implements `ActionListener` is required for each event. By creating the class in place, where it is needed, the code is a little easier to read. The code is not elegant, because quite a bit of code is still required just to define one method.

2.2 Functional Interfaces

The code that defines the `ActionListener` interface, looks something like this:

```
package java.awt.event;
import java.util.EventListener;

public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

The `ActionListener` example is an interface with only one method. With Java SE 8, an interface that follows this pattern is known as a *functional interface*[1].

Using functional interfaces with anonymous inner classes are a common pattern in Java. In addition to the `EventListener` classes, interfaces like `Runnable` and `Comparator` are used in a similar manner. Therefore, functional interfaces are leveraged for use with lambda expressions.

2.3 Lambda Expression Syntax

Lambda expressions address the bulkiness of anonymous inner classes by converting five lines of code into a single statement. This simple horizontal solution solves the "vertical problem" presented by inner classes.

A lambda expression is composed of three parts:

Argument List	Arrow Symbol	Body
`(int x, int y)`	`->`	`x + y`

[1]This type of interface, was previously known as a *Single Abstract Method* type (SAM).

The body can be either a single expression or a statement block. In the expression form, the body is simply evaluated and returned. In the block form, the body is evaluated like a method body and a `return` statement returns control to the caller of the anonymous method. The `break` and `continue` keywords are illegal at the top level, but are permitted within loops. If the body produces a result, every control path must return something or throw an exception.

Take a look at these examples:

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```

The first expression takes two integer arguments, named `x` and `y`, and uses the expression form to return `x+y`. The second expression takes no arguments and uses the expression form to return an integer 42. The third expression takes a string and uses the block form to print the string to the console, and returns nothing.

With the basics of syntax covered, let's look at some examples.

3 Lambda Examples

3.1 Runnable Lambda

Here is how you write a `Runnable` using lambdas (see the file):

RunnableTest.java

```
public class RunnableTest {
    public static void main(String[] args) {

        System.out.println("=== RunnableTest ===");

        // Anonymous Runnable
        Runnable r1 = new Runnable() {

            @Override
            public void run() {
                System.out.println("Hello world one!");
            }
        };

        // Lambda Runnable
        Runnable r2 = () -> System.out.println("Hello world two!");

        // Run em!
        r1.run();
        r2.run();
```

```
21
22      }
23  }
```

In both cases, notice that no parameter is passed and is returned. The `Runnable` lambda expression, which uses the block format, converts five lines of code into one statement.

3.2 Comparator Lambda

In Java, the `Comparator` class is used for sorting collections. In the following example, an `ArrayList` consisting of `Person` objects is sorted based on the value of the `surName` field. The following are the fields included in the `Person` class.

```
1  public class Person {
2     private String givenName;
3     private String surName;
4     private int age;
5     private Gender gender;
6     private String eMail;
7     private String phone;
8     private String address;
```

The following code applies a `Comparator` by using an anonymous inner class and a couple lambda expressions (see the file):

ComparatorTest.java

```
1  import java.util.Collections;
2  import java.util.Comparator;
3  import java.util.List;
4
5  public class ComparatorTest {
6
7      public static void main(String[] args) {
8
9          List<Person> personList = Person.createShortList();
10
11         // Sort with Inner Class
12         Collections.sort(personList, new Comparator<Person>() {
13             public int compare(Person p1, Person p2) {
14                 return p1.getSurName().compareTo(p2.getSurName());
15             }
16         });
17
18         System.out.println("=== Sorted Asc SurName ===");
19         for (Person p : personList) {
20             p.printName();
21         }
22
23         // Use Lambda instead
```

```
24
25          // Print Asc
26          System.out.println("=== Sorted Asc SurName ===");
27          Collections.sort(personList, (Person p1, Person p2)
28                  -> p1.getSurName().compareTo(p2.getSurName()));
29
30          for (Person p : personList) {
31              p.printName();
32          }
33
34          // Print Desc
35          System.out.println("=== Sorted Desc SurName ===");
36          Collections.sort(personList, (p1, p2)
37                  -> p2.getSurName().compareTo(p1.getSurName()));
38
39          for (Person p : personList) {
40              p.printName();
41          }
42
43      }
44  }
```

Lines 15–19 are easily replaced by the lambda expression on line 30. Notice that the first lambda expression declares the parameter type passed to the expression. However, as you can see from the second expression, this is optional. Lambda supports "target typing" which infers the object type from the context in which it is used. As we are assigning the result to a `Comparator` defined with a generic, the compiler can infer that the two parameters are of the `Person` type.

3.3 Listener Lambda

Finally, let's revisit the ActionListener example (see the file).

ListenerTest.java

```
1   import javax.swing.JButton;
2   import javax.swing.JFrame;
3   import java.awt.BorderLayout;
4   import java.awt.event.ActionEvent;
5   import java.awt.event.ActionListener;
6
7   public class ListenerTest {
8       public static void main(String[] args) {
9
10          JButton testButton = new JButton("Test Button");
11          testButton.addActionListener(new ActionListener() {
12              @Override
13              public void actionPerformed(ActionEvent ae) {
14                  System.out.println("Click Detected by Anon Class");
15              }
16          });
```

```
17
18          testButton.addActionListener(e
19                  -> System.out.println("Click Detected by Lambda Listener"));
20
21          // Swing stuff
22          JFrame frame = new JFrame("Listener Test");
23          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24          frame.add(testButton, BorderLayout.CENTER);
25          frame.pack();
26          frame.setVisible(true);
27
28      }
29  }
```

Notice that the lambda expression is passed as a parameter. Target typing is used in a number of contexts including the following:

- Variable declarations
- Assignments
- `return` statements
- Array initialisers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions `?:`
- Cast expressions

4 Improving Code with Lambda Expressions

This section builds upon the previous examples to show you how lambda expressions can improve your code. Lambdas should provide a means to better support the *Don't Repeat Yourself* (DRY) principle and make your code simpler and more readable.

4.1 A Common Query example

A common use case for programs is to search through a collection of data to find items that match a specific criteria. Given a list of people, various criteria are used to make *robo calls* (automated phone calls) to matching persons.

In this example, our message needs to get out to three different groups (based on an example presented at the JavaOne 2012 conference):

Drivers: Persons over the age of 16

Draftees: Male persons between the ages of 18 and 25

Pilots (specifically commercial pilots): Persons between the ages of 23 and 65

The actual robot that does all this work has not yet arrived at our place of business. Instead of calling, mailing or emailing, a message is printed to the console. The message contains a person's name, age, and information specific to the target medium (for example, email address when emailing or phone number when calling).

`Person` class

Each person in the test list is defined by using the `Person` class with the following properties:

```
public class Person {
   private String givenName;
   private String surName;
   private int age;
   private Gender gender;
   private String eMail;
   private String phone;
   private String address
```

The `Person` class uses a `Builder` to create new objects. A sample list of people is created with the `createShortList` method. Here is a short code fragment of that method.

```
public static List<Person> createShortList(){
     List<Person> people = new ArrayList<>();

     people.add(
       new Person.Builder()
             .givenName("Bob")
             .surName("Baker")
             .age(21)
             .gender(Gender.MALE)
             .email("bob.baker@example.com")
             .phoneNumber("201-121-4678")
             .address("44 4th St, Smallville, KS 12333")
             .build()
       );

     people.add(
       new Person.Builder()
             .givenName("Jane")
             .surName("Doe")
             .age(25)
             .gender(Gender.FEMALE)
             .email("jane.doe@example.com")
             .phoneNumber("202-123-4678")
```

```
            .address("33 3rd St, Smallville, KS 12333")
            .build()
      );

    people.add(
      new Person.Builder()
            .givenName("John")
            .surName("Doe")
            .age(25)
            .gender(Gender.MALE)
            .email("john.doe@example.com")
            .phoneNumber("202-123-4678")
            .address("33 3rd St, Smallville, KS 12333")
            .build()
    );
```

4.2 A First Attempt

With a `Person` class and search criteria defined, you can write a `RoboContact` class. One possible solution defines a method for each use case (see the file):

RoboContactMethods.java

```
import java.util.List;

public class RoboContactMethods {

    public void callDrivers(List<Person> pl) {
        for (Person p : pl) {
            if (p.getAge() >= 16) {
                roboCall(p);
            }
        }
    }

    public void emailDraftees(List<Person> pl) {
        for (Person p : pl) {
            if (p.getAge() >= 18 && p.getAge() <= 25
                    && p.getGender() == Gender.MALE) {
                roboEmail(p);
            }
        }
    }

    public void mailPilots(List<Person> pl) {
        for (Person p : pl) {
            if (p.getAge() >= 23 && p.getAge() <= 65) {
                roboMail(p);
            }
        }
    }
```

```
29
30
31      public void roboCall(Person p) {
32          System.out.println("Calling " + personInfo(p)
33                  + " at " + p.getPhone());
34      }
35
36      public void roboEmail(Person p) {
37          System.out.println("EMailing " + personInfo(p)
38                  + " at " + p.getEmail());
39      }
40
41      public void roboMail(Person p) {
42          System.out.println("Mailing " + personInfo(p)
43                  + " at " + p.getAddress());
44      }
45
46      private String personInfo(Person p) {
47          return p.getGivenName() + " " + p.getSurName()
48                  + " age " + p.getAge();
49      }
50
51  }
```

As you can see from the names (`callDrivers`, `emailDraftees`, and `mailPilots`) the methods describe the kind of behaviour that is taking place. The search criteria is clearly conveyed and an appropriate call is made to each robo action. However, this design has some negatives aspects:

- The DRY principle is not followed.
  - Each method repeats a looping mechanism.
  - The selection criteria must be rewritten for each method.
- A large number of methods are required to implement each use case.
- The code is inflexible. If the search criteria changed, it would require a number of code changes for an update. Thus, the code is not very maintainable.

4.3 Refactor the Methods

How can the class be fixed? The search criteria is a good place to start. If test conditions are isolated in separate methods, that would be an improvement (see the file).

RoboContactMethods2.java

```
1   import java.util.List;
2
3   public class RoboContactMethods2 {
4
```

```
5       public void callDrivers(List<Person> pl) {
6           for (Person p : pl) {
7               if (isDriver(p)) {
8                   roboCall(p);
9               }
10          }
11      }
12
13      public void emailDraftees(List<Person> pl) {
14          for (Person p : pl) {
15              if (isDraftee(p)) {
16                  roboEmail(p);
17              }
18          }
19      }
20
21      public void mailPilots(List<Person> pl) {
22          for (Person p : pl) {
23              if (isPilot(p)) {
24                  roboMail(p);
25              }
26          }
27      }
28
29      public boolean isDriver(Person p) {
30          return p.getAge() >= 16;
31      }
32
33      public boolean isDraftee(Person p) {
34          return p.getAge() >= 18
35                  && p.getAge() <= 25 && p.getGender() == Gender.MALE;
36      }
37
38      public boolean isPilot(Person p) {
39          return p.getAge() >= 23
40                  && p.getAge() <= 65;
41      }
42
43      public void roboCall(Person p) {
44          System.out.println("Calling " + personInfo(p)
45                  + " at " + p.getPhone());
46      }
47
48      public void roboEmail(Person p) {
49          System.out.println("EMailing " + personInfo(p)
50                  + " at " + p.getEmail());
51      }
52
53      public void roboMail(Person p) {
54          System.out.println("Mailing " + personInfo(p)
55                  + " at " + p.getAddress());
```

```
56      }
57
58      private String personInfo(Person p) {
59          return p.getGivenName() + " " + p.getSurName()
60                  + " age " + p.getAge();
61      }
62  }
```

The search criteria are encapsulated in a method, an improvement over the previous example. The test conditions can be reused and changes flow back throughout the class. However there is still a lot of repeated code and a separate method is still required for each use case. Is there a better way to pass the search criteria to the methods?

4.4 Anonymous Classes

Before lambda expressions, anonymous inner classes were an option. For example, an interface (`MyTest.java`) written with one `test` method that returns a boolean (a functional interface) is a possible solution. The search criteria could be passed when the method is called. The interface looks like this:

```
1  public interface MyTest<T> {
2    public boolean test(T t);
3  }
```

The updated robot class looks like (see the file):

RoboContactAnon.java

```
1  import java.util.List;
2
3  public class RoboContactAnon {
4
5      public void phoneContacts(List<Person> pl, MyTest<Person> aTest) {
6          for (Person p : pl) {
7              if (aTest.test(p)) {
8                  roboCall(p);
9              }
10         }
11     }
12
13     public void emailContacts(List<Person> pl, MyTest<Person> aTest) {
14         for (Person p : pl) {
15             if (aTest.test(p)) {
16                 roboEmail(p);
17             }
18         }
19     }
20
21     public void mailContacts(List<Person> pl, MyTest<Person> aTest) {
22         for (Person p : pl) {
```

```
            if (aTest.test(p)) {
                roboMail(p);
            }
        }
    }

    public void roboCall(Person p) {
        System.out.println("Calling " + personInfo(p)
                + " at " + p.getPhone());
    }

    public void roboEmail(Person p) {
        System.out.println("EMailing " + personInfo(p)
                + " at " + p.getEmail());
    }

    public void roboMail(Person p) {
        System.out.println("Mailing " + personInfo(p)
                + " at " + p.getAddress());
    }

    private String personInfo(Person p) {
        return p.getGivenName() + " " + p.getSurName()
                + " age " + p.getAge();
    }
}
```

That is definitely another improvement, because only three methods are needed to perform robotic operations. However, there is a slight problem with ugliness when the methods are called. Consider the test class used for this class (see the file):

RoboCallTest03.java

```
import java.util.List;

public class RoboCallTest03 {

    public static void main(String[] args) {

        List<Person> pl = Person.createShortList();
        RoboContactAnon robo = new RoboContactAnon();

        System.out.println("\n==== Test 03 ====");
        System.out.println("\n=== Calling all Drivers ===");
        robo.phoneContacts(pl,
                new MyTest<Person>() {
                    @Override
                    public boolean test(Person p) {
                        return p.getAge() >= 16;
                    }
                }
        );
```

```
20
21          System.out.println("\n=== Emailing all Draftees ===");
22          robo.emailContacts(pl,
23                  new MyTest<Person>() {
24                      @Override
25                      public boolean test(Person p) {
26                          return p.getAge() >= 18 && p.getAge() <= 25
27                                  && p.getGender() == Gender.MALE;
28                      }
29                  }
30          );
31
32
33          System.out.println("\n=== Mail all Pilots ===");
34          robo.mailContacts(pl,
35                  new MyTest<Person>() {
36                      @Override
37                      public boolean test(Person p) {
38                          return p.getAge() >= 23 && p.getAge() <= 65;
39                      }
40                  }
41          );
42
43
44      }
45  }
```

This is an example of the "vertical" problem in practice. This code is a little difficult to read. In addition, we have to write custom search criteria for each use case[2].

4.5 Lambda Expressions

Lambda expressions solve all the problems encountered so far but first we need to consider some provided methods.

`java.util.function`

In the previous example, the `MyTest` functional interface passed anonymous classes to methods. However, writing that interface was not necessary. Java SE 8 provides the `java.util.function` package with a number of standard functional interfaces. In this case, the `Predicate` interface meets our needs.

```
1  public interface Predicate<T> {
2    public boolean test(T t);
3  }
```

[2]If you are using an IDE it might well indicate that the AIC could be replaced by a lambda expression!

The `test` method takes a generic class and returns a boolean result. This is just what is needed to make selections. Here is the final version of the robot class (see the file).

RoboContactLambda.java

```
import java.util.List;

public class RoboContactLambda {
    public void phoneContacts(List<Person> pl, Predicate<Person> pred) {
        for (Person p : pl) {
            if (pred.test(p)) {
                roboCall(p);
            }
        }
    }

    public void emailContacts(List<Person> pl, Predicate<Person> pred) {
        for (Person p : pl) {
            if (pred.test(p)) {
                roboEmail(p);
            }
        }
    }

    public void mailContacts(List<Person> pl, Predicate<Person> pred) {
        for (Person p : pl) {
            if (pred.test(p)) {
                roboMail(p);
            }
        }
    }

    public void roboCall(Person p) {
        System.out.println("Calling " + personInfo(p)
                + " at " + p.getPhone());
    }

    public void roboEmail(Person p) {
        System.out.println("EMailing " + personInfo(p)
                + " at " + p.getEmail());
    }

    public void roboMail(Person p) {
        System.out.println("Mailing " + personInfo(p)
                + " at " + p.getAddress());
    }

    private String personInfo(Person p) {
        return p.getGivenName() + " " + p.getSurName()
                + " age " + p.getAge();
    }

}
```

With this approach only three methods are needed, one for each contact method. The lambda expression passed to the method selects the `Person` instances that meet the test conditions.

4.6 Vertical Problem Solved

Lambda expressions solve the vertical problem and allow the easy reuse of any expression. Consider our new test class updated for lambda expressions (see the file).

RoboCallTest04.java

```
import java.util.List;

public class RoboCallTest04 {

    public static void main(String[] args) {

        List<Person> pl = Person.createShortList();
        RoboContactLambda robo = new RoboContactLambda();

        // Predicates
        Predicate<Person> allDrivers = p -> p.getAge() >= 16;
        Predicate<Person> allDraftees = p -> p.getAge() >= 18
                && p.getAge() <= 25 && p.getGender() == Gender.MALE;
        Predicate<Person> allPilots =
                p -> p.getAge() >= 23 && p.getAge() <= 65;

        System.out.println("\n==== Test 04 ====");
        System.out.println("\n=== Calling all Drivers ===");
        robo.phoneContacts(pl, allDrivers);

        System.out.println("\n=== Emailing all Draftees ===");
        robo.emailContacts(pl, allDraftees);

        System.out.println("\n=== Mail all Pilots ===");
        robo.mailContacts(pl, allPilots);

        // Mix and match becomes easy
        System.out.println("\n=== Mail all Draftees ===");
        robo.mailContacts(pl, allDraftees);

        System.out.println("\n=== Call all Pilots ===");
        robo.phoneContacts(pl, allPilots);

    }
}
```

Notice that a `Predicate` is set up for each group: `allDrivers`, `allDraftees`, and `allPilots`. You can pass any of these `Predicate` interfaces to the contact methods. The code is compact and easy to read, and it is not repetitive.

5 The `java.util.function` Package

`Predicate` is not the only functional interface provided with Java SE 8. A number of standard interfaces are designed as a starter set for developers:

`Predicate`: A property of the object passed as argument

`Consumer`: An action to be performed with the object passed as argument

`Function`: Transform a `T` to a `U`

`Supplier`: Provide an instance of a `T` (such as a factory)

`UnaryOperator`: A unary operator from `T -> T`

`BinaryOperator`: A binary operator from `(T, T) -> T`

In addition, many of these interfaces also have versions that operate on primitive types.

5.1 Method References

Now let us add a flexible printing system to the `Person` class. One feature requirement is to display names in both a western style and an eastern style. In the West, names are displayed with the given name first and the surname second. In many eastern cultures, names are displayed with the surname first and the given name second.

5.2 An Old Style Approach

Here is an example of how to implement a `Person` printing class without lambda support.

```
1  public void printWesternName(){
2    System.out.println("\nName: " + this.getGivenName() + " "
3          + this.getSurName() + "\n" +
4               "Age: " + this.getAge() + "  " + "Gender: "
5                 + this.getGender() + "\n" +
6               "EMail: " + this.getEmail() + "\n" +
7               "Phone: " + this.getPhone() + "\n" +
8               "Address: " + this.getAddress());
9  }
10
11 public void printEasternName(){
12   System.out.println("\nName: " + this.getSurName() + " "
13         + this.getGivenName() + "\n" +
14              "Age: " + this.getAge() + "  " + "Gender: "
15                + this.getGender() + "\n" +
16              "EMail: " + this.getEmail() + "\n" +
17              "Phone: " + this.getPhone() + "\n" +
18              "Address: " + this.getAddress());
19 }
```

A method exists for each style that prints out a person.

5.3 The `Function` Interface

The `Function` interface is useful for solving this problem. It has only one method `apply` with the following signature:

```
public R apply(T t){}
```

It takes a generic class `T` and returns a generic class `R`. For this example, we will pass the `Person` class and return a `String`. A more flexible print method for person could be written as follows:

```
  public String printCustom(Function<Person,String> f){
    return f.apply(this);
  }
```

That is quite a bit simpler. A `Function` is passed to the method and a string returned. The `apply` method processes a lambda expression which determines what `Person` information is returned. How are the `Functions` defined? Here is the test code that calls the previous method (see the file).

NameTestNew.java

```
import java.util.List;
import java.util.function.Function;

public class NameTestNew {

    public static void main(String[] args) {

        System.out.println("\n==== NameTestNew ===");

        List<Person> list1 = Person.createShortList();

        // Print Custom First Name and e-mail
        System.out.println("===Custom List===");
        for (Person person : list1) {
            System.out.println(
                    person.printCustom(p
                            -> "Name: " + p.getGivenName()
                            + " EMail: " + p.getEmail())
            );
        }

        // Define Western and Eastern Lambdas

        Function<Person, String> westernStyle = p -> {
            return "\nName: " + p.getGivenName() + " "
                    + p.getSurName() + "\n" +
                    "Age: " + p.getAge() + "  " + "Gender: "
                    + p.getGender() + "\n" +
                    "EMail: " + p.getEmail() + "\n" +
```

```
                    "Phone: " + p.getPhone() + "\n" +
                    "Address: " + p.getAddress();
        };

        Function<Person, String> easternStyle = p -> "\nName: "
                + p.getSurName() + " "
                + p.getGivenName() + "\n" + "Age: " + p.getAge() + "  " +
                "Gender: " + p.getGender() + "\n" +
                "EMail: " + p.getEmail() + "\n" +
                "Phone: " + p.getPhone() + "\n" +
                "Address: " + p.getAddress();

        // Print Western List
        System.out.println("\n===Western List===");
        for (Person person : list1) {
            System.out.println(
                    person.printCustom(westernStyle)
            );
        }

        // Print Eastern List
        System.out.println("\n===Eastern List===");
        for (Person person : list1) {
            System.out.println(
                    person.printCustom(easternStyle)
            );
        }


    }
}
```

The first loop just prints the given name and the email address. Any valid expression could be passed to the `printCustom` method. Eastern and western print styles are defined with lambda expressions and stored in a variable. The variables are then passed to the final two loops. The lambda expressions could very easily be incorporated into a `Map` to make their reuse much easier. Here the lambda expression provides a great deal of flexibility.

Run the program to illustrate its flexibility.

6 Lambda Expressions and Collections

The previous section introduced the `Function` interface and finished with an example of basic lambda expression syntax. This section reviews how lambda expressions improve the `Collection` classes. In the examples created so far, the collections classes were used quite a bit. However, a number of new lambda expression features change the way collections are used. This section introduces a few of these new features.

6.1 Class additions

The drivers, pilots, and draftees search criteria have been encapsulated in the following `SearchCriteria` class (see the file).

SearchCriteria.java

```
import java.util.HashMap;
import java.util.Map;
import java.util.function.Predicate;

public class SearchCriteria {
    private static final int LOW = 16;
    private static final int START = 18;
    private static final int MID = 23;
    private static final int END = 25;
    private static final int RETIRED = 65;

    private final Map<String, Predicate<Person>> searchMap = new HashMap<>();

    private SearchCriteria() {
        super();
        initSearchMap();
    }

    public static SearchCriteria getInstance() {
        return new SearchCriteria();
    }

    private void initSearchMap() {
        Predicate<Person> allDrivers = p -> p.getAge() >= LOW;
        Predicate<Person> allDraftees = p -> p.getAge() >= START
                && p.getAge() <= END && p.getGender() == Gender.MALE;
        Predicate<Person> allPilots = p -> p.getAge() >= MID
                && p.getAge() <= RETIRED;

        searchMap.put("allDrivers", allDrivers);
        searchMap.put("allDraftees", allDraftees);
        searchMap.put("allPilots", allPilots);
    }

    public Predicate<Person> getCriteria(String PredicateName) {
        Predicate<Person> target;

        target = searchMap.get(PredicateName);

        if (target == null) {
            System.out.println("Search Criteria not found... ");
            System.exit(1);
        }
        return target;
    }
}
```

The `Predicate` based search criteria are stored in this class and available for our test methods.

6.2 Looping

The first feature to examine is the new `forEach` method available to any collection class. Here are a couple of examples that print out a `Person` list (see the file).

Test01ForEach.java

```
import java.util.List;

public class Test01ForEach {

    public static void main(String[] args) {

        List<Person> pl = Person.createShortList();

        System.out.println("\n=== Western Phone List ===");
        pl.forEach(p -> p.printWesternName());

        System.out.println("\n=== Eastern Phone List ===");
        pl.forEach(Person::printEasternName);

        System.out.println("\n=== Custom Phone List ===");
        pl.forEach(p -> {
            System.out.println(p.printCustom(r -> "Name: "
                    + r.getGivenName() + " EMail: " + r.getEmail()));
        });

    }

}
```

The first example shows a standard lambda expression which calls the `printWesternName` method to print out each person in the list. The second example demonstrates a *method reference*. In a case where a method already exists to perform an operation on the class, this syntax can be used instead of the normal lambda expression syntax. Finally, the last example shows how the `printCustom` method can also be used in this context. Notice the slight change in variable names when one lambda expression is included in another.

You can iterate through any collection this way. The basic structure is similar to the enhanced for loop. However, including an iteration mechanism within the class provides a number of benefits.

6.3 Chaining and Filtering

In addition to looping through the contents of a collection, you can chain methods together. The first method to look at is `filter` which takes a `Predicate` interface as a parameter.

The following example loops though a `List` after first filtering the results (see the file).

Test02Filter.java

```
import java.util.List;

public class Test02Filter {

    public static void main(String[] args) {

        List<Person> pl = Person.createShortList();

        SearchCriteria search = SearchCriteria.getInstance();

        System.out.println("\n=== Western Pilot Phone List ===");

        pl.stream().filter(search.getCriteria("allPilots"))
                .forEach(Person::printWesternName);


        System.out.println("\n=== Eastern Draftee Phone List ===");

        pl.stream().filter(search.getCriteria("allDraftees"))
                .forEach(Person::printEasternName);

    }
}
```

The first and last loops demonstrate how the `List` is filtered based on the search criteria. The output from the last loop looks something like the following:

```
=== Eastern Draftee Phone List ===

Name: Baker Bob
Age: 21  Gender: MALE
EMail: bob.baker@example.com
Phone: 201-121-4678
Address: 44 4th St, Smallville, KS 12333

Name: Doe John
Age: 25  Gender: MALE
EMail: john.doe@example.com
Phone: 202-123-4678
Address: 33 3rd St, Smallville, KS 12333
```

6.3.1 Getting Lazy

These features are useful, but why add them to the collections classes when there is already a perfectly good for loop? By moving iteration features into a library, it allows the developers of Java to do more code optimisations. To explain further, a couple of terms need definitions:

Laziness: In programming, laziness refers to processing only the objects that you want to process when you need to process them. In the previous example, the last loop is "lazy" because it loops only through the two `Person` objects left after the `List` is filtered. The code should be more efficient because the final processing step occurs only on the selected objects.

Eagerness: Code that performs operations on every object in a list is considered "eager". For example, an enhanced for loop that iterates through an entire list to process two objects, is considered a more "eager" approach.

By making looping part of the collections library, code can be better optimised for "lazy" operations when the opportunity arises. When a more eager approach makes sense (for example, computing a sum or an average), eager operations are still applied. This approach is a much more efficient and flexible than always using eager operations.

6.3.2 The `stream` Method

In the previous code example, notice that the `stream` method is called before filtering and looping begin. This method takes a `Collection` as input and returns a `java.util.stream.Stream` interface as the output. A `Stream` represents a sequence of elements on which various methods can be chained. By default, once elements are consumed they are no longer available from the stream. Therefore, a chain of operations can occur only once on a particular `Stream`. In addition, a `Stream` can be serial (default) or parallel depending on the method called. An example of a parallel stream is included at the end of this section.

6.4 Mutation and Results

As previously mentioned, a `Stream` is disposed of after its use. Therefore, the elements in a collection cannot be changed or mutated with a `Stream`. However, what if you want to keep elements returned from your chained operations? You can save them to a new collection. The following code shows how to do just that (see the file):

Test03toList.java

```
1  import java.util.List;
2  import java.util.stream.Collectors;
3
4  public class Test03toList {
5
6      public static void main(String[] args) {
7
8          List<Person> pl = Person.createShortList();
9
10         SearchCriteria search = SearchCriteria.getInstance();
11
12         // Make a new list after filtering.
13         List<Person> pilotList = pl
14                 .stream()
15                 .filter(search.getCriteria("allPilots"))
```

```
                .collect(Collectors.toList());

        System.out.println("\n=== Western Pilot Phone List ===");
        pilotList.forEach(Person::printWesternName);

    }

}
```

The `collect` method is called with one parameter, the `Collectors` class. The `Collectors` class is able to return a `List` or `Set` based on the results of the stream. The example shows how the result of the stream is assigned to a new `List` which is iterated over.

6.5 Calculating with `map`

The `map` method is commonly used with `filter`. The method takes a property from a class and does something with it. The following example demonstrates this by performing calculations based on the `age` field of `Person` (see the file):

Test04Map.java

```
import java.util.List;
import java.util.OptionalDouble;

public class Test04Map {

    public static void main(String[] args) {
        List<Person> pl = Person.createShortList();

        SearchCriteria search = SearchCriteria.getInstance();

        // Calc average age of pilots old style
        System.out.println("== Calc Old Style ==");
        int sum = 0;
        int count = 0;

        for (Person p : pl) {
            if (p.getAge() >= 23 && p.getAge() <= 65) {
                sum = sum + p.getAge();
                count++;
            }
        }

        long average = sum / count;
        System.out.println("Total Ages: " + sum);
        System.out.println("Average Age: " + average);


        // Get sum of ages
        System.out.println("\n== Calc New Style ==");
```

```
        long totalAge = pl
                .stream()
                .filter(search.getCriteria("allPilots"))
                .mapToInt(p -> p.getAge())
                .sum();

        // Get average of ages
        OptionalDouble averageAge = pl
                .parallelStream()
                .filter(search.getCriteria("allPilots"))
                .mapToDouble(p -> p.getAge())
                .average();

        System.out.println("Total Ages: " + totalAge);
        System.out.println("Average Age: " + averageAge.getAsDouble());

    }

}
```

And the output from the program is:

```
== Calc Old Style ==
Total Ages: 150
Average Age: 37

== Calc New Style ==
Total Ages: 150
Average Age: 37.5
```

The program calculates the average age of pilots in our list. The first loop demonstrates the old style of calculating the number by using a `for` loop. The second loop uses the `map` method to get the age of each person in a serial stream. Notice that `totalAge` is a `long`. The `map` method returns an `IntSteam` object, which contains a `sum` method that returns a `long`.

Note: To compute the average the second time, calculating the sum of ages is unnecessary. However, it is instructive to show an example with the `sum` method.

The last loop computes the average age from the stream. Notice that the `parallelStream` method is used to get a parallel stream so that the values can be computed concurrently. The return type has also changed.

7 A longer example and the `Optional` class

The following program creates an `ArrayList` called `myList` that holds a collection of integers (which are automatically boxed into the `Integer` reference type). Next, it obtains a stream that uses `myList` as a source. It then demonstrates various stream operations (see the file).

StreamDemo.java

```
package one;

import java.util.*;
import java.util.stream.*;

public class StreamDemo {

  public static void main(String[] args) {

    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>();
    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    System.out.println("Original list: " + myList);

    // Obtain a Stream to the array list.
    Stream<Integer> myStream = myList.stream();

    // Obtain the minimum and maximum value by uses of min(),
    // max(), isPresent(), and get().
    Optional<Integer> minVal = myStream.min(Integer::compare);
    if (minVal.isPresent())
      System.out.println("Minimum value: " + minVal.get());

    // Must obtain a new stream because previous call to min()
    // is a terminal operation that consumed the stream.
    myStream = myList.stream();
    Optional<Integer> maxVal = myStream.max(Integer::compare);
    if (maxVal.isPresent())
      System.out.println("Maximum value: " + maxVal.get());

    // Sort the stream by use of sorted().
    Stream<Integer> sortedStream = myList.stream().sorted();

    // Display the sorted stream by use of forEach().
    System.out.print("Sorted stream: ");
    sortedStream.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values by use of filter().
    Stream<Integer> oddVals = myList.stream().sorted()
        .filter((n) -> (n % 2) == 1);
    System.out.print("Odd values: ");
    oddVals.forEach((n) -> System.out.print(n + " "));
    System.out.println();
```

```
51
52      // Display only the odd values that are greater than 5. Notice that
53      // two filter operations are pipelined.
54      oddVals = myList.stream().filter((n) -> (n % 2) == 1)
55                      .filter((n) -> n > 5);
56      System.out.print("Odd values greater than 5: ");
57      oddVals.forEach((n) -> System.out.print(n + " "));
58      System.out.println();
59    }
60  }
```

The output is shown here:

```
1  Original list: [7, 18, 10, 24, 17, 5]
2  Minimum value: 5
3  Maximum value: 24
4  Sorted stream: 5 7 10 17 18 24
5  Odd values: 5 7 17
6  Odd values greater than 5: 7 17
```

We will now examine each stream operation. After creating an `ArrayList`, the program obtains a stream for the list by calling `stream()`:

```
1  Stream<Integer> myStream = myList.stream();
```

As `Collection` is implemented by every collection class, `stream()` can be used to obtain a stream for any type of collection, including the `ArrayList` used here. In this case, a reference to the stream is assigned to `myStream`.

Next, the program obtains the minimum value in the stream (which is, of course, also the minimum value in the data source) and displays it, as shown here:

```
1  Optional<Integer> minVal = myStream.min(Integer::compare);
2  if (minVal.isPresent())
3    System.out.println("Minimum value: " + minVal.get());
```

Remember that `min()` is declared as:

```
1  Optional<T> min(Comparator<? super T> comp)
```

First, notice that the type of `min()`'s parameter is a `Comparator`. This comparator is used to compare two elements in the stream. In the example, `min()` is passed a method reference to `Integer`'s `compare()` method, which is used to implement a `Comparator` capable of comparing two `Integer`s.

Next, notice that the return type of `min()` is `Optional`. `Optional` is a generic class packaged in `java.util` and declared as follows:

```
class Optional<T>
```

Here, `T` specifies the element type. An `Optional` instance can either contain a value of type `T` or be empty. You can use `isPresent()` to determine if a value is present. Assuming that a value is available, it can be obtained by calling `get()`. In this example, the object returned will hold the minimum value of the stream as an `Integer` object.

One other point about the preceding line: `min()` is a terminal operation that consumes the stream. Thus, `myStream` cannot be used again after `min()` executes.

The next lines obtain and display the maximum value in the stream:

```
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if (maxVal.isPresent())
  System.out.println("Maximum value: " + maxVal.get());
```

First, `myStream` is once again assigned the stream returned by `myList.stream()`. As just explained, this is necessary because the previous call to `min()` consumed the previous stream and therefore a new one is needed. Next, the `max()` method is called to obtain the maximum value. Like `min()`, `max()` returns an `Optional` object. Its value is obtained by calling `get()`.

The program then obtains a sorted stream through the use of this line:

```
Stream<Integer> sortedStream = myList.stream().sorted();
```

Here, the `sorted()` method is called on the stream returned by `myList.stream()`. As `sorted()` is an intermediate operation, its result is a new stream, and this is the stream assigned to `sortedStream`. The contents of the sorted stream are displayed by use of `forEach()`:

```
sortedStream.forEach((n) -> System.out.println(n + " "));
```

Here, the `forEach()` method executes an operation on each element in the stream. In this case, it simply calls `System.out.print()` for each element in `sortedStream`. This is accomplished by use of a lambda expression. The `forEach()` method has this general form:

```
void forEach(Consumer<? super T> action)
```

`Consumer` is a generic functional interface declared in `java.util.function`. Its abstract method is `accept()`, shown here:

```
void accept(T objRef)
```

The lambda expression in the call to `forEach()` provides the implementation of `accept()`. The `forEach()` method is a terminal operation. After it completes, the stream has been consumed.

Next, a sorted stream is filtered by `filter()` so that it contains only odd values:

```
Stream<Integer> oddVals = myList.stream().sorted()
                                         .filter((n) -> (n % 2) == 1);
```

The `filter()` method filters a stream based on a predicate. It returns a new stream that contains only those elements that satisfy the predicate. It is shown here:

```
Stream<T> filter(Predicate<? super T> pred)
```

`Predicate` is a generic functional interface defined in `java.util.function`. Its abstract method is `test()`, which is shown here:

```
boolean test(T objRef)
```

It returns `true` if the object referred to by `objRef` satisfies the predicate, and `false` otherwise. The lambda expression passed to `filter()` implements this method. As `filter()` is an intermediate operation, it returns a new stream that contains filtered values, which, in this case, are the odd numbers. These elements are then displayed via `forEach()` as before.

As `filter()`, or any other intermediate operation, returns a new stream, it is possible to filter a filtered stream a second time. This is demonstrated by the following line, which produces a stream that contains only those odd values greater than 5:

```
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                         .filter((n) -> n > 5);
```

Notice that lambda expressions are passed to both filters.

8 Summary

In these notes we have (briefly) considered the following aspects of Java SE 8:

- Anonymous inner classes in Java.
- Lambda expressions to replace anonymous inner classes in Java SE 8.
- The correct syntax for lambda expressions.
- A `Predicate` interface to perform searches on a list.
- A `Function` interface to process an object and produce a different type of object.
- New features added to `Collections` in Java SE 8 that support lambda expressions.

Credits

These notes are based upon a set of tutorials produced at Oracle by Michael Williams and Juan Quesada Nunez.