Java Object-Oriented Concepts Unit

Lesson 10: Lambdas and Streams

The Learning House

427 S 4th Street #300

Louisville KY 40202

Lesson 10: Lambdas and Streams

Overview

Up to this point, we have only been able to pass data as parameters into methods. Sometimes, it is useful to pass code, or functionality, into a method. Lambdas are a new feature implemented in Java 8 that allow us to do this — in this lesson, we'll explore how they can be used.

What are Lambdas?

Lambdas are a new Java language feature that allows us to pass functionality or behavior into methods as parameters. We have only seen examples of passing data into methods, so you may be asking yourself when this might be useful. One example that illustrates the usefulness of Lambdas comes from UI coding. When a user clicks on button on a user interface, it usually causes some action to occur in the application. In this case, we really want to pass a behavior into the onClick(...) method so that the application will execute the given behavior when the button is clicked. In previous versions of Java, we accomplished this by passing an anonymous inner class (that implemented a known interface) into the method. Interfaces used in this scenario usually contain only one method which defined the behavior we wished to pass into the onClick(...) method. Although this works, the syntax is unwieldy. Anonymous inner classes still work for this purpose, but the new Lambda syntax is much cleaner.

Java 8 Aggregate Operations

When we use Collections to store objects in our programs, we generally need to do more than simply put the objects in the collection — we need to store, retrieve, remove, and update these objects. Aggregate operations use Lambdas to perform actions on the objects in a Collection. For example, you can use aggregate operations to:

- Print the names of all the people in a Collection of Address objects
- Return all of the Address objects for people from Akron, OH
- Return all of the Address objects for people from Akron, OH grouped by zip code
- Calculate and return the average age of Servers in your inventory (provided the Server object has a purchase date field)

These tasks can be accomplished by using aggregate operations along with pipelines and streams.

Pipelines and Streams

A pipeline is simply a sequence of aggregate operations. A stream is a sequence of items, **not a data structure**, that carries items from the source through the pipeline.

Pipelines are comprised of the following:

1. A data source. Most commonly, this is a Collection, but it could be an array, the return from a method call, or some sort of I/O channel.

2. Zero or more intermediate operations. For example, a Filter operation. Intermediate operations produce a new stream. A filter operation takes in a stream and then produces another stream that contains only the items matching the criteria of the filter.
3. A terminal operation. Terminal operations return a non-stream result. This result could be a primitive type (for example, an integer), a Collection, or no result at all (for example, the operation might just print the name of each item in the stream).

Relationship Between Aggregate Operations and Iterators

Some aggregate operations (i.e. forEach) look like iterators, but they have fundamental differences:

1. Aggregate operations use internal iteration. Your application has no control over how or when the elements are processed (there is no next() method).
2. Aggregate operations process items from a stream, not directly from a Collection.
3. Aggregate operations support Lambda expressions as parameters.

Lambda Syntax

Now that we have discussed the concepts related to Lambda expressions, it is time to look at their syntax. You can think of Lambda expressions as anonymous methods because they have no name. Lambda syntax consists of the following:

- A comma-separated list of formal parameters enclosed in parentheses. Data types of parameters can be omitted in Lambda expressions. The parentheses can be omitted if there is only one formal parameter.
- The arrow token: `->`
- A body consisting of a single expression or code block

We'll look at examples in the next section.

Using Lambdas and Aggregate Operations

The following examples show how we can use Lambdas and Aggregate Operations to filter and process Server objects from two different data structures — an ArrayList and a HashMap.

Server

This is the Server object that we'll process in our examples:

```
package serverinventory;

import java.time.LocalDate;
import java.time.Period;

/**
 *
 * @author apprentice
 */
public class Server {
    private String name;
    private String ip;
    private String make;
    private String ram;
    private String numProcessors;
    private LocalDate purchaseDate;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getIp() {
        return ip;
    }
    public void setIp(String ip) {
        this.ip = ip;
    }
    public String getMake() {
        return make;
    }
    public void setMake(String make) {
        this.make = make;
    }
```

```
    public String getRam() {
        return ram;
    }
    public void setRam(String ram) {
        this.ram = ram;
    }
    public String getNumProcessors() {
        return numProcessors;
    }
    public void setNumProcessors(String numProcessors) {
        this.numProcessors = numProcessors;
    }
    public LocalDate getPurchaseDate() {
        return purchaseDate;
    }
    public void setPurchaseDate(LocalDate purchaseDate) {
        this.purchaseDate = purchaseDate;
    }
    public long getServerAge() {
        Period p = purchaseDate.until(LocalDate.now());
        return p.getYears();
    }
}
```

ArrayList Example

The following code shows examples of using Lambdas, Aggregate Operations and an ArrayList to filter and process our server objects:

```
public static void main (String[] args) {
        // We'll hold our server objects in an ArrayList for this example
        List<Server> servers = new ArrayList<>();

        // Create several server objects and add them to our list
        Server one = new Server();
        one.setName("web01");
        one.setIp("192.168.1.1");
        one.setMake("Dell");
        one.setRam("8GB");
        one.setNumProcessors("9");
        one.setPurchaseDate(LocalDate.parse("2010-01-01", DateTimeFormatter.ISO_DATE));

        servers.add(one);

        one = new Server();
        one.setName("db01");
        one.setIp("192.168.3.45");
        one.setMake("HP");
        one.setRam("16GB");
        one.setNumProcessors("24");
        one.setPurchaseDate(LocalDate.parse("2013-01-01", DateTimeFormatter.ISO_DATE));

        servers.add(one);

        one = new Server();
        one.setName("hr124");
        one.setIp("192.168.32.111");
        one.setMake("IBM");
        one.setRam("16GB");
        one.setNumProcessors("12");
        one.setPurchaseDate(LocalDate.parse("2014-01-01", DateTimeFormatter.ISO_DATE));

        servers.add(one);


        one = new Server();
        one.setName("eng16");
        one.setIp("192.168.32.56");
        one.setMake("HP");
        one.setRam("4GB");
        one.setNumProcessors("8");
        one.setPurchaseDate(LocalDate.parse("2001-01-01", DateTimeFormatter.ISO_DATE));
```

```
        servers.add(one);

        one = new Server();
        one.setName("eng64");
        one.setIp("192.168.34.56");
        one.setMake("HP");
        one.setRam("8GB");
        one.setNumProcessors("16");
        one.setPurchaseDate(LocalDate.parse("2001-01-01", DateTimeFormatter.ISO_DATE));

        servers.add(one);

        // example - find all servers for given make
        // note we are ignoring case when we compare make
        String make = "dElL";
        System.out.println("All " + make + " servers in inventory: ");
        servers
                .stream()
                .filter(s -> s.getMake().equalsIgnoreCase(make))
                .forEach(e -> System.out.println(e.getName()));

        // example - print all server names older than a given age
        long testAge = 3;
        System.out.println("All servers older than " + testAge + " years in inventory:");
        servers
                .stream()
                .filter(s -> s.getServerAge() > testAge)
                .forEach(e -> System.out.println(e.getName()));

        // get a list of all servers older than a given age
        List<Server> oldServers
                = servers
                .stream()
                .filter(s -> s.getServerAge() > testAge)
                .collect(Collectors.toList());

        // print the size of our list
        System.out.println(oldServers.size());

        // print out the name of each server in the list
        oldServers.forEach(s -> System.out.println(s.getName()));



        // example - calculate the average age of servers in inventory
        double averageAge = servers
                .stream()
                .mapToLong(Server::getServerAge)
                .average()
                .getAsDouble();
        System.out.println("Average age of servers is " + averageAge + " years.");
    }
```

HashMap Example

The following code shows examples of using Lambdas, Aggregate Operations and an ArrayList to filter and process our server objects:

```
public class ServerInventoryMapExample {
    public static void main(String[] args) {
        // Use a Map to hold our Server objects
        // Use server name as the key since servers must be unique on the
        // network
        Map<String, Server> serverMap = new HashMap<>();
        // Create servers and add to our Map
        Server one = new Server();
        one.setName("web01");
        one.setIp("192.168.1.1");
        one.setMake("Dell");
        one.setRam("8GB");
        one.setNumProcessors("9");
        one.setPurchaseDate(LocalDate.parse("2010-01-01", DateTimeFormatter.ISO_DATE));
        serverMap.put(one.getName(), one);

        one = new Server();
        one.setName("db01");
        one.setIp("192.168.3.45");
        one.setMake("HP");
        one.setRam("16GB");
        one.setNumProcessors("24");
        one.setPurchaseDate(LocalDate.parse("2013-01-01", DateTimeFormatter.ISO_DATE));
        serverMap.put(one.getName(), one);

        one = new Server();
        one.setName("hr124");
        one.setIp("192.168.32.111");
        one.setMake("IBM");
        one.setRam("16GB");
        one.setNumProcessors("12");
        one.setPurchaseDate(LocalDate.parse("2014-01-01", DateTimeFormatter.ISO_DATE));
        serverMap.put(one.getName(), one);


        one = new Server();
        one.setName("eng16");
        one.setIp("192.168.32.56");
        one.setMake("HP");
        one.setRam("4GB");
        one.setNumProcessors("8");
        one.setPurchaseDate(LocalDate.parse("2001-01-01", DateTimeFormatter.ISO_DATE));
        serverMap.put(one.getName(), one);
```

```
        one = new Server();
        one.setName("eng64");
        one.setIp("192.168.34.56");
        one.setMake("HP");
        one.setRam("8GB");
        one.setNumProcessors("16");
        one.setPurchaseDate(LocalDate.parse("2001-01-01", DateTimeFormatter.ISO_DATE));
        serverMap.put(one.getName(), one);

        // Find all servers of a particular make (case insensitive)
        // Note that we use the Collection returned from values() as the
        // source of our stream
        String make = "dElL";
        System.out.println("All " + make + " servers in inventory: ");
        serverMap.values()
                .stream()
                .filter(s -> s.getMake().equalsIgnoreCase(make))
                .forEach(e -> System.out.println(e.getName()));

        // example - find all servers older than a given age
        int testAge = 3;
        System.out.println("All servers older than " + testAge + " years in inventory:");
        serverMap.values()
                .stream()
                .filter(s -> s.getServerAge() > testAge)
                .forEach(e -> System.out.println(e.getName()));

        // get a list of all servers older than a given age
        Collection<Server> oldServers
                = serverMap.values()
                .stream()
                .filter(s -> s.getServerAge() > testAge)
                .collect(Collectors.toList());

        System.out.println(oldServers.size());
        oldServers.forEach(s -> System.out.println(s.getName()));

        // example - calculate the average age of servers in inventory
        double averageAge = serverMap.values()
                .stream()
                .mapToLong(Server::getServerAge)
                .average()
                .getAsDouble();
        System.out.println("Average age of servers is " + averageAge + " years.");
    }
}
```