Java 8 Features

Student Workbook

Version 1.1

Table of Contents

Module 1

Java 8

Section 1–1

Java 8

Java 8

- **When Java 8 was released in 2014, it contained many new, powerful features, including:**
  - an improved Date/Time API
  - support for Lambda expressions .
  - a new Optional type to handle null values
  - introducing default methods for interfaces
  - a new stream API to enable pipeline processing
  - the ability to use functions as parameters
- **In the next few pages, we will look at a few of these**

Section 1–2

Date/Time API Changes

Interesting Date/Time Changes

- **Java 8's a new Date/Time API solves some of the problems with earlier Java dates and time**
  - The new Date/Time API is found in the package `java.time`
- **Some of the problems addressed include:**
  - Providing more support for date operations
  - Improved time zone handling
  - Making the new Date/Time API thread safe
- **The API is broken down into two subsets:**
  - Local - contains a Date/Time API that doesn't have the complexity of time zone management
  - Zoned – a Date/Time API specifically designed to deal with time zones

Local Date/Time API

- **It contains classes such as: `LocalDate`, `LocalTime`, and `LocalDateTime`**

- **You can grab the current date/time and then extract the data and time portions**

Example

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
...

LocalDateTime rightNow = LocalDateTime.now();
LocalDate today = rightNow.toLocalDate();
LocalTime thisMoment = rightNow.toLocalTime();

System.out.println("Right now it is: " + rightNow);
System.out.println("Today is: " + today);
System.out.println("At this moment, it is: " + thisMoment);
```

OUTPUT

```
Right now it is: 2021-09-11T09:12:30.236
Today is: 2021-09-11
At this moment, it is: 09:12:30
```

- **You can use static methods to create dates and times**

Example

```
import java.time.Month;
...

LocalDate holiday = LocalDate.of(2021, Month.SEPTEMBER, 6);
System.out.println("Labor day is: " + holiday);
```

OUTPUT

```
Labor day is: 2021-09-05
```

Local Date/Time API *cont'd*

Example

```
LocalTime startTime = LocalTime.of(8, 0);
System.out.println("Class starts at: " + startTime);
```

OUTPUT

```
Class starts at: 08:00
```

- **You can do date and/or time arithmetic**

Example

```
LocalDateTime rightNow = LocalDateTime.now();
LocalDateTime returnBy = rightNow.plusDays(7);

System.out.println("Right now it is: " + rightNow);
System.out.println("You should return by: " + returnBy);
```

OUTPUT

```
Right now it is: 2021-09-11T09:12:30.236
Right now it is: 2021-09-18T09:12:30.236
```

- **You can reach in an extract pieces if a date and/or time you are interested in**

Example

```
LocalDateTime rightNow = LocalDateTime.now();
LocalDateTime returnBy = rightNow.plusMonths(6);

if (rightNow.getYear() != returnBy.getYear()) {
   System.out.println("See you next year!");
}
```

Zoned Date/Time API

- **The zoned Date/Time API takes into account that times time zones can impact how people view numbers**
  - It contains classes such as: `ZonedDateTime` and `ZoneId`
  - `https://garygregory.wordpress.com/2013/06/18/what-are-the-java-timezone-ids/`

- **You can grab the current date/time by specifying a time zone**

Example

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

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

    ZonedDateTime defaultNow = ZonedDateTime.now();

    ZonedDateTime zonedNow =
       ZonedDateTime.now(ZoneId.of("America/Chicago"));

    ZonedDateTime indiaNow =
       ZonedDateTime.now(ZoneId.of("Indian/Cocos"));

    System.out.println("Local: " + defaultNow);
    System.out.println("Zoned: " + zonedNow);
    System.out.println("India: " + indiaNow);
  }
}
```

OUTPUT

```
Local: 2021-10-24T17:45:45.973668Z[GMT]
Zoned: 2021-10-24T12:45:46.001340-05:00[America/Chicago]
India: 2021-10-25T00:15:46.003296+06:30[Indian/Cocos]
```

Working with Periods and Durations

- **Java 8 introduced classes to deal with the date and time differences**
  - The `Period` holds date-based amounts of time
  - The `Duration` holds time-based amounts of time

Example

```
import java.time.LocalDateTime;
import java.time.Period;

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

      LocalDateTime rightNow = LocalDateTime.now();
      LocalDateTime returnBy = rightNow.plusDays(90);

      Period period = Period.between(returnBy.toLocalDate(),
                                     rightNow.toLocalDate());

      System.out.println("Period: " + period);
    }
}
```

OUTPUT

```
Period: P-2M-29D
```

  - You can use methods to extract units from a period such as:
    - `getDays()`
    - `getMonths()`
    - `getYears()`

Working with Periods and Durations *cont'd*

Example

```
import java.time.LocalDateTime;
import java.time.Duration;

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

      LocalDateTime rightNow = LocalDateTime.now();
      LocalDateTime returnBy =
         rightNow.plusHours(3).plusMinutes(15);

      Duration duration =
         Duration.between(returnBy.toLocalTime(),
                          rightNow.toLocalTime());
      System.out.println("Duration: " + duration);
    }
}
```

OUTPUT

```
Duration: PT-3H-15M
```

- You can use methods to extract units from a duration such as:
  - `getHours()`
  - `getMinutes()`

Working with Temporal Adjusters

- **The `TemporalAdjuster` class is used to perform the date arithmetic**
  - For example, it can be used to find out what date is "Next Friday"

Example

```
import java.time.LocalDate;
import java.time.DayOfWeek;
import java.time.temporal.TemporalAdjusters;

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

      LocalDate today = LocalDate.now();

      // Determine what date is the upcoming Friday
      LocalDate nextFriday = today.with(
         TemporalAdjusters.next(DayOfWeek.FRIDAY));

      System.out.println("Next Friday is : " + nextFriday);
   }
}
```

OUTPUT

```
Next Friday is : 2021-10-01
```

  - Or it can be used to find the "First Sunday of the Month"

Example

```
import java.time.*;
import java.time.temporal.*;

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

Working with Temporal Adjusters *cont'd*

```
        // get today
        LocalDate today = LocalDate.now();

        // get the first of next month
        LocalDate oneMonthAway = today.plusMonths(1);
        LocalDate nextMonth =
             LocalDate.of(oneMonthAway.getYear(),
                          oneMonthAway.getMonth(), 1);

        // determine the first Sunday of the month
        LocalDate firstSunday = nextMonth.with(
            TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY));

        System.out.println("BENBROOK FARMERS MARKET");
        System.out.println(
           "Next month it is on : " + firstSunday);
    }
}
```

OUTPUT

```
BENBROOK FARMERS MARKET
Next month it is on : 2021-10-03
```

Backward Compatibility

- **To encourage Java programmers to upgrade to the new Date/Time API, Java 8 added the `toInstant()` method to Java's `Date` and `Calendar` classes**
  - `toInstant()` returns an `Instant` object containing a date/time as a number of milliseconds since January 1, 1979
  - You can then use the `LocalDateTime` and `ZonedDateTime` methods to use an `Instant` object to build a new Date/Time

Example

```
import java.util.Date;
import java.time.*;

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

      // Use older Java class to get the current date
      Date currentDate = new Date();

      // Get the instant of current date
      Instant now = currentDate.toInstant();

      // Convert the instant to a ZonedDateTime in DFW
      ZonedDateTime nowInDFW =
         now.atZone(ZoneId.of("America/Chicago"));

      System.out.println("Date: " + currentDate);
      System.out.println("in DFW: " + nowInDFW);
   }
}
```

OUTPUT

```
Date: Sun Oct 24 18:50:00 GMT 2021
in DFW: 2021-10-24T13:50:00.040-05:00[America/Chicago]
```

Section 1–3

Lambda Expressions

Lambda Expressions

- **One of the most anticipated features added in Java 8 was lambda expressions**
  - They are similar to what you saw with arrow functions in JavaScript and can simplify coding quite a bit

- **A lambda expression is a function in disguise**
  - It is passed as a parameter to another function

Syntax

```
parameter -> expression body
```

or

```
(parameter1, parameter2, …) -> expression body
```

- **Rules for lambda expressions include:**
  - Optional parameter type declaration – the compiler infers the type from the value of the parameter
  - Parenthesis are only needed around parameters if there are more than one
  - Curly braces are only needed around the expression body if the body contains more than one statement
  - No return statement needed if the expression body is just a single expression

Using Lambdas with Collections

- **Java 8 added a `stream()` method to the `Collection` interface that returns the collection in a way that will work with lambdas**

Example

```
ArrayList<Employee> employees = new ArrayList<>();
...

Stream<Employee> = employees.stream();
```

- **The Stream provides a pipeline that you can use to provide processing on the elements in the stream**
  - `https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html`
  - In the following example, we will examine `filter()`

Example

```
import java.util.*;
import java.util.stream.Collectors;

// Use the List interface as the type of employees
// so that we can work with some of the generic methods
List<Employee> employees = new ArrayList<>();
...

List<Employee> matchingEmps = employees.stream()
      .filter(p -> p.getLastName().equals(lastName))
      .collect(Collectors.toList());

System.out.println("Matching: " + matchingEmps);
```

Lambdas and `forEach()`

- **Java 8 introduced `forEach()` for collections**
  - The lambda expression is executed for ***each*** element in the collection

Example

```
import java.util.ArrayList;

public class Program {

  public static void main(String[] args) {

    List<String> names = new ArrayList<String>();
    names.add("Ezra");
    names.add("Ian");
    names.add("Siddalee");
    names.add("Elisha");
    names.add("Pursalane");
    names.add("Zephaniah");

    names.forEach(name -> {
       System.out.println(name);
    });
  }
}
```

- **The lambda expression that is passed to `forEach` could be multiple lines long**

Example

Employee.java

```
public class Employee {
   private String name;
   private String jobTitle;
   private double salary;
```

Lambdas and `forEach()` *cont'd*

```
    public Employee(String name, String jobTitle,
    double salary) {
       this.name = name;
       this.jobTitle = jobTitle;
       this.salary = salary;
    }

    public double getSalary() {
       return salary;
    }

    public void setSalary(double salary) {
       this.salary = salary;
    }
}
```

Program.java

```
import java.util.ArrayList;

public class Program {

  public static void main(String[] args) {

    List<Employee> emps = new ArrayList<Employee>();
    emps.add(new Employee("Ezra", "Actor", 72750.0));
    emps.add(new Employee("Ian", "Banker", 252750.0));
    emps.add(new Employee("Siddalee", "Model", 1500000.0));
    emps.add(new Employee("Elisha", "Programmer", 103500.0));
    emps.add(new Employee("Pursalane", "Teacher", 697250.0));
    emps.add(new Employee("Zephaniah", "Engineer", 136000.0));

    emps.forEach(emp -> {
       if (emp.getSalary() < 100000) {
          emp.setSalary(101000.0);
       }
       else {
          emp.setSalary(emp.getSalary() * 1.1);
       }
       System.out.println(emp.getName() + " earns $" +
                     String.format("%.2f", emp.getSalary());
    });
  }
}
```

Section 1–4

(More About) Streams
(Self-Study)

Streams

- **Java 8 introduced Streams to make it easier to process data without writing as many loops and calling if statements**
  - Not only does it make it easier on the programmer, but it has been optimized to work on multi-core processors a\
- **A Java stream essentially represents a sequence of objects**
  - The source is typically collections, arrays or I/O resources
- **The stream supports aggregate operations like `filter()`, `findFirst()`, `findAny()`, `limit()`, and `sort()`**
- **Most `Stream` methods return the stream, which allows chaining method calls to provide a pipeline-like channel of operations**
  - You can use `collect()` to mark the end of the processing and return output

Generating Streams and Collecting Results

- **With Java 8, the `Collection` interface has two methods to generate a `Stream`**
  - `stream()` returns a sequential stream of the collection data
  - `parallelStream()` returns a parallel stream of the collection data
    - Parallel streams are beyond the scope of this class

Example

```
List<String> states = Arrays.asList(
      "Alabama", "Alaska", "Arizona", "Arkansas",
      "California", "Colorado", "Connecticut",
      /* others not shown */ );

String letter = "C";

List<String> matchingStates = states.stream()
             .filter(state -> state.startsWith(letter))
             .collect(Collectors.toList());
```

- **Collectors combine the result of processing on the elements of a stream**
  - `Collectors.toList()` returns a `List<T>` where `T` is the type of data collected
  - If you want to collect the results into an `ArrayList`, you can use a different collector
    - In this case, use `Collectors.toCollection(ArrayList::new)`

Generating Streams and Collecting Results *cont'd*

Example

```
public ArrayList<String> getMatchingStates(String firstChars) {

   // this.states is a class-level ArrayList<String>
   ArrayList<String> matchingStates = this.states.stream()
             .filter(state -> state.startsWith(firstChars))
             .collect(Collectors.toCollection(ArrayList::new));
   return matchingStates;
}
```

filter() and count()

- **The `filter()` method reduces the stream to only the elements that match a condition**

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

List<String> matching = titles.stream()
     .filter(title -> title.toLowerCase().contains("halloween))
     .collect(Collectors.toList());
```

- **The `count()` method returns a count of the number of items in the stream**
  - In this case, it only returns the count

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

int count = titles.stream()
    .filter(title -> title.toLowerCase().contains("Halloween"))
    .count();
```

forEach()

- **The `forEach()` method calls a named method for each value in the stream**

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

titles.stream()
    .filter(title -> title.toLowerCase().contains("halloween"))
    .forEach(System.out::println);
```

map()

- **The `map()` method maps each element in the stream to a new result**

Example

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

// get list of unique squares
List<Integer> squaresList = numbers.stream()
              .map(num -> num * num)
              .distinct()
              .collect(Collectors.toList());

System.out.println(squaresList);
```

OUTPUT

```
[9, 4, 49, 25]
```

sorted()

- **The `sorted()` method sorts the stream**

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

titles.stream()
   .filter(title -> title.toLowerCase().contains("halloween))
   .sorted()
   .forEach(System.out::println);
```

Example

```
List<String> titles = Arrays.asList(
       "Halloween", "Ghost", "Halloween 2",
       "Friday the 13th", "Twister", "Halloween 3");

List<String> sortedMovies = titles.stream()
   .filter(title -> title.toLowerCase().contains("halloween))
   .sorted()
   .collect(Collectors.toList());
```

Section 1–5

Default Methods

Default Methods

- **In Java 8, interfaces can have methods that provide a default implementation**

- **This was added for backward compatibility so that old Java interfaces could work with lambda expressions**
  - The `List` and `Collection` interfaces did not have a `forEach` method declaration
    - But it was needed to make lambda expressions work
  - They couldn't retroactively add it because classes that implemented List or Collection would break
  - So they provided default method implementations in Java 8 to let lambda expressions work with Lists and Collections

Syntax

```
public interface IMovable
{
    public void move(int xUnits, int yUnits);

    default void reportPosition()
    {
      System.out.println("I'm lost");
    }
}
```

Multiple Defaults

- **When interfaces can have default methods, it is possible a class might inherit from two interfaces with same default methods**

- **To solve this problem, explicitly implement the method**

Example

```
public interface IMovable
{
    public void move(int xUnits, int yUnits);

    default void reportPosition()
    {
      System.out.println("I'm lost");
    }
}

public interface IDecider
{
    public boolean makeDecision(String question);

    default void reportPosition()
    {
      System.out.println("I'm for this issue");
    }
}
```

- **Create a method that overrides the default implementation**
  - Within it, you can call the default method of your interfaces if you need to

Multiple Defaults *cont'd*

Example

```
public class MobileCommandUnit implements IMovable, IDecider {

   private int x = 0;
   private int y = 0;

   public void move(int xUnits, int yUnits) {
      x += xUnits;
      y += yUnits;
   }

   public boolean makeDecision(String question) {

      if (question.equalsIgnoreCase("attack?")) {
         if (x <= 20 && y <= 20) {
             return true;
         }
      }

      return false;
  }

  // provide your own implementation
  public void reportPosition()
  {
      System.out.printf("I'm at (%d, %d).", x, y);
  }
}
```

Section 1–6

Optionals

Optionals

- **In Java 8, an `Optional` object is a container that many, or may not, contain a non-null value**
  - It lets you check whether a value is `'available'` or `'not available'` instead of checking for a null value

- **To create an `Optional`, use `Optional.ofNullable()`**

Example

```
String[] words = new String[10];
// ...
Optional<String> optVal = Optional.ofNullable(words[8]);
```

- **To determine if an `Optional` has a value, use `isPresent()`**

Example

```
Optional<String> optVal = /* some value */;
//...
if(optVal.isPresent()) {

}
```

- **To extract the value from an `Optional`, use `get()`**

Example

```
Optional<String> optVal = /* some value */;
//...
```

Optionals *cont'd*

```
if(optVal.isPresent()) {
   String s = optVal.get();
   s = s.toLowerCase();
   //...

}
```

- **There are many other methods you can use with optionals**
  - See the following for more info: https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

Example: Testing for `null` Without `Optional`

Example

```
public class MainApp {

   public static void main(String args[]) {

       String[] words = new String[10];
       // load "some" of the array

       // this code might generate an exception
       // if words[0] is null
       String word = words[8].toLowerCase();
       System.out.print(word);
    }
}
```

Example

```
import java.util.Optional;

public class MainApp {

   public static void main(String args[]) {

       String[] words = new String[10];
       // load "some" of the array

       // this would be a common way to test to see
       // if the code would work
       if (words[8] != null) {
         String word = words[8].toLowerCase();
         System.out.print(word);
       }
    }
}
```

Example: Using Optional

Example

```
import java.util.Optional;

public class MainApp {

   public static void main(String args[]) {

       String[] words = new String[10];
       // load "some" of the array

       // put word[5] into an Optional
       Optional<String> word5 =
                 Optional.ofNullable(words[5]);

       // check if word5 has a value
       if (word5.isPresent()) {
          String word = word5.get().toLowerCase();
          System.out.print(word);
       }
    }
}
```

- **Note: At first glance, this might seem like extra work**
  - But it can be a useful way to send a value to another method and is descriptive that the parameter might be null