PerScholas - Quality Engineering

Java Programming - 03

Date & Time

- LocalDate, LocalTime & LocalDateTime classes
  - LocalDate represents a local date without time zone information
  - Read this page and the page right after it:
    - https://docs.oracle.com/javase/tutorial/datetime/iso/date.html
  - LocalTime represents a time without a time zone while LocalDateTime represents date and time without a time zone
  - Read this page on the LocalTime and LocalDateTime classes:
    - https://docs.oracle.com/javase/tutorial/datetime/iso/datetime.html
  - Use these pages as references for LocalTime and LocalDateTime:
    - https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html
    - https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
  - Here are two other references on Java date and time:
    - https://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html
    - http://tutorials.jenkov.com/java-date-time/index.html
- Advanced Challenge - read through the entire Oracle tutorial on Standard Calendar
  - https://docs.oracle.com/javase/tutorial/datetime/iso/index.html

LocalDate and LocalDateTime examples:

```java
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;

public class JavaBasics {
  public static void main(String[] args) {
    // Retrieve & print the current time using a specified format
    LocalDateTime currentDateTime = LocalDateTime.now();
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("MMMM dd, yyyy");
    String s = dtf.format(currentDateTime);
    System.out.println("Today's date is: " + s);
    // Create a LocalDate object and print it
    LocalDate platformLaunchDate = LocalDate.of(2018, 3, 5);
    System.out.println("Platform launch date: " + platformLaunchDate);
    // Compare 2 LocalDate objects
    LocalDate today = LocalDate.now();
    int c = today.compareTo(platformLaunchDate);
    if (c > 0) {
      System.out.println("Today is later then Platform Dallas launch date");
    }
    // Calculate difference between 2 LocalDateTime objects
    LocalDateTime platformLaunchTime = LocalDateTime.of(2018, 03, 5, 9, 0);
    long t = ChronoUnit.MINUTES.between(platformLaunchTime, currentDateTime);
    System.out.println("Minutes since Platform launch: " + t);
  }
}
```

Output:
Today's date is: November 07, 2018
Platform launch date: 2018-03-05
Today is later then Platform Dallas launch date
Minutes since Platform launch: 356146

- java.util.Date is an older class but is still used. However, the java.time package classes (e.g., LocalDate and LocalDateTime) include more features and are recommended for use with Java 8 (i.e., Java 1.8) or newer. Here is an article which discusses some of the differences.
  - The java.uitl.Date class represents a specific instance in time with millisecond precision. Equivalent to the newer Instant class.

- Use this page as a reference for the java.util.Date class: https://docs.oracle.com/javase/8/docs/api/java/util/Date.html
- The SimpleDateFormat class allows the Date class to be printed in a specified format
  - Use this page as a reference for the SimpleDateFormat class: https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html

java.util.Date examples

```
3   import java.text.SimpleDateFormat;
4   import java.util.Date;
5
6   public class JavaBasics {
7     public static void main(String[] args) {
8       // Get & print the current date & time
9       Date currentTime = new Date();
10      System.out.println("The current date & time is " + currentTime);
11      // Print the current date & time in a specified format
12      SimpleDateFormat sdf = new SimpleDateFormat("MMMM d, yyyy");
13      String currentTimeString = sdf.format(currentTime);
14      System.out.println(currentTimeString);
15    }
16  }
```

Output:
The current date & time is Wed Nov 07 11:49:23 CST 2018
November 7, 2018

- java.sql.Date is also an older class, but you may still encounter it in some software applications
  - Extends java.util.Date class - primary difference is that java.sql.Date does not include the time (if java.util.Date is converted to java.sql.Date, the time gets cut off)
  - Use this page as a reference for the java.sql.Date class: https://docs.oracle.com/javase/8/docs/api/java/sql/Date.html

Examples of converting between java.util.Date and java.sql.Date:

```
3   public class JavaBasics {
4     public static void main(String[] args) {
5       // java.util.Date to java.sql.Date
6       java.util.Date utilDate = new java.util.Date();
7       java.sql.Date sqlDate = new java.sql.Date(utilDate.getTime());
8       System.out.println("SQL Date from util: " + sqlDate);
9       // java.sql.Date to java.util.Date
10      java.sql.Date platformLaunchSql = new java.sql.Date((2018-1900),2,5);
11      java.util.Date platformLaunchUtil = new java.util.Date(platformLaunchSql.getTime());
12      System.out.println("Util Date from sql: " + platformLaunchUtil);
13    }
14  }
```

Output:
SQL Date from util: 2018-11-07
Util Date from sql: Mon Mar 05 00:00:00 CST 2018

- Calendar abstract class

  - An abstract class that provides methods for converting between an instant in time and specific date and/or time units (e.g., year, month, hour, min, etc.)
  - The Calendar abstract class can be implemented with the GregorianCalendar class
  - However, the Calendar abstract class can also be directly instantiated with the getInstance( ) method which creates an instance (i.e., object) of the Calendar class that is initialized with the current date and time

Examples of Calendar class features:

```java
import java.util.Calendar;
import java.util.Date;

public class JavaBasics {
  public static void main(String[] args) {
    // Create Calendar instance
    Calendar cal = Calendar.getInstance();
    System.out.println("Date & time: " + cal.getTime());
    System.out.println("Year: " + cal.get(Calendar.YEAR));
    System.out.println("Week of the year: " + cal.get(Calendar.WEEK_OF_YEAR));
    System.out.println("Day of the year: " + cal.get(Calendar.DAY_OF_YEAR));
    // Convert Calendar object to Date object
    Date date = cal.getTime();
    System.out.println("java.util.Date: " + date);
  }
}
```

Output:
Date & time: Wed Nov 07 14:52:49 CST 2018
Year: 2018
Week of the year: 45
Day of the year: 311
java.util.Date: Wed Nov 07 14:52:49 CST 2018

- Read these pages on the Calendar and GregorianCalendar classes:
  - https://www.geeksforgeeks.org/java-util-gregoriancalendar-class-java/
  - https://www.mkyong.com/java/java-date-and-calendar-examples/
- Use this page as a reference on the Calendar abstract class:
  - https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html
- Use this page as a reference on the GregorianCalendar class:
  - https://docs.oracle.com/javase/8/docs/api/java/util/GregorianCalendar.html

Exceptions

- Used to handle exceptional events which disrupt the flow of the program's instructions

- Exceptional event will cause the method to “throw an exception”
- Runtime system looks for a method to handle the exception (exception handler) - referred to as “catching the exception”
- Checked and Unchecked Exceptions
  - Unchecked exceptions - Error class or the RuntimeException class
    - Exceptions that inherit from the Error class should not be handled because they usually indicate a critical error and can rarely be dealt with in the program (e.g., running out of memory)
    - Exceptions that inherit from RuntimeException are the result of programming errors and should also not be handled, but rather corrected through improved programming code

      Here is an example of the unchecked exception ArithmeticException which was thrown because we attempted to divide by zero. Notice there is no error identified in the IDE. This is because the error doesn’t get detected until runtime.

```
3 public class JavaBasics {
4     public static void main(String[] args) {
5         double result = 5/0;
6         System.out.println(result);
7     }
8 }
```

    Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.perscholas.java_basics.JavaBasics.main(JavaBasics.java:5)
```

  - Checked exceptions are those that should be handled
    - Methods that are at risk for throwing a checked exception should handled it with a try-catch block or a throws clause in the header
- A try-catch block is a block of code that might throw an exception

- In order to associate an exception handler with a try block, a catch block must follow it
- No code can be between the try and catch blocks and there can be several catch blocks
- The catch block contains code that is run if the exception handler is invoked

- finally block
  - The finally block always runs when the try block exits
  - Useful for unexpected exceptions or providing cleanup code

Here is an example of the use of the checked exception IOException along with a try-catch-finally block (Note: The FileReader class will be discussed in the Java I/O section). The exception will be thrown because we have not created a file named "example.txt" and placed it in the path specified.

```
6  public class JavaBasics {
7      public static void main(String[] args) throws IOException {
8          FileReader inputStream = null;
9          try
10         {
11             inputStream = new FileReader("example.txt");
12         }
13         catch (IOException e)
14         {
15             System.out.println("File was not found in the path"
16                     + " specified.");
17         }
18         finally
19         {
20             if (inputStream != null) {
21                 inputStream.close();
22             }
23         }
24
25     }
26 }
```

Output:
File was not found in the path specified.

- Read the first few pages of this lesson on Java Exceptions up to "The Finally Block"
  - Start here: https://docs.oracle.com/javase/tutorial/essential/exceptions/
- Read this page for further and more concise explanation of Java Exceptions: https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/
- Use this page as a reference on the Throwable class: https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

- Creating a Custom Exception

First, we will create a new exception class and name it InvalidScoreException which extends the class Exception (Note: The attribute “serialVersionUID” is for serialization/deserialization purposes which is beyond the scope of this lesson but you can read more about this here and here):

```
3   public class InvalidScoreException extends Exception {
4     private static final long serialVersionUID = 1L;
5     /*There are two constructors, one no-arg constructor
6     and a constructor which accepts a string as
7     a parameter*/
8     public InvalidScoreException() {
9       super();
10    }
11    public InvalidScoreException(String eMessage) {
12      super(eMessage);
13    }
14  }
```

Next we will modify the Student class setter method setSqlScore (look in the Student class in the getter/setter section for this method), to test for a valid range of scores (greater than or equal to 0 and less than or equal to 100). If the criteria is not met then the custom exception InvalidScoreException will be thrown:

```
50    public void setSqlScore(double sqlScore) throws InvalidScoreException {
51      if (sqlScore >= 0 && sqlScore <= 100) {
52      this.sqlScore = sqlScore;
53      } else {
54        throw new InvalidScoreException("Invalid score."
55            + " Range of SQL Score is 0 to 100");
56      }
57    }
```

Here is an example of the StudentApp main method attempting to set an sqlScore of 120:

```java
import com.perscholas.student.exceptions.InvalidScoreException;

public class StudentApp {
  public static void main(String[] args) throws InvalidScoreException {
    QEAStudent student1 = new QEAStudent("John","john@doe.com");
    try
    {
      student1.setSqlScore(120);
    }
    catch (InvalidScoreException e)
    {
      System.out.println(e.getMessage());
    }
  }
}
```

Output:
Student count = 1
Invalid score. Range of SQL Score is 0 to 100

Java I/O

- I/O Streams
  - A stream is a sequence of data
  - Applications use streams to read data from a source
  - Applications use output streams to write data to a destination
  - Character Streams
    - Read this page on character streams and try the code in your IDE: https://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
    - Use these pages on FileReader and FileWriter as references:
      - https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
      - https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html

- Here is a demonstration of the FileReader and FileWriter classes (be sure you have created a .txt file called “InputFile.txt” and placed it in the root directory of the Java project you’re currently working with.

```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class InputOutput {
  public static void main(String[] args) throws IOException {
    // Create instances of FileWriter & FileReader classes
    FileWriter fw = null;
    FileReader fr = null;

    try
    {
      fr = new FileReader("InputFile.txt");
      fw = new FileWriter("NewInputFile.txt");
      int nextChar;
      while ((nextChar = fr.read()) != -1) {
        System.out.print((char)nextChar);
        fw.write(nextChar);
      }
    }
    catch (IOException e)
    {
      System.out.println(e.getMessage());
    }
    finally
    {
      /* If we don't test the FileReader instance fr for
       *  null value, then we run the risk of a
       *  NullPointerException if it doesn't get
       *  instantiated due to the file "InputFile.txt"
       *  not being found. This could happen if the file
       *  doesn't exist or if there is a spelling error.*/
      if (fr != null) {
        fr.close(); /*This will throw a NullPointerException
        if fr is not instantiated as there will be no instance
        and therefore no method to call (i.e., fr will point
        to null and hence the NullPointerException.*/
      }
      //The same applies to the FileWriter instance fw.
      if (fw != null) {
        fw.close();
      }
    }
  }
}
```

This application will copy “InputFile.txt” and print it out the content to the console and write it to a new file called “NewInputFile.txt”. If “NewInputFile.txt” doesn’t exist, the application will create the file, but if it does exist (in the project root directory) it will overwrite any text in the file with the text from “InputFile.txt”.

- The PrintWriter class is similar to the FileWriter class. The primary difference is the inclusion of methods that support formatting such printf and format. The PrintWriter class also does not throw IOException.
  - Use this page as a reference for the PrintWriter class: https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html
- The BufferedReader class is similar to FileReader but has the added capability to read input streams by line as demonstrated in the web page on character streams
  - Read these pages on the BufferedReader and BufferedWriter classes
    - https://docs.oracle.com/javase/tutorial/essential/io/buffers.html
    - https://www.mkyong.com/java/how-to-read-file-from-java-bufferedreader-example/
    - https://www.mkyong.com/java/how-to-write-to-file-in-java-bufferedwriter-example/
  - Use this page as a reference on BufferedReader: https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

Here are examples of the use of the BufferedReader and PrintWriter classes:

```
3   import java.io.BufferedReader;
4   import java.io.FileReader;
5   import java.io.FileWriter;
6   import java.io.IOException;
7   import java.io.PrintWriter;
8
9   public class InputOutput {
10    public static void main(String[] args) throws IOException {
11      BufferedReader br = null;
12      PrintWriter pw = null;
13
14      try
15      {
16        br = new BufferedReader(new FileReader("InputFile.txt"));
17        pw = new PrintWriter(new FileWriter("NewOutputFile.txt"));
18        String currentLine;
19        while((currentLine = br.readLine()) != null) {
20          System.out.println(currentLine); // Prints currentLine to console
21          pw.println(currentLine); // Prints currentLine to "NewOutputFile.txt"
22        }
23      }
24      catch (IOException e)
25      {
26        e.getMessage();
27      }
28      finally
29      {
30        if (br != null) {
31          br.close();
32        }
33        if (pw != null) {
34          pw.close();
35        }
36      }
37    }
38  }
```

- Scanner
  - Breaks input into tokens using a delimiter pattern, which by default matches whitespace
  - The Scanner class is not safe for multithreaded use without external synchronization

    - Use this page as a reference on the Scanner class: https://docs.oracle.com/javase/tutorial/essential/io/scanning.html

- File Classes & Methods

  - Copying Files

    - Read this page on copying files: https://docs.oracle.com/javase/tutorial/essential/io/copy.html

    - Use these pages as references on handling files in Java:

      - https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

      - https://www.baeldung.com/java-nio-2-file-api

Examples of the use of the Files class:

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.UUID;

public class InputOutput {
  public static void main(String[] args) throws IOException {
    String s = System.getProperty("user.home");
    System.out.println(s);
    Path p = Paths.get(s);
    /* Note: the Files.exists() method will test for files
     * and directories*/
    System.out.println(Files.exists(p)); // true
    /* The isRegularFile() method will test for only files*/
    System.out.println(Files.isRegularFile(p)); // false

    /*Creating files: Notice the use of the UUID (universally
     * unique identifier) class. Each time the application
     * is run a unique file name is generated.*/
      String fileName = "myfile_" + UUID.randomUUID().toString() + ".txt";
      p = Paths.get(s + "/" + fileName); // This line creates the file
      System.out.println(Files.exists(p)); // false
      Files.createFile(p);
      System.out.println(Files.exists(p)); // true
  }
}
```

- Read this page on deleting files in Java:
  https://docs.oracle.com/javase/tutorial/essential/io/delete.html
- Read this page on moving files or directories in Java:
  https://docs.oracle.com/javase/tutorial/essential/io/move.html
- Use this page as a reference on the IOException class:
  https://docs.oracle.com/javase/8/docs/api/index.html?java/io/IOException.html

• Further reading

- https://docs.oracle.com/javase/tutorial/essential/io/
- https://www.geeksforgeeks.org/different-ways-reading-text-file-java/

Multithreading (Concurrency)

- A multithreaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs
- Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program
- Watch these videos on concurrency in Java
  - https://www.youtube.com/watch?v=Xj1uYKa8rIw
  - https://www.youtube.com/watch?v=RH7G-N2pa8M
- A thread can be in one of five states
  - New: The thread is in new state if you create an instance of the Thread class but before the invocation of the start() method
  - Runnable: a thread executing in the Java virtual machine
  - Blocked: a thread that is blocked and is waiting for a monitor lock
  - Waiting: a thread that is waiting indefinitely for another thread to perform a particular action
  - Timed_Waiting - a thread that is waiting for another thread to perform an action for up to a specified waiting time
  - Terminated: A thread is in terminated or dead state when its run() method exits
- Thread Priorities
  - Java thread priorities are in the range between 1 (MIN_PRIORITY) and 10 (MAX_PRIORITY) with a default of 5 (NORM_PRIORITY)
  - Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities

cannot guarantee the order in which threads execute and are very much platform dependent

- Thread methods
  - start( ) - causes the thread to begin execution
  - sleep( ) - causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
  - run( ) - causes Runnable object's run method to be called
  - join( ) - allows one thread to wait for the completion of another (waits for another thread to terminate)
  - yield( ) - a hint to the scheduler that the current thread is willing to yield its current use of a processor
  - isAlive( ) - boolean method to test if a thread is alive
- Thread-related methods inherited from the Object class
  - wait( ) - causes the current thread to wait until another thread invokes the notify( ) method of the notifyAll( ) method for this object
  - notify( ) - wakes up a single thread that is waiting on this objects monitor
  - notifyAll( ) - wakes up all threads that are waiting on this objects monitor
- Synchronization - assures a thread/process will complete before another thread can access the same resource
- Deadlock - a situation where two or more threads are blocked forever, waiting on each other
- Inter-thread communication
- Pausing a thread with Sleep
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html
- Further Reading

- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- https://www.geeksforgeeks.org/multithreading-in-java/

Java Database Connectivity (JDBC)

- Tutorials
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html
  - https://www.youtube.com/watch?v=5vzCjvUwMXg
- Driver interface
  - Interface that every driver must implement
  - Handles communication between the application and the database
  - DriverManager will try to load as many drivers as it can find and for each request ask the driver to connect to the target URL
  - User can load and register a driver with the statement: Class.forName("foo.bah.Driver")
- DriverManager class
  - Connects an application to a data source specified by a URL
  - Tracks available drivers and attempt to load driver classes referenced in the "jdbc.drivers" system property
  - The getConnection( ) method attempts to establish a connection to the database URL
  - The DataSource interface is preferred to DataManager, but DatabaseManager is easier to use
  - https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html

- Connection interface
  - Serves as the connection (session) between the application and the database
  - SQL statements and results are processed through the Connection instance
  - https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html
- ResultSet interface
  - Provides methods for retrieving and processing the results of an SQL query
  - A table of data representing a database result set generated by running a query of a database
  - https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html
  - https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html
- Statement interface
  - Running Statement objects will return ResultSet objects
  - https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html
  - https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html
  - PreparedStatement interface
    - Accepts input parameters through use of “?” and setXXX(int) methods
    - Used when SQL statement will be used multiple times
    - https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html
    - https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html
- System queries (all schemas)
  - MySql: show schemas
  - Oracle: select USERNAME from SYS.ALL_USERS

- SQL Server: SELECT * FROM sys.schemas