Java 8 features, Java I/O and Reflection

Compiling Topics

- Streams
- Scanning and Formatting
- Directory Operations
- File Operations
- Serialization and Deserialization
- Reflection API
- Java 8 Features

Streams

- An I/O Stream represents an input or an output destination
- Streams supports all types of data i.e. bytes, characters, objects, localized objects, primitive data types etc
- Can perform different operations like reading, writing, deleting, creating new instance etc kind of operations on streams.

Streams

Types of Streams

- Byte Stream
- Character Stream
- Buffered Stream

Byte Stream

- Programs use byte streams to perform input and output of 8-bit bytes.
- All byte stream classes are descended from InputStream and OutputStream.
- For eg, for files we use FileInputStream and FileOutputStream.

Byte Stream

FileInputStream

Commonly used constructor:

- FileInputStream(File file)
- FileInputStream(String fileName)

Commonly used functions:

- read()
- read(byte[] byte)
- close()

Byte Stream

FileOutputStream

Commonly used constructor:

- FileOutputStream(File file)
- FileOutputStream(String fileName)

Commonly used functions:

- write()
- write(byte[] byte)
- close()

Character Stream

Java Byte streams are used to perform input and output of 8-bit bytes, where as Java Character streams are used to perform input and output for 16-bit unicode.

Most frequently used classes are FileReader and FileWriter

Character Stream

FileReader class inherits from the InputStreamReader class

Constructors:	FileReader (File file)
	FileReader (String fileName)
Common methods:	read()
	read(char [] char, int offset, int length)

FileWriter class inherits from the InputStreamReader class

Constructors:	FileWriter (File file)
	FileWriter (String fileName)
Common methods:	write()
	write(char [] char, int offset, int length)
	flush()

Buffered Stream

In case of others streams every piece of data that s being read or write requires an direct support from underlying OS.

- Make a program much less efficient as makes a extensive use of memory and resources

Buffered input streams read data from a memory area known as a buffer.

- While reading API is called only when the buffer is emptyand while writing API is called only when the buffer is full.

Scanning and formatting

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs.

The **scanner API** breaks input into individual tokens associated with bits of data.
The **formatting API** assembles data into nicely formatted, human-readable form.

Directory Operations

- Java empowers us with operaions on directories.
- **File** is the class used for directory operations.

We can perform following actions on directories:

File file = new File(“fileName With Absolute Path”);

- **Create**

file.mkdir(), file.mkdirs()
- **Copy**

FileUtils.copyFile(sourceFileObject, destinationFileObject);
- **Delete**

file.delete()
- **Rename**

file.renameTo(newFileObject)

File Stream

Java also empowers us with operaions on files. We can perform following actions on files:

- **Create**
- **Copy**
- **Delete**
- **Rename**
- **Read**
- **Write**
- **Search files**

Serialization

Serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After serializion, object can be read from the file and deserialized to recreate the object in memory.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

Serialization

Here are some uses and reason why Serialization is needed

- Persist data for future use.
- Send data to a remote computer using such client/server Java technologies as RMI or socket programming.
- "Flatten" an object into array of bytes in memory.
- Store user session in Web applications.

Serialization – caching objects

```
ObjectOutputStream out = new ObjectOutputStream(...);

    Employee e = new Employee();
    e.setHireDate(new Date());
    e.setName("Danny");
    oos.writeObject(e);    //saves Employee state with name="Danny".

    e.setName("John");
    oos.writeObject(e);    // does not save Employee's new state.
```

The solution is to close the stream after every write call.

Serialization – Version Control

Let's say we have an `Employee` instance written to an output stream. Meanwhile, we add a new field to the `Employee` instance, say `String departmentName`. When you try to de-serialize the persisted object, a `java.io.InvalidClassException` would be thrown.

The identifier that is part of all classes is maintained in a field called `serialVersionUID`. If you wish to control versioning, you simply have to provide the `serialVersionUID` field manually and ensure it is always the same, no matter what changes you make to the class file.

Prevent Serialization

Let's say we have a sub-class, which is not marked as `Serializable`, but whose super-class is `Serializable`. In these cases, override the `writeObject()` and `readObject()` methods in your class, and throw a `NotSerializableException`from there:

```
public class SuperClass implements Serializable {
       ….
    }

    public class SubClass extends SuperClass {

        private void readObject(ObjectInputStream ois)
                throws ClassNotFoundException, IOException {
            throw new NotSerializableException();
        }

        private void writeObject(ObjectOutputStream oos)
                throws IOException {
            throw new NotSerializableException();
        }
    }
```

Reflection

- Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime.
- Can instantiate new objects, invoke methods and get/set field values using reflection.
- `Object` class has `getClass()` method, resulting in all Java classes having the method.
- The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class, respectively. All three classes have a method called `getName()` that returns the name of the item.

Reflection

- The `Field` class has a method `getType()` that returns an object, again of type `Class`, that describes the field type.
- The `getFields()`, `getMethods()`, and `getConstructors()` methods of the `Class` class return arrays of the public fields, operations, and constructors that the class supports. This includes public members of superclasses.
- The `getDeclaredFields()`, `getDeclaredMethods()`, and `getDeclaredConstructors()` methods of the `Class` class return arrays consisting of all fields, operations, and constructors that are declared in the class. This includes private and protected members, but not members of superclasses.

- All three of these classes also have a method called `getModifiers()` that returns an integer, with various bits turned on and off, that describe the modifiers used, such as `public` and `static`. You can then use the static methods in the `Modifier` class to analyze the integer that `getModifiers()` returns.
- For example, there are methods like `isPublic()` or `isFinal()` in the `Modifier` class that you could use to tell whether a method or constructor was `public`, `private`, or `final`.

Reflection – access private member

- Default behavior of reflection mechanism is to respect Java control:

```
Employee e = new Employee("harry");;
Field nameField = e.getClass().getField("name");
Object value = nameField.get(e);   // IllegalAccessException
```

- The above code throws exception since `name` is a `private` field in `Employee` class.

- Solution is to invoke `setAccessible(true)` on a `Field`, `Method`,. Or `Constructor` object.

```
Employee e = new Employee("harry");;
Field nameField = e.getClass().getDeclaredField("name");
nameField.setAccessible(true);
Object value = nameField.get(e);       // works fine.
```

Reflection – invoke methods at run-time

- Suppose that `Employee` class has a method `raiseSalary(double)`. To invoke the method at run-time using reflection:

```
Employee employee = new Employee("harry", 2.5);
Class clazz = employee.getClass();
Method method= clazz.getMethod("raiseSalary", new Class[]
  {double.class});

Object[] args = new Object[] {1.5};
method.invoke(employee, args);
```

Java 8 Features (Lambda Expressions)

A lambda expression is characterized by the following syntax –
parameter -> expression body

Lambda expressions are anonymous methods, aimed at mainly addressing the “vertical problem” by replacing the machinery of anonymous inner classes with a lighter-weight mechanism in applicable cases.

Following are the important characteristics of a lambda expression –

- **Optional type declaration** – No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.

Java 8 Features (Lambda Expressions)

- **Optional parenthesis around parameter** – No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
- **Optional curly braces** – No need to use curly braces in expression body if the body contains a single statement.
- **Optional return keyword** – The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

Java 8 Features (Streams)

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream –

- **Sequence of elements** – A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
- **Source** – Stream takes Collections, Arrays, or I/O resources as input source.
- **Aggregate operations** – Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

Java 8 Features (Streams)

- **Pipelining** - Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- **Automatic iterations** - Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Java 8 Features (Streams)

Streams vs Collections

At the basic level, the difference between Collections and Streams has to do with when things are computed. A Collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the Collection has to be computed before it can be added to the Collection. A Stream is a conceptually fixed data structure, in which elements are computed on demand. This gives rise to significant programming benefits. The idea is that a user will extract only the values they require from a Stream, and these elements are only produced—invisibly to the user—as and when required. This is a form of a producer-consumer relationship.

Java 8 Features (Interface with static method)

Prior to Java 8, we could have only method declarations in the interfaces. But from Java 8, we can have default methods and static methods in the interfaces.

Earlier

```
public interface Interface1 {

    void method1(String str);

}

public class example implements Interface1{

void method1(String str){

...}

}
```

```
Adding any method in interface will create error in all classes implementing it
```

Java 8 Features (Interface with static and static method)

Prior to Java 8, we could have only method declarations in the interfaces. But from Java 8, we can have default methods and static methods in the interfaces.

Earlier

```
public interface Interface1 {
    void method1(String str);
}
public class example implements Interface1{
void method1(String str){
...}
}
```

Adding any method in interface will create error in all classes implementing it

Java 8 Features (Interface with static and static method)

Now

```
public interface Interface1 {
    void method1(String str);

    default void log(String str){
         ...
    }

    static boolean isNull(String str) {
         …..
    }
}

public class example implements Interface1{
void method1(String str){
...}
}
```

Adding default or static methods won't generate error with classes already implementing interface

Questions??

Thank you!!