

Core Java

Day 15 Agenda

- Q & A
- Method references
- Java IO Framework

Method references

- If lambda expression involves single method call, it can be shortened by using method reference.
- Method references are converted into instances of functional interfaces.
- Method reference can be used for class static method, class non-static method, object non-static method or constructor.

Examples

- Class static method: `Integer::sum [(a,b) -> Integer.sum(a,b)]`
 - Both lambda param passed to static function explicitly
- Class non-static method: `String::compareTo [(a,b) -> a.compareTo(b)]`
 - First lambda param become implicit param (this) of the function and second is passed explicitly (as arguments).
- Object non-static method: `System.out::println [x -> System.out.println(x)]`
 - Lambda param is passed to function explicitly.
- Constructor: `Date::new [() -> new Date()]`
 - Lambda param is passed to constructor explicitly.

Java IO framework

- Input/Output functionality in Java is provided under package `java.io` and `java.nio` package.
- IO framework is used for File IO, Network IO, Memory IO, and more.
- File is a collection of data and information on a storage device.
- File = Data + Metadata

- Two types of APIs are available file handling
 - FileSystem API -- Accessing/Manipulating Metadata
 - File IO API -- Accessing/Manipulating Contents/Data

java.io.File class

- A path (of file or directory) in file system is represented by "File" object.
- Used to access/manipulate metadata of the file/directory.
- Provides FileSystem APIs
 - String[] list() -- return contents of the directory
 - File[] listFiles() -- return contents of the directory
 - boolean exists() -- check if given path exists
 - boolean mkdir() -- create directory
 - boolean mkdirs() -- create directories (child + parents)
 - boolean createNewFile() -- create empty file
 - boolean delete() -- delete file/directory
 - boolean renameTo(File dest) -- rename file/directory
 - String getAbsolutePath() -- returns full path (drive:/folder/folder/...)
 - String getPath() -- return path
 - File getParentFile() -- returns parent directory of the file
 - String getParent() -- returns parent directory path of the file
 - String getName() -- return name of the file/directory
 - static File[] listRoots() -- returns all drives in the systems.
 - long getTotalSpace() -- returns total space of current drive
 - long getFreeSpace() -- returns free space of current drive
 - long getUsableSpace() -- returns usable space of current drive
 - boolean isDirectory() -- return true if it is a directory
 - boolean isFile() -- return true if it is a file
 - boolean isHidden() -- return true if the file is hidden
 - boolean canExecute()
 - boolean canRead()
 - boolean canWrite()

- `boolean setExecutable(boolean executable)` -- make the file executable
- `boolean setReadable(boolean readable)` -- make the file readable
- `boolean setWritable(boolean writable)` -- make the file writable
- `long length()` -- return size of the file in bytes
- `long lastModified()` -- last modified time
- `boolean setLastModified(long time)` -- change last modified time

Java IO

- Java File IO is done with Java IO streams.
- Stream is abstraction of data source/sink.
 - Data source -- `InputStream` or `Reader`
 - Data sink -- `OutputStream` or `Writer`
- Java supports two types of IO streams.
 - Byte streams (binary files) -- byte by byte read/write
 - Character streams (text files) -- char by char read/write
- All these streams are `AutoCloseable` (so can be used with `try-with-resource` construct)

Chaining IO Streams

- Each IO stream object performs a specific task.
 - `FileOutputStream` -- Write the given bytes into the file (on disk).
 - `BufferedOutputStream` -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
 - `DataOutputStream` -- Convert primitive types into sequence of bytes. Inherited from `DataOutput` interface.
 - `ObjectOutputStream` -- Convert object into sequence of bytes. Inherited from `ObjectOutput` interface.
 - `PrintStream` -- Convert given input into formatted output.
 - Note that input streams does the counterpart of `OutputStream` class hierarchy.
- Streams can be chained to fulfil application requirements.

Primitive types IO

- `DataInputStream` & `DataOutputStream` -- convert primitive types from/to bytes
 - primitive type --> `DataOutputStream` --> bytes --> `FileOutputStream` --> file.

- DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
- primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
- DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...

DataOutput/DataInput interface

- interface DataOutput
 - writeUTF(String s)
 - writeInt(int i)
 - writeDouble(double d)
 - writeShort(short s)
 - ...
- interface DataInput
 - String readUTF()
 - int readInt()
 - double readDouble()
 - short readShort()
 - ...

Serialization

- ObjectOutputStream & ObjectOutputStream -- convert java object from/to bytes
 - Java object --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
 - ObjectOutputStream interface provides method for conversion - writeObject().
 - Java object <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
 - ObjectInput interface provides methods for conversion - readObject().
- Converting state of object into a sequence of bytes is referred as Serialization. The sequence of bytes includes object data as well as metadata.
- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).
- Converting (serialized) bytes back to the Java object is referred as Deserialization.

- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).
- demo16_04 -- Movie object read/write
- demo16_05 -- `List<Movie>` object read/write

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
 - `writeObject(obj)`
- interface ObjectInput extends DataInput
 - `obj = readObject()`

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise `writeObject()` throws `NotSerializableException`.
- Serializable is a marker interface.

transient fields

- `writeObject()` serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".
- The transient and static fields (except `serialVersionUID`) are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a `serialVersionUID`.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this `serialVersionUID`.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then `InvalidClassException` will be thrown.
- If a serializable class does not explicitly declare a `serialVersionUID`, then the runtime will calculate a default `serialVersionUID` value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specification).

Buffered streams

- Each `write()` operation on `FileOutputStream` will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- `BufferedOutputStream` classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
 - Java object --> `ObjectOutputStream` --> `BufferedOutputStream` --> `FileOutputStream` --> file on disk.
- Data is sent to underlying stream when buffer is full or `flush()` called explicitly.
- `BufferedInputStream` provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

PrintStream class

- Produce formatted output (in bytes) and send to underlying stream.
- Formatted output is done using methods `print()`, `println()`, and `printf()`.
- `System.out` and `System.err` are objects of `PrintStream` class.

Scanner class

- Added in Java 5 to get the formatted input.
- It is `java.util` package (not part of `java.io` framework).

```
Scanner sc = new Scanner(inputStream);  
// OR  
Scanner sc = new Scanner(inputFile);
```

- Helpful to read text files line by line.

Character streams

- Character streams are used to interact with text file.
- Java char takes 2 bytes (unicode), however char stored in disk file may take 1 or more bytes depending on char encoding.
 - <https://www.w3.org/International/questions/qa-what-is-encoding>
- The character stream does conversion from java char to byte representation and vice-versa (as per char encoding).
- The abstract base classes for the character streams are the Reader and Writer class.
- Writer class -- write operation
 - void close() -- close the stream
 - void flush() -- writes data (in memory) to underlying stream/device.
 - void write(char[] b) -- writes char array to underlying stream/device.
 - void write(int b) -- writes a char to underlying stream/device.
- Writer Sub-classes
 - FileWriter, OutputStreamWriter, PrintWriter, BufferedWriter, etc.
- Reader class -- read operation
 - void close() -- close the stream
 - int read(char[] b) -- reads char array from underlying stream/device
 - int read() -- reads a char from the underlying device/stream. Returns -1
- Reader Sub-classes
 - FileReader, InputStreamReader, BufferedReader, etc.

Java NIO

- Java NIO (New IO) is an alternative IO API for Java.
- Java NIO offers a different IO programming model than the traditional IO APIs.
- Since Java 7.
- Java NIO enables you to do non-blocking (not fully) IO.
- Java NIO consist of the following core components:
 - Channels e.g. FileChannel, ...
 - Buffers e.g. ByteBuffer, ...
 - Selectors
- Java NIO also provides "helper" classes Paths & Files.

- exists()
- ...

Paths and Files

- A Java Path instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

- Files class (Files) provides several static methods for manipulating files in the file system.

```
static InputStream newInputStream(Path, OpenOption...) throws IOException;
static OutputStream newOutputStream(Path, OpenOption...) throws IOException;
static DirectoryStream<Path> newDirectoryStream(Path) throws IOException;
static Path createFile(Path, attribute.FileAttribute<?>...) throws IOException;
static Path createDirectory(Path, attribute.FileAttribute<?>...) throws IOException;
static void delete(Path) throws IOException;
static boolean deleteIfExists(Path) throws IOException;
static Path copy(Path, Path, CopyOption...) throws IOException;
static Path move(Path, Path, CopyOption...) throws IOException;
static boolean isSameFile(Path, Path) throws IOException;
static boolean isHidden(Path) throws IOException;
static boolean isDirectory(Path, LinkOption...);
static boolean isRegularFile(Path, LinkOption...);
static long size(Path) throws IOException;
static boolean exists(Path, LinkOption...);
static boolean isReadable(Path);
static boolean isWritable(Path);
static boolean isExecutable(Path);
static List<String> readAllLines(Path) throws IOException;
static Stream<String> lines(Path) throws IOException;
```


Homework

- Read docs of RandomAccessFile.

Assignment

1. In menu-driven program of Books (with ArrayList - Day12 Q1) add two more menus. Save books in file and load books from file. Use DataOutputStream and DataInputStream.
2. In menu-driven program of Books (with ArrayList - Day12 Q1) add two more menus. Save books in file and load books from file. Use ObjectOutputStream and ObjectInputStream.
3. Write a program that inputs 4 lines and stored them in a text file. Use BufferedWriter class.

```
for(int i=1; i<=4; i++) {  
    String line = sc.nextLine();  
    bwr.write(line);  
}
```

4. Read a text file line by line and display on terminal. Hint: file --> FileReader --> BufferedReader --> ...

```
// ...  
String line = brd.readLine();
```