Core JDK APIs (part 1)

Agenda

- Overview
- Lambdas
- Streams
- Basic IO

Agenda

- Cloning objects
- Serializing objects
- Logging
- Process handling

Overview

Java SE

- Apart from the Core language Java SE (standard edition) comes with a number of useful utilities …

collections	Io/nio	date & time
object comparison	object cloning	XML/JSON parsing
concurrency	object serialization	security
networking	logging	GUI APIs
JDBC	process API	JShell
JMX	reflection	
lambdas	regex	
streams	locales	

In blue are the ones already covered and in green are the ones for which there is a dedicated session

Lambdas

Overview

- Lambdas bring anonymous function types in Java (JSR 335):

```
(parameters) -> {body}
```

- Example:

```
(x,y) -> x + y
```

… well … Groovy (a JVM language which is a superset of Java) already has lambdas

Functional interfaces

- Lambdas can be used in place of functional interfaces (interfaces with just one method such as **Runnable**)

- Example:

```
new Thread(new Runnable() {

   @Override
   public void run() {
       System.out.println(
              "It runs !");
       }
   }
).start();
```

⇨

```
 new Thread(() -> {
System.out.println(
  "It runs !"); })
      .start();
```

Functional interfaces

- Examples of functional interfaces include:

java.lang.Runnable -> run()

java.util.concurrent.Callable -> call()

java.security.PrivilegedAction -> run()

java.util.Comparator -> compare(T o1, T o2)

java.awt.event.ActionListener ->
    actionPerformed (ActionEvent e)

java.lang.Iterable ->
    forEach(Consumer<? super T> action)

java.lang.Iterable#forEach() is not breaking backward compatibility because it is an extension method (default)

Functional interfaces

- Extension methods provide a mechanism for extending an existing interface without breaking backward compatibility

```
public interface Iterable<T> {

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
}
```

Functional interfaces

- Additional functional interfaces are provided by the **java.util.function** package for use by lambdas such as:
  - **Predicate<T>** - one method with param of type T and boolean return type
  - **Consumer<T>** - one method with param T and no return type
  - **Function<T, R>** - one method with param T and return type R
  - **Supplier<T>** - one method with no params and return type T

Lambda performance

- This essentially means that lambdas can be used to replace all functional interfaces used in a gazillion of libraries

- Moreover lambdas increase performance when used instead of anonymous inner classes because they are implemented with the **invokedynamic** instruction

- In this regard many of the standard JDK class

  are being refactored to use lambdas

Streams

The Streams API

- Databases and other programming languages allow us to specify aggregate operations explicitly

- The streams API provides this mechanism in the Java platform

- The notion of streams is derived from functional programming languages

The Streams API

- The stream API makes use of lambdas and extension methods
- Streams can be applied on collections, arrays, IO streams and generator functions
- Streams can be finite or infinite
- Streams can apply intermediate functions on the data that produce another stream (e.g. map, reduce)

The Streams API

- Streams are represented by a java.util.stream.Stream<T> instance

- Streams in the JDK can also be parallel

```
collection.stream();
collection.parallelStream();
```

- Stream API methods include:
  - filter(Predicate), map(Function), reduce(BinaryOperator), collect()
  - sum(), min(), max(), count()
  - anyMatch(), allMatch(), forEach()

Streams

- Stream operations are composed into a pipeline
- Streams are lazy: computation is performed when the terminal operation is invoked

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

Method references

- Intended to be used in lambda expressions, preventing unnecessary boilerplate

```
books.stream().map(b -> b.getTitle())
     books.stream().map(Book::getTitle)
```

- Lambda parameter list and return type must match the signature of the method

- There are four kinds of method references:

  - to static method
  - to instance method (from class)
  - to instance method (from instance)
  - to constructor

Method references: static

```
public class Printers {
      public static void print(String s) {...}
}

Arrays.asList("a", "b", "c").forEach(Printers::print)
```

Method references: instance

```
public class Document {
     public String getPageContent(int pageNumber) {
          return this.pages.get(pageNumber)
               .getContent();
     }
}
```

```
public static void printPages(Document doc, int[]
pageNumbers) {
   Arrays.stream(pageNumbers)
        .map(doc::getPageContent)
        .forEach(Printers::print);
 }
```

```
public static void printDocuments(List<Page> pages) {
   pages.stream()
        .map(Page::getContent)
        .forEach(Printers::print);
 }
```

Method references: constructor

```
public static Stream<Page> createPagesFrom(
        Stream<String> contents) {
    return contents.map(Page::new)
}
```

Basic IO

Overview

- Java IO provides utilities for working with input and output streams

- An extension of these utilities is provided by the Java NIO (NIO stands for **non-blocking IO**) classes

- The **java.io** and **java.nio** packages are provided for the purposes

Capabilities

- The main capabilities provides by the IO utilities are:
  - reading/writing to/from the standard input/output
  - reading/writing files
  - reading/writing special formats of files such as ZIP and JAR
  - serializing data
  - inter-thread communication (pipes)
  - communication over the network

Hierarchy

- **InputStream** and **OutputStream** instances are used mostly for byte-oriented streams

- **Reader** and **Writer** instances are used for character-oriented streams

- All of them are defined in the **java.io** package

IO Classes

	Input	Output	Reader	Writer
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Formatted data		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		

credits: http://tutorials.jenkov.com/java-io/overview.html

Standard input

```
BufferedReader reader = new BufferedReader(
      new InputStreamReader(System.in));
String line = reader.readLine();
```

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
int next = in.nextInt();
```

```
String name = System.console().readLine();
```

Cloning objects

Object cloning

- Java SE defines a standard mechanism for cloning of objects

- Objects of a class can be cloned if:

  – The class implements the **java.lang.Cloneable** marker interface (does not contain any methods)

  – Implements properly the **clone()** method defined in the java.lang.Object class

Object cloning

- Main benefit of object cloning is the possibility to reduce the amount of boilerplate code written
- Two types of object copies can be created:
  - shallow: only references to the objects are cloned but the underlying data is shared
  - full: the object references and the underlying data is copied. The clones do not share data

Object#clone()

- If the clone() method is called on an object that does not implement the Cloneable interface a **CloneNotSupportedException** is thrown

- By default the clone() method creates a shallow copy if not overriden

Separate method for creating a shallow/deep copy can be implemented by that should be avoided

Deep copy

```
public class Table implements Cloneable {

      private int size;

      private Location location;

      @Override
      protected Object clone() throws
                  CloneNotSupportedException {
            Table cloned = (Table)super.clone();
            cloned.setLocation((Location)
                  location.clone());
            return cloned;
      }
}
```

Serializing objects

Object serialization

- Object serialization provides a mechanism to state the save an object
- Java SE provides a standard mechanism for implementing object serialization
- Classes of serializable objects need to implement the **java.io.Serializable** marker interface

Object serialization

- Objects can be serialized using a **java.io.ObjectOutputStream** instance
- Objects can be deserialized using a **java.io.ObjectInputStream** instance
- The object is serialized as a stream of bytes
- The JVM associates a serialization ID with each Serializable class for object compatibility

Object serialization

- One assumption is that the object and all of its objects fields implement the **Serializable** interface

- If there is a field that is not serializable then a **NotSerializableException** is thrown

- In many scenarios that is a problem as we cannot modify the classes of some objects we reference (i.e. ones coming from third-party libraries) !

Custom serialization

- In order to avoid that problem we can define custom serialization for the class by:
  - Implementing the **readObject** and **writeObject** methods in the class

```
private void writeObject(ObjectOutputStream oos) {
        …
}

private void readObject(ObjectInputStream ois) {
        …
}
```

  - Marking the non-serializable fields as **transient**

transient is also used by ORM frameworks like Hibernate to mark fields as non-persistable

Custom serialization

```
public class Table implements Serializable {

      private int size;

      private transient Location location;

      private void writeObject(ObjectOutputStream os)
            throws IOException {
            // writes non-transient fields
            os.defaultWriteObject();
      }

       private void readObject(ObjectInputStream is)
            throws IOException, ClassNotFoundException {
             // writes non-transient fields
             is.defaultReadObject();
       }
```

Alternatives

- Standard Java serialization is not the only mechanism that can be used for object serialization ....
- Many applications convert objects to a suitable format such as XML, JSON or YML for serialization (and further store in a file, DB or send it over the wire)
- Objects can also be persisted using an ORM framework such as Hibernate

In the next session we are covering conversion to XML and JSON

There are dedicated sessions on persistence with Hibernate

Logging

Java logging framework

- The Java SE logging framework is provided by the **java.util.logging** package

- The root class of the framework is **java.util.logging.Logger**

- Logger instances are typically bound to the class that uses them as static fields

```
private final static Logger LOGGER =
        Logger.getLogger(Table.class.getName());
```

Java logging framework

- Each logged entry has a log level
- Loggers are hierarchical based on the package
- Loggers may define filters for the log messages
- Loggers may use different handlers as output sources for the log entries (console, file, etc.)

Java logging framework

- The **java.util.logging** framework lacks capabilities provides by third-party logging frameworks (especially in terms of configuration)

- A number of third-party logging frameworks are more preferable in practice:

  – Log4J
  – Logback
  – Apache Commons Logging (a logging façade)
  – SLF4j (a logging facade)

SLF4j over Log4j is one of the most preferable choices due to the wide adoption by third-party libraries

Process Handling

External processes

- Before Java 9 …

```
Process p = Runtime.getRuntime().exec("cmd /c notepad");
```

```
ProcessBuilder pb = new ProcessBuilder("cmd", "/c",
      "notepad");
Process p = pb.start();
```

- Limited support for process handling up to JDK 9

- Mostly for creation of a process …

External processes

```
public static void startProcess() throws IOException,
        InterruptedException, ExecutionException {
        Process process = Runtime.getRuntime()
                .exec("cmd /c notepad");
        LOGGER.info("PID: " + process.pid());
        LOGGER.info("Number of children: " +
                process.children().count());
        LOGGER.info("Number of descendants: " +
                process.descendants().count());
        ProcessHandle.Info info = process.info();
        LOGGER.info("Process command: " + info.command());
        LOGGER.info("Info: " + info.toString());
        CompletableFuture<Process> exitedFuture =
                process.onExit();
        exitedFuture.whenComplete((p, e) -> {
                LOGGER.info("Process exited ... ");});
        exitedFuture.get();
}
```

New process API

- **java.lang.Process** extended with new methods as of JDK 9

```
children()              onExit()
descendants()           supportsNormalTermination()
pid()                   toHandle()
info()
```

- New **java.lang.ProcessHandle** class for process management in a streamlike manner

New process API

```
public static long getCurrentPid() {
        long pid = ProcessHandle.current().pid();
        LOGGER.info("PID: " + pid);
        return pid;
}
```

New process API

```
public static ProcessHandle[] getProcessesByName(
      String name) {

   ProcessHandle[] processes = ProcessHandle.allProcesses()
      .filter((pHandle) -> { return
            pHandle.info().toString().contains(name);
      }).toArray(ProcessHandle[] :: new);

   for(ProcessHandle process : processes) {
      LOGGER.info("Process details: " +
            process.info().toString());
   }

    return processes;
}
```

Questions ?