Core Java

Agenda

- Java IO - Character streams
- Stream programming
- Method references
- Reflection
- Annotations

IO Framework

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 8 Streams

- Java 8 Stream is NOT IO streams.
- java.util.stream package.
- Streams follow functional programming model in Java 8.
- The functional programming is based on functional interface (SAM).
- Number of predefined functional interfaces added in Java 8. e.g. Consumer, Supplier, Function, Predicate, ...
- Lambda expression is short-hand way of implementing SAM -- arg types & return type are inferred.
- Java streams represents pipeline of operations through which data is processed.
- Stream operations are of two types
  - Intermediate operations: Yields another stream.
    - filter()
    - map(), flatMap()
    - limit(), skip()
    - sorted(), distinct()
  - Terminal operations: Yields some result.
    - reduce()
    - forEach()
    - collect(), toArray()
    - count(), max(), min()
  - Stream operations are higher order functions (take functional interfaces as arg).

Java stream characteristics

- No storage: Stream is an abstraction. Stream doesn't store the data elements. They are stored in source collection or produced at runtime.
- Immutable: Any operation doesn't change the stream itself. The operations produce new stream of results.
- Lazy evaluation: Stream is evaluated only if they have terminal operation. If terminal operation is not given, stream is not processed.
- Not reusable: Streams processed once (terminal operation) cannot be processed again.

Stream creation

- Collection interface: stream() or parallelStream()
- Arrays class: Arrays.stream()
- Stream interface: static of() method
- Stream interface: static generate() method
- Stream interface: static iterate() method
- Stream interface: static empty() method
- nio Files class: `static Stream<String> lines(filePath)` method

Stream creation

- Collection interface: stream() or parallelStream()

```
List<String> list = new ArrayList<>();
// ...
Stream<String> strm = list.stream();
```

- Arrays class: Arrays.stream()
- Stream interface: static of() method

```
Stream<Integer> strm = Stream.of(arr);
```

- Stream interface: static generate() method
  - generate() internally calls given Supplier in an infinite loop to produce infinite stream of elements.

```
Stream<Double> strm = Stream.generate(() -> Math.random()).limit(25);
```

```
Random r = new Random();
Stream<Integer> strm = Stream.generate(() -> r.nextInt(1000)).limit(10);
```

- Stream interface: static iterate() method
  - iterate() start the stream from given (arg1) "seed" and calls the given UnaryOperator in infinite loop to produce infinite stream of elements.

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(10);
```

- Stream interface: static empty() method
- nio Files class: static Stream lines(filePath) method

Stream operations

- Source of elements

```
String[] names = {"Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh", "Rohan", "Pradnya", "Rohan",
"Pooja", "Lalita"};
```

- Create Stream and display all names

```
Stream.of(names)
    .forEach(s -> System.out.println(s));
```

- filter() -- Get all names ending with "a"
  - `Predicate<T>`: (T) -> boolean

```
Stream.of(names)
    .filter(s -> s.endsWith("a"))
    .forEach(s -> System.out.println(s));
```

- map() -- Convert all names into upper case
  - `Function<T,R>`: (T) -> R

```
Stream.of(names)
    .map(s -> s.toUpperCase())
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in ascending order
  - String class natural ordering is ascending order.
  - sorted() is a stateful operation (i.e. needs all element to sort).

```
Stream.of(names)
    .sorted()
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in descending order
  - `Comparator<T>`: (T,T) -> int

```
Stream.of(names)
    .sorted((x,y) -> y.compareTo(x))
    .forEach(s -> System.out.println(s));
```

- skip() & limit() -- leave first 2 names and print next 4 names

```
Stream.of(names)
    .skip(2)
    .limit(4)
    .forEach(s -> System.out.println(s));
```

- distinct() -- remove duplicate names
    - duplicates are removed according to equals().

```
Stream.of(names)
    .distinct()
    .forEach(s -> System.out.println(s));
```

- count() -- count number of names
    - terminal operation: returns long.

```
long cnt = Stream.of(names)
    .count();
System.out.println(cnt);
```

- collect() -- collects all stream elements into an collection (list, set, or map)

```
List<String> list = Stream.of(names)
        .collect(Collectors.toList());
// Collectors.toList() returns a Collector that can collect all stream elements into a list
```

```
Set<String> set = Stream.of(names)
    .collect(Collectors.toSet());
// Collectors.toSet() returns a Collector that can collect all stream elements into a set
```

- reduce() -- addition of 1 to 5 numbers

```
int result = Stream
    .iterate(1, i -> i+1)
    .limit(5)
    .reduce(0, (x,y) -> x + y);
```

- max() -- find the max string
  - terminal operation
  - See examples.

Optional<> type

- Few stream operations yield Optional<> value.
- Optional value is a wrapper/box for object of T type or no value.
- It is safer way to deal with null values.
- Get value in the Optional<>:
  - optValue = opt.get();
  - optValue = opt.orElse(defValue);
- Consuming Optional<> value:
  - opt.isPresent() --> boolean;
  - opt.ifPresent(consumer);

Collect Stream result

- Collecting stream result is terminal operation.

- Object[] toArrray()
- R collect(Collector)
    - Collectors.toList(), Collectors.toSet(), Collectors.toCollection(), Collectors.joining()
    - Collectors.toMap(key, value)

Stream of primitive types

- Efficient in terms of storage and processing. No auto-boxing and unboxing is done.
- IntStream class
    - IntStream.of() or IntStream.range() or IntStream.rangeClosed() or Random.ints()
    - sum(), min(), max(), average(), summaryStatistics(),
    - OptionalInt reduce().

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.

Reflection

- .class = Byte-code + Meta-data + Constant pool + ...

- When class is loaded into JVM all the metadata is stored in the object of java.lang.Class (heap area).
- This metadata includes class name, super class, super interfaces, fields (field name, field type, access modifier, flags), methods (method name, method return type, access modifier, flags, method arguments, ...), constructors (access modifier, flags, ctor arguments, ...), annotations (on class, fields, methods, ...).

Reflection applications

- Inspect the metadata (like javap)
- Build IDE/tools (Intellisense)
- Dynamically creating objects and invoking methods
- Access the private members of the class

Get the java.lang.Class object

- way 1: When you have class-name as a String (taken from user or in properties file)

```
Class<?> c = Class.forName(className);
```

- way 2: When the class is in project/classpath.

```
Class<?> c = ClassName.class;
```

- way 3: When you have object of the class.

```
Class<?> c = obj.getClass();
```

Access metadata in java.lang.Class

- Name of the class

```
String name = c.getName();
```

- Super class of the class

```
Class<?> supcls = c.getSuperclass();
```

- Super interfaces of the class

```
Class<?> supintf[] = c.getInterfaces();
```

- Fields of the class

```
Field[] fields = c.getFields(); // all fields accessible (of class & its super class)
```

```
Field[] fields = c.getDeclaredFields(); // all fields in the class
```

- Methods of the class

```
Method[] methods = c.getMethods(); // all methods accessible (of class & its super class)
```

```
Method[] methods = c.getDeclaredMethods(); // all methods in the class
```

- Constructors of the class

```
Constructor[] ctors = c.getConstructors(); // all ctors accessible (of class & its super class)
```

```
Constructor[] ctors = c.getDeclaredConstructor(); // all ctors in the class
```

Invoking method dyanmically

~~~
```Java
public class Middleware {
    public static Object invoke(String className, String methodName, Class[] methodParamTypes, Object[]
methodArgs) throws Exception {
        // load the given class
        Class c = Class.forName(className);
        // create object of that class
        Object obj = c.newInstance(); // also invokes param-less constructor
        // find the desired method
        Method method = c.getDeclaredMethod(methodName, methodParamTypes);
        // allow to access the method (irrespective of its access specifier)
        method.setAccessible(true);
        // invoke the method on the created object with given args & collect the result
        Object result = method.invoke(obj, methodArgs);
        // return the results
        return result;
    }
}
```
~~~

~~~~
```Java
// invoking method statically
Date d = new Date();
String result = d.toString();
```
```Java
// invoking method dyanmically
String result = Middleware.invoke("java.util.Date", "toString", null, null);
```
~~~~

Reflection Tutorial

- Refer at Home, if required.
- Part 1: https://youtu.be/lAoNJ_7LD44
- Part 2: https://youtu.be/UVWdtk5ibK8

Annotations

- Added in Java 5.0.
- Annotation is a way to associate metadata with the class and/or its members.
- Annotation applications
  - Information to the compiler
  - Compile-time/Deploy-time processing
  - Runtime processing
- Annotation Types
  - Marker Annotation: Annotation is not having any attributes.
    - @Override, @Deprecated, @FunctionalInterface ...
  - Single value Annotation: Annotation is having single attribute -- usually it is "value".
    - @SuppressWarnings("deprecation"), ...
  - Multi value Annotation: Annotation is having multiple attribute
    - @RequestMapping(method = "GET", value = "/books"), ...

Pre-defined Annotations

- @Override
  - Ask compiler to check if corresponding method (with same signature) is present in super class.
  - If not present, raise compiler error.
- @FunctionalInterface
  - Ask compiler to check if interface contains single abstract method.
  - If zero or multiple abstract methods, raise compiler error.
- @Deprecated
  - Inform compiler to give a warning when the deprecated type/member is used.
- @SuppressWarnings
  - Inform compiler not to give certain warnings: e.g. deprecation, rawtypes, unchecked, serial, unused
  - @SuppressWarnings("deprecation")
  - @SuppressWarnings({"rawtypes", "unchecked"})
  - @SuppressWarnings("serial")
  - @SuppressWarnings("unused")

Meta-Annotations

- Annotations that apply to other annotations are called meta-annotations.
- Meta-annotation types defined in java.lang.annotation package.

@Retention

- RetentionPolicy.SOURCE
  - Annotation is available only in source code and discarded by the compiler (like comments).
  - Not added into .class file.
  - Used to give information to the compiler.
  - e.g. @Override, ...
- RetentionPolicy.CLASS
  - Annotation is compiled and added into .class file.
  - Discared while class loading and not loaded into JVM memory.
  - Used for utilities that process .class files.
  - e.g. Obfuscation utilities can be informed not to change the name of certain class/member using @SerializedName, ...

- RetentionPolicy.RUNTIME
  - Annotation is compiled and added into .class file. Also loaded into JVM at runtime and available for reflective access.
  - Used by many Java frameworks.
  - e.g. @RequestMapping, @Id, @Table, @Controller, ...

@Target

- Where this annotation can be used.
- ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE, TYPE_PARAMETER, TYPE_USE
- If annotation is used on the other places than mentioned in @Target, then compiler raise error.

@Documented

- This annotation should be documented by javadoc or similar utilities.

@Repeatable

- The annotation can be repeated multiple times on the same class/target.

@Inherited

- The annotation gets inherited to the sub-class and accessible using c.getAnnotation() method.

Custom Annotation

- Annotation to associate developer information with the class and its members.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME) // the def attribute is considered as "value" = @Retention(value =
RetentionPolicy.RUNTIME )
@Taget({TYPE, CONSTRUCTOR, FIELD, METHOD}) // { } represents array
@interface Developer {
    String firstName();
    String lastName();
    String company() default "Sunbeam";
    String value() default "Software Engg";
}

@Repeatable
@Retention(RetentionPolicy.RUNTIME)
@Taget({TYPE})
@interface CodeType {
    String[] value();
}
```

```
//@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director") // compiler error --
@Developer is not @Repeatable
@CodeType({"businessLogic", "algorithm"})
@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director")
class MyClass {
    // ...
    @Developer(firstName="Rajiv", lastName="K", company="Sunbeam Pune")
    private int myField;
    @Developer(firstName="Rahul", lastName="Sansuddi")
    public MyClass() {

    }
    @Developer(firstName="Yogesh", lastName="K", company="Sunbeam Pune")
    public void myMethod() {
        @Developer(firstName="James", lastName="Bond") // compiler error
        int localVar = 1;
    }
}
```

```java
// @Developer is inherited
@CodeType("frontEnd")
@CodeType("businessLogic") // allowed because @CodeType is @Repeatable
class YourClass extends MyClass {
    // ...
}
```

Annotation processing (using Reflection)

```java
Annotation[] anns = MyClass.class.getDeclaredAnnotations();
for (Annotation ann : anns) {
    System.out.println(ann.toString());
    if(ann instanceof Developer) {
        Developer devAnn = (Developer) ann;
        System.out.println(" - Name: " + devAnn.firstName() + " " + devAnn. lastName());
        System.out.println(" - Company: " + devAnn.company());
        System.out.println(" - Role: " + devAnn.value());
    }
}
System.out.println();

Field field = MyClass.class.getDeclaredField("myField");
anns = field.getAnnotations() ;
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();

//anns = YourClass.class.getDeclaredAnnotations();
anns = YourClass.class.getAnnotations();
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();
```

Annotation tutorials

- Part 1: https://youtu.be/7zjWPJqlPRY
- Part 2: https://youtu.be/CafN2ABJQcg

Java Proxies (Tutorials)

- Not in our/C-DAC syllabus, but a useful topic.
- Part 1: https://youtu.be/4X_sZNOeR7g
- Part 2: https://youtu.be/jRv3GJuaudA

Assignments

1. Calculate the factorial of the given number using stream operations.
2. Write a program to calculate sum of 10 random integers using streams.
3. Create an IntStream to represent numbers from 1 to 10. Call various functions like sum(), summaryStatistics() and observe the output.