Core Java

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(),

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.

enum

- "enum" keyword is added in Java 5.0.
- Used to make constants to make code more readable.
- Typical switch case

```
int choice;
// ...
switch(choice) {
    case 1: // addition
        c = a + b;
```

```
            break;
        case 2: // subtraction
            c = a - b;
            break;
        // ...
    }
```

- The switch constants can be made more readable using Java enums.

```
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION;
}

ArithmeticOperations choice = ArithmeticOperations.ADDITION;
// ...
switch(choice) {
    case ADDITION:
        c = a + b;
        break;
    case SUBTRACTION:
        c = a - b;
        break;
    // ...
}
```

- In java, enums cannot be declared locally (within a method).
- The declared enum is converted into enum class.

```
// user-defined enum
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION;
}
```

```
// generated enum code
final class ArithmeticOperations extends Enum {
    public static ArithmeticOperations[] values() {
        return (ArithmeticOperations[])$VALUES.clone();
    }
    public static ArithmeticOperations valueOf(String s) {
        return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations, s);
    }
    private ArithmeticOperations(String name, int ordinal) {
        super(name, ordinal); // invoke sole constructor Enum(String,int);
    }
    public static final ArithmeticOperations ADDITION;
    public static final ArithmeticOperations SUBTRACTION;
    public static final ArithmeticOperations MULIPLICATION;
```

```
    public static final ArithmeticOperations DIVISION;
    private static final ArithmeticOperations $VALUES[];
    static {
        ADDITION = new ArithmeticOperations("ADDITION", 0);
        SUBTRACTION = new ArithmeticOperations("SUBTRACTION", 1);
        MULIPLICATION = new ArithmeticOperations("MULIPLICATION", 2);
        DIVISION = new ArithmeticOperations("DIVISION", 3);
        $VALUES = (new ArithmeticOperations[] {
            ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
        });
    }
}
```

- The enum type declared is implicitly inherited from java.lang.Enum class. So it cannot be extended from another class, but enum may implement interfaces.
- The enum constants declared in enum are public static final fields of generated class. Enum objects cannot be created explicitly (as generated constructor is private).
- The generated class will have a values() method that returns array of all constants and valueOf() method to convert String to enum constant.
- The enums constants can be used in switch-case and can also be compared using == operator.
- The java.lang.Enum class has following members:

```
public abstract class Enum<E> implements java.lang.Comparable<E>,
java.io.Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String,int); // sole constructor - can be called from
user-defined enum class only
    public final String name(); // name of enum const
    public final int ordinal(); // position of enum const (0-based)

    public String toString(); // returns name of const
    public final int compareTo(E);// compares with another enum of same type
on basis of ordinal number
    public static <T> T valueOf(Class<T>, String);
    // ...
}
```

- The enum may have fields and methods.

```
enum Element {
    H(1, "Hydrogen"),
    HE(2, "Helium"),
    LI(3, "Lithium");

    public final int num;
    public final String label;
```

```
    private Element(int num, String label) {
        this.num = num;
        this.label = label;
    }
}
```

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
```

Reflection Tutorial

- https://youtu.be/lAoNJ_7LD44
- 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 )
@Target({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)
@Target({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="Shubham", lastName="Patil", company="Sunbeam Karad
")
    private int myField;
    @Developer(firstName="Rahul", lastName="Sansuddi")
    public MyClass() {

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

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

Annotation tutorials

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

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)