Lesson:

APIs and Annotation

List of Concepts Involved:

- Date and Time API in java
- Stream API in Java
- Enums
- What is Annotation?
- In Built Annotation
- Custom Annotation
- Reflection API in Java

Date and Time API in java

Date and Time API: (Joda-Time API)
Until Java 1.7 version the classes present in Java.util package to handle Date and Time (like Date, Calendar, TimeZone etc) are not up to the mark with respect to convenience and performance.

To overcome this problem in the 1.8 version oracle people introduced Joda-Time API.
This API developed by joda.org and available in Java in the form of the "java.time" package.

Ex: program to display System Date and time.

```
import Java.time.*;
public class DateTime {
  public static void main(String[] args) {
    LocalDate date = LocalDate.now();
    System.out.println(date);
    LocalTime time=LocalTime.now();
    System.out.println(time);
  }
}
```

Output
2022-10-30
09:13:34.692

java.util.Date vs java.sql.Date

```
public class Test {
 public static void main(String[] args) {
  java.util.Date utilDate = new java.util.Date();
  System.out.println(utilDate);

  long l = utilDate.getTime();

  java.sql.Date sqlDate = new java.sql.Date(l);
  System.out.println(sqlDate);
 }
}
```

Output
Sun Oct 30 10:05:33 IST 2022
2022-10-30

Difference b/w java.util.Date and java.sql.Date

java.util.Date

- It is a utility class to handle Date in our java program.
- It represents both Date and Time

java.sql.Date

- It is designed class to handle Dates w.r.t DB operations
- It represents only Date,but not Time.

Note: In sql package

Time(C) => Time value
TimeStamp(C) => Date and Time value

Stream API in Java

Streams

To process objects of the collection, in 1.8 version Streams concept introduced.

What is the difference between Java.util.streams and Java.io streams?

java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection but Java.io streams are meant for processing binary and character data with respect to file. i.e it represents a stream of binary data or character data from the file. Hence Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?

- If we want to represent a group of individual objects as a single entity then We should go for collection.
- If we want to process a group of objects from the collection then we should go for streams.
- We can create a stream object to the collection by using the stream() method of the Collection interface.
- stream() method is a default method added to the Collection in 1.8 version.

```
default Stream stream()
Ex: Stream s = c.stream();
import java.util.*;
import java.util.stream.*;

public class Test {
 public static void main(String[] args) {
   ArrayList<Integer> al = new ArrayList<Integer>();
   al.add(0);
   al.add(5);
   al.add(10);
   al.add(15);
   al.add(20);
   al.add(25);

   System.out.println(al);
```

```
        ArrayList<Integer> doubleList = new ArrayList<Integer>();
          for ( Integer i1: al )
                        doubleList.add(i1*2);
          System.out.println(doubleList);

          List<Integer> streamList = al.stream().map(I→I*2).collect(Collectors.toList());
   System.out.println(streamList);
 }
}
```

=> Stream is an interface present in java.util.stream. Once we get the stream, by using that we can process objects of that collection.

We can process the objects in the following 2 phases

1.Configuration

2.Processing

1) Configuration:

We can configure either by using a filter mechanism or by using map mechanism.

Filtering:

We can configure a filter to filter elements from the collection based on some boolean condition by using filter()method of Stream interface.

public Stream filter(Predicate<T> t)

here (Predicate<T > t) can be a boolean valued function/lambda expression

Ex:

Stream s = c.stream();

Stream s1 = s.filter(i -> i%2==0);

Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for

map() method of Stream interface.

public Stream map (Function f);

It can be lambda expression also

Ex:

Stream s = c.stream();

Stream s1 = s.map(i-> i+10);

Once we perform configuration we can process objects by using several methods.

2) Processing

processing by collect() method

Processing by count()method

Processing by sorted()method

Processing by min() and max() methods

forEach() method

toArray() method

Stream.of()method

1. collect()

This method collects the elements from the stream and adds the specified to the collection indicated

(specified) by argument.

2. count()
This method returns the number of elements present in the stream.
public long count()

3. Processing by sorted()method
If we sort the elements present inside the stream then we should go for the sorted() method.
The sorting can either default to natural sorting order or customized sorting order specified by comparator.
sorted()- default natural sorting order
sorted(Comparator c)-customized sorting order.

4. Processing by min() and max() methods
min(Comparator c): returns minimum value according to specified comparator.
max(Comparator c): returns maximum value according to the specified comparator.

5. forEach() method
This method will not return anything.
This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

6. toArray() method
We can use toArray() method to copy elements present in the stream into specified array

7. Stream.of()method
We can also apply a stream for a group of values and for arrays.
Ex:

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```

Enums

We can use enum to define a group of named constants.
Example 1:

```
enum Month
{
 JAN,FEB,MAR, ... DEC; //; ⟶optional
}
Example 2:
enum Color
{
 RED,BLUE,GREEN;
}
```

- Enum concept introduced in 1.5 versions.
- When compared with old languages, enum java's enum is more powerful.
- By using enum we can define our own data types which also come with enumerated data types.

Internal implementation of enum:

```
enum Color
 {
  RED,BLUE,GREEN;
 }
 public final class Color extends java.lang.Enum
 {
  public static final Color RED=new Color();
  public static final Color BLUE=new Color();
 }
```

- Internally enum's are implemented by using class concepts.
- Every enum constant is a reference variable to that enum type object.
- Every enum constant is implicitly a public static final always.

Declaration and usage of enum:

Example 4:

```
enum Color
{
 RED,BLUE,GREEN;//here semicolon is optional.
}

class Test
{
 public static void main(String args[]){
  Color c=Color.RED;
  System.out.println(c);
 }
}
```

Output:

D:\Enum>java Test
RED

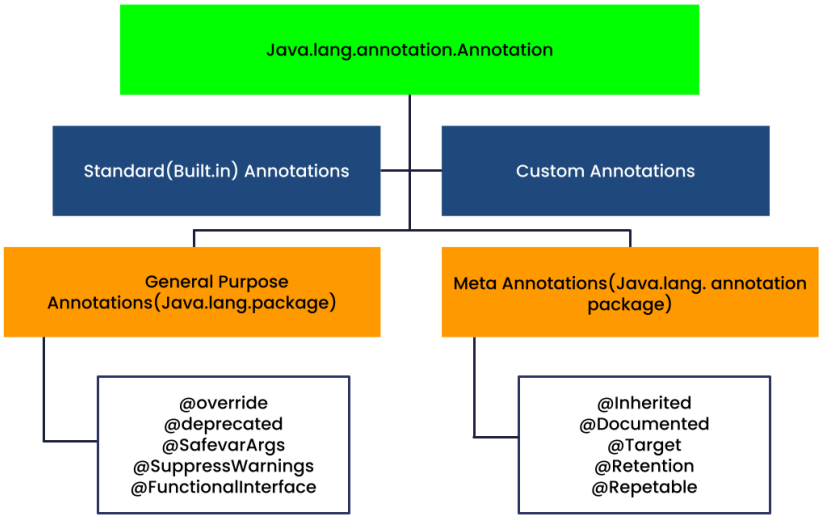
Note:

- Every enum constant is internally static hence we can access it by using "enum name".
- Internally inside every enum toString() method is implemented to return name of the constant

What is Annotation?

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

- **In Built Annotation**



Java provides several built-in annotations that can be used to provide additional information about code elements. Here are some of the most commonly used built-in annotations in Java:
@Override: This annotation is used to indicate that a method in a subclass is intended to override a method in the superclass.
@Deprecated: This annotation is used to mark a program element (such as a method or class) as deprecated, indicating that it is no longer recommended for use and may be removed in future versions of the code.
@SuppressWarnings: This annotation is used to suppress specific warnings generated by the Java compiler, such as unchecked cast warnings or deprecation warnings.
@FunctionalInterface: This annotation is used to indicate that an interface is intended to be a functional interface, which means it has a single abstract method and can be used with lambda expressions.
@Retention: This annotation is used to specify the retention policy for an annotation, indicating whether it should be retained at runtime or only used for compilation.
@Documented: This annotation is used to indicate that an annotation should be included in JavaDoc documentation.
@Target: This annotation is used to specify the types of program elements to which an annotation can be applied, such as classes, methods, or fields.
@Inherited: This annotation is used to indicate that an annotation should be inherited by subclasses of an annotated class or interface.

- **Custom Annotation:**

Custom Annotations
Java Custom annotations or Java User-defined annotations are easy to create and use. The @interface element is used to declare an annotation. For example:

```
@interface MyAnnotation{}
```

Here, MyAnnotation is the custom annotation name.

Types of Annotation
There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

- **Reflection API in Java**

Java reflection is an API used to inspect and modify Java classes, fields, methods, and constructors at runtime. Java has a class named class that collects and stores all information about classes and objects, this class helps facilitate the ability to use reflection.

- **.getClass():** The getClass method is called on the object of a class to retrieve and reflect the class you want.
- **.class Extension:** The .class extension is also used to retrieve and reflect class information. This extension is chained to the end of a class, creating an object of the Class.

Ex:

```
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Car
      Car c1 = new Car();

      // create an object of Class
      // using getClass()
      Class obj = c1.getClass();

      // get name of the class
      String name = obj.getName();
      System.out.println("Name: " + name);

      // get the access modifier of the class
      int modifier = obj.getModifiers();

      // convert the access modifier to string
      String mod = Modifier.toString(modifier);
      System.out.println("Modifier: " + mod);

      // get the superclass of Car
      Class superClass = obj.getSuperclass();
      System.out.println("Superclass: " + superClass.getName());
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```