Miscellaneous Topics

Interfaces New Features

- Interface can have static implemented methods (From Java 8 onwards)
- Interface can have default implemented methods (From Java 8 onwards)
- Interface can have private methods (from Java 9 onwards)
- Interface can have private static methods (from Java 9 onwards)

Functional Interface: Interface with only one abstract method.

Marker Interface: A marker interface is an interface that has **no methods or constants inside it.** An interface is called a marker interface when it is provided as a handle by Java interpreter to mark a class so that it can provide special behaviour to it at runtime and they do not have any method declarations. It is also called **tagging interface. Ex. Serializable, Cloneable, Remote etc.**

Interfaces New Features

```
interface MyInterface  {
        void fun();
        static void fun1()
        {
                System.out.println("fun1 of MyInterface");
        }

        default void fun2()
        {
                System.out.println("fun2 of MyInterface");
        }

        //private void fun3(){}
        //static private void fun4(){}
}
```

Interfaces New Features

```
class InterfaceNewFeatures implements MyInterface
{
	// We can override default method of interface here. Don't mention default here while overriding.
	// public void fun2(){}
	public void call()
	{
		MyInterface.fun1();
		//fun1();		//Error
		hello();
		fun2();
	}
}
```

Java Beans

JavaBeans are classes that encapsulate many objects into a single object (the bean). It is a java class that should follow following conventions:

- Must implement Serializable.
- It should have a public no-arg constructor.
- All properties in java bean must be private with public getters and setter methods.

Java Beans

```
class EmployeeBean implements Serializable
{
	private int empId;
	private String name;

	public EmployeeBean()
	{
	}

	public int getEmpId()
	{
		return this.empId;
	}
```

Java Beans

```
	public void setEmpId(int empId)
	{
		this.empId = empId;
	}

	public String getName()
	{
		return name;
	}

	public void setName(String name)
	{
		this.name = name;
	}
}
```

Inner class

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.
1) Nested Inner class
2) Method Local inner classes
3) Anonymous inner classes
4) Static nested classes

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.
Like class, interface can also be nested and can have access specifiers.
Inner class can be declared within a method called **Method Local inner class**.
Static nested classes are not technically an inner class. They are like a static member of outer class.
Anonymous inner class a class without any name.

Inner class

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance (within the inner class only).
To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass()

static import

- To import static members of a particular class so that it can be used without specifying class name.
- Ex.

```
/* Imports only static members of a class */
import static java.lang.System.*;

class StaticImportDemo
{
	public static void main(String args[])
	{
		out.println("hello");
	}
}
```

Object Cloning (Shallow copy and Deep copy)

- The object cloning is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.
- The java.lang.**Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.
- Syntax of clone method: **protected Object clone() throws CloneNotSupportedException**
- Every class that implements clone() should call **super.clone()** to obtain the cloned object reference.
- A **shallow copy** copies all fields, and does not make copies of dynamically allocated memory pointed to by the fields.
- A **deep copy** copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Objects eligible for garbage collection

- In java, the memory allocated at runtime i.e. heap area can be made free by the process of garbage collection. It is nothing but just a method of making the memory free which is not being used by the programmer. **Only the objects who have no longer reference to them are eligible for garbage collection in java**.
-
- **Garbage Collection Methods**:
  System.gc()
  Runtime.getRuntime.gc()

- Calling one of the gc() methods does not force garbage collection to happen, either; it **only suggests** to the JVM that now might be a good time for some garbage collection

- **Difference** between System.gc() and Runtime.getRuntime.gc() . System.gc()internally calls Runtime.getRuntime.gc() . The only difference is System.gc() is a class method whereas Runtime.getRuntime.gc() is an instance method.

Objects eligible for garbage collection

```
class Demo
{
        String objName;

        Demo(String objName)
        {
                this.objName = objName;
        }

        protected void finalize()
        {
                System.out.println("garbage collector called for "+objName);
        }
}
```

Objects eligible for garbage collection

```
class GarbageCollectorDemo
{

        //Object created inside a method.
        public static void call()
        {
                Demo d1 = new Demo("obj1");//Local reference variable goes out of scope so no
reference of object. Garbage collector called.
        }
```

Objects eligible for garbage collection

```
public static void main(String args[])
        {
                call();
                Demo d2 = new Demo("obj2");
                Demo d3 = new Demo("obj3");
                Demo d4 = new Demo("obj4");
                d2 = null;   //Nullifying the reference variable, garbage collector called for obj2
                d3 = d4;     //Reassigning the reference variable, garbage collector called for obj3
                new Demo("obj5");       //Anonymous object created, garbage collector called for obj5
                //System.gc();
                Runtime.getRuntime().gc();
        }
}
```

Varargs

```
class VarArgsDemo
{

	public static void main(String args[])
	{
		First f = new First();
		f.fun("Hello",1,2,3);
	}
}
```

Varargs

```
class VarArgsDemo
{

	public static void main(String… args)
	{
		First f = new First();
		f.fun("Hello",1,2,3);
	}
}
```

Enumerations

- An enumeration is a list of named constants. These constants are called self-typed.
- Each constant inside enumeration is a public, static member.
- In Java, an enumeration defines a class type.
- In Java, an enumeration can have constructors, methods, and instance variables.
- Once a enumeration is defined, we can create a variable of that type.
- However, even though enumerations define a class type, we do not instantiate an enum using new.
- Instead, we declare and use an enumeration variable in much the same way as we do one of the primitive types.

Enumerations

- It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when we define a constructor for an enum, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

- In the case statement of switch-case enumeration constants are used without being qualified by their enumeration type name.

Enumerations

Methods from Enum class:

int ordinal(); //index position of a constant within enumeration

boolean equals(); // returns true only if both the enumeration variables are type of the same enumeration and refering same constant.

int compareTo(); // compares two enumeration c

Enumerations

```
enum TShirtColors
{
	Red(2), Blue(3), Green(3), Yellow(5), Pink(5);
	int rating = 0;

	TShirtColors(int r)
	{
		rating = r;
	}

	public void printRating()
	{
		System.out.println("Rating : " +rating);
	}
```

Enumerations

```
        public void setRating(int r)
        {
                rating = r;
        }
}

class EnumDemo
{
        public static void main(String args[])
        {
                TShirtColors t1 = TShirtColors.Red;
                TShirtColors t2 = TShirtColors.Blue;
```

Enumerations

```
t1.setRating(5);
        t1.printRating();
        t2.printRating();

        System.out.println(TShirtColors.Red);

        //TShirtColors ch = TShirtColors.Red;
        TShirtColors ch = t2;
        switch (ch)
        {
                case Red:
                        System.out.println("Red purchased");
                        break;
```

Enumerations

```
        case Blue:
                        System.out.println("Blue purchased");
                        break;
                default:
                        System.out.println("No shirt purchased");
            }

        }
}
```

Date and Calendar Class

Calendar class is useful to know the system date and time.
Calendar c = Calendar.getInstance();
Calendar class methods:
Int get (int field): This method returns the value of the given Calendar field.
Void set (int field, int value): This method sets the given field in Calendar object to the given value.

Field values: Calendar.DATE, Calendar.MONTH, Calendar.YEAR, Calendar.HOUR, Calendar.MINUTE, Calendar.SECOND

Date and Calendar Class

Date class is useful to display the date and time at a particular moment.
DateFormat and **SimpleDateFormat** are the classes used to format the date.
DateFormat fmt = DateFormat.getDateInstance(formatconst,region)
DateFormat fmt = DateFormat.getTimeInstance(formatconst,region)
DateFormat fmt = DateFormat.getDateTimeInstance(dateformatconst,timeformatconst,region)

formatconst values: DateFormat.FULL, DateFormat.LONG, DateFormat.MEDIUM, DateFormat.SHORT.
region values represents values for countries in the world: Locale.UK, Locale.KOREA, Locale.FRANCE etc.

SimpleDateFormat sdf = new SimpleDateFormat();
SimpleDateFormat sdf = new SimpleDateFormat(String pattern);
Example : SimpleDateFormat sdf = new SimpleDateFormat (“dd/MM/yyyy”);
SimpleDateFormat used **format( )** method to format the date.

Date and Calendar Class

```
import java.util.*;
import java.text.*;
class SimpleDateFormatDemo
{
	public static void main(String args[]) throws ParseException
	{
		Date d = new Date();
		System.out.println(d);

		DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM,Locale.US);
		String s = df.format(d);
		System.out.println("Formatted Date (Locale.UK): "+s);

		df = DateFormat.getTimeInstance(DateFormat.MEDIUM,Locale.UK);
		s = df.format(d);
		System.out.println("Formatted Time: "+s);
```

Date and Calendar Class

```
            df =
DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM,Locale.UK);
            s = df.format(d);
            System.out.println("Formatted Date Time: "+s);

            SimpleDateFormat sdf = new SimpleDateFormat();
            sdf.applyPattern("dd/MMM/yyyy hh:mm");
            s = sdf.format(d);
            System.out.println("SimpleDateFormat date: "+s);

            //String to Date conversion
            String strDate="24/11/2020";
            Date date1=new SimpleDateFormat("dd/MM/yyyy").parse(strDate);
            System.out.println(strDate+"\t"+date1);
      }
}
```

Date and Calendar Class

Date and Time component

y : Year

d : day of month

M : month in year

H : Hour in day (0-24)

h : Hour in am/pm (1-12)

m : minute in hour

s : second in minute

S : milliseconds in second

a : am/pm marker

Reflection

Reflection is an API which is used **to examine or modify** the behavior of methods, classes, interfaces at runtime.
java.lang.reflect is the package where the required classes for the reflection has been provided.

Important classes
Class
Field
Method
Constructor

Reflection

Some important methods:

1.

Class classObj = obj.getClass()
Getting Class object for the respective class.

2.

Method[] methods = classObj.methods()
Getting array of Method objects for public methods.

Field[] fields = classObj.fields()
Getting array of Field objects for public fields.

Reflection

Some important methods:

3.
Constructor constructor = classObj.getConstructor()
Getting constructor

4.
Method method1 = classObj.getDeclaredMethod("myFun",int.class,double.class)
Getting Method object for a particular method of a class

5.
Field field1 = classObj.getDeclaredField("empName")
Getting Field object for a particular field

Reflection

Some important methods:

6.
method1.**setAccessible**(true) - Making method accessible irrespective of the associated access specifier.
field1.setAccessible(true) - Making field accessible irrespective of the associated access specifier.

7.
method1.**invoke**(obj,5,10.5) - Invoking the method with required parameter values

Note: static methods can also be invoked either by using classObj or obj.

8.
field1.**set**(obj,"Shyam")- Setting the field value

Reflection

Demo program

```
import java.lang.reflect.*;

class MyClass
{
	public int a;
	private int b;
	static int c;

	public MyClass()
	{
		System.out.println("No argument constructor invoked");
	}
```

Reflection

Demo program contd..

```
public MyClass(int x)
        {
                System.out.println("One argument constructor invoked");
        }

        public MyClass(int x, int y)
        {
                System.out.println("Two arguments constructor invoked");
        }

        public void myFun(int a, double b)
        {
                System.out.println("fun1 invoked");
        }
```

Reflection

Demo program contd..

```
public void myFun2()
	{
		System.out.println("myfun2 invoked");
	}

	private void myFun3()
	{
		System.out.println("myfun3 invoked");
	}

	private static void myFun4()
	{
		System.out.println("myfun4 invoked");
	}
}
```

Reflection

Demo program contd..

```
class ReflectionDemo
{
	public static void main(String args[]) throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException
	{
		MyClass myObj = new MyClass(5,10);
		Class classObj = myObj.getClass();

		Method[] methods = classObj.getMethods();
		Field[] fields = classObj.getFields();

		for(Method m : methods)
		{
			System.out.println(m.getName());
		}
```

Reflection

Demo program contd..

```
for(Field f : fields)
        {
                System.out.println(f.getName());
        }

        Constructor constructor = classObj.getConstructor(int.class);
        System.out.println(constructor.getName());

        Method method1 = classObj.getDeclaredMethod("myFun3");
        method1.setAccessible(true);
        method1.invoke(myObj);

        Method method2 = classObj.getDeclaredMethod("myFun4");
        method2.setAccessible(true);
        method2.invoke(classObj);
    } }
```

Reflection

OUTPUT:

Two arguments constructor invoked
myFun
myFun2
wait
wait
wait
equals
toString
hashCode
getClass
notify
notifyAll
a
MyClass
myfun3 invoked
myfun4 invoked

Backed set Collections

In Java, there are these array-backed lists that are generated when you convert arrays to lists using Arrays.asList(...). The list and the array objects point to the same data stored in the heap. Changes to the existing contents through either the list or array result to changing the same data.

Arrays.asList() returns a list backed by the array, because it doesn't make a copy, but Collection.toArray() makes a copy, so it is not backed by the collection.

```
import java.util.*;

public class BackedSetDemo {

    public static void main(String[] args) {
        String[] names = { "Ram", "Geeta", "Mohan" };
       //Array to list
       List<String> listNames = Arrays.asList(names);
```

Backed set Collections

```
        listNames.set(0, "Ramesh");

        System.out.println("List contents:" + listNames.toString());
        System.out.println("Array contents: " + Arrays.toString(names));
    }
}
```

Output:

List contents:[Ramesh, Geeta, Mohan]
Array contents: [Ramesh, Geeta, Mohan]

Backed set Collections

```
import java.util.*;

public class NotBackedSetDemo {

    public static void main(String[] args) {
        List<String> listNames = new ArrayList<String> ();
        listNames.add("Ram");
        listNames.add("Geeta");
        listNames.add("Mohan");

        Object[] names = new Object[listNames.size()];

         //List to array
        names = listNames.toArray();
        listNames.set(0, "Ramesh");
```

Backed set Collections

```
        System.out.println("List contents:" + listNames.toString());
        System.out.println("Array contents: " + Arrays.toString(names));
    }
}
```

Output:

List contents:[Ramesh, Geeta, Mohan]

Array contents: [Ram, Geeta, Mohan]

Regular Expressions

A regular expression (regex or regexp for short) is a special text string for describing a search pattern.

. - Any character except newline
\d - Digit (0-9)
\D - Not a Digit (0-9)
\w - Word character(a-z,A-Z,0-9,_)
\W - Not a word character
\s - Whitespace (space, tab, newline)
\S - Not a whitespace

\b - Word boundary
\B - Not a word boundary
^ - Begining of a string
$ - End of string

Regular Expressions

[ ] - Matches characters in bracket
[^] - Matches characters not in bracket
| - Either Or
() - Group

Quantifiers:
* - O or more times
+ - 1 or more times
? - O or one time
{n} - Exact number (n number of times) e.g. {3}
{m,n} - m to n number of times e.g. {3,5}
{n,} - n or more than n number of times e.g. {3,}

Regular Expressions

[a-zA-Z0-9] - any character from a to z, A to Z or 0 to 9
[0-9] - Any number between 0 to 9.
[a-z] - Any character between a to z.
[A-Z] - Any character between A to Z.
[abc] - a, b or c
[^abc] - any character except a, b or c

Metacharacters (Need to be escaped):

.[{()\^$|?*+

Regular Expressions

```
import java.util.Scanner;
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String args[])
	{
		String regex1 = "(\\+91|0)[1-9][0-9]{9}";	//mobile no. starting with 0 or +91
		//String regex2 = "[a-zA-Z0-9.-_]+@(gmail|cdac)\\.(com|in)";	//valid email id mohit.er_29@gmail.com
		String regex2 = "[a-zA-Z0-9.-_]+@(gmail\\.com|cdac\\.in)";	//valid email id mohit.er_29@gmail.com
```

Regular Expressions

```
String targetStr;

System.out.println("Enter string :");
Scanner sc = new Scanner(System.in);
targetStr = sc.nextLine();

//Pattern pattern = Pattern.compile("(\\+91|0)[1-9][0-9]{9}");
//Matcher matcher = pattern.matcher(targetStr);
//boolean matches = matcher.find();     //Uncomment 3 lines and comment
regex and respective matches. It will also work.

boolean matches = targetStr.matches(regex2);
```

Regular Expressions

```
            if(matches)
                System.out.println("Given string is valid");
            else
                System.out.println("Given string is not valid");
        }
}
```