Lesson 1:Introduction to Java

Q. 1 **Define :**

- **a) Bytecode verifier**
- **b) Class loader**
- **c) JIT compiler**
- **d) JVM**
- **e) JDK**

Ans :

- a) Component of the JVM that ensures that byte code does not access private data
- b) Responsible for keeping classes from different servers separate from each other as well as the local classes
- c) JIT compiler: It is enabled by default and is activated when a java method is called .JIT compiler compiles the byte the code of that method into native machine code compiling it "just in time" to run. When a method has been compiled JVM calls the compile code of that method directly instead of interpreting it.
- d) Java Virtual Machine where the byte code is executed.
- e) Java Development Kit, it is a super set consisting of java compiler, JRE, JVM.

Lesson 3: Language Fundamentals

Q. 2 **What is the default value for float , char ,byte data types ?**
Or
What is the default value of all instance variables

Ans :

int = 0
String = null
float = 0.0f
double = 0.0
boolean=false
Char : a blank space
Any Reference type = null

Q. 3 **a) What happens when you ignore break in switch. Given an Example.**
b) Discuss the valid data types that w.r.t switch case ?

Ans :

a) All the following cases will get executed till it finds the break statement.

Example :

```
public class SwitchExample {
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            int i = 0;
          switch (i) {

          case 0:
            System.out.println("i is 0");
          case 1:
            System.out.println("i is 1");
          case 2:
            System.out.println("i is 2");
          default:
            System.out.println("Free flowing switch example!");

          }
      }
}
```

Output

```
i is 0
```

	`i is 1`
	`i is 2`
	`Free flowing switch example!`
	
	b) Switch statement in java works with the following datatypes:
	- Primitive datatypes: Byte, Short, Char, Int
	- Enumerated datatypes (java enums)
	- string class
	- few classes that wrap primitive types: Character, Byte, Short, Integer

	Lesson 4: Classes and Objects
Q. 4	**What will happen if you don't initialize a local variable and try to print it?**
Ans.	Local variable does not have a default value. Unless a local variable has been assigned a value the compiler will refuse to compile the code that reads it.
Q. 5	**What is the need of :** **a) Final** **b) Finalize** **c) Finally**
Ans.	a) Final: Final is used to apply restrictions on class, method, variable. It is a keyword. b) Finalize: It is used to perform clean up processing just before object is garbage collected. It is a method c) Finally: Finally is used to place important code. It will be executed whether exception is handled or not. It is a block.
Q. 6	**Discuss Static method.**
	Normally we cannot call a method of a class without first creating an instance of that class. By declaring a method using the static keyword, you can call it without first creating an object because it becomes a class method. - If you apply static keyword with any method, it is known as static method. - A static method belongs to the class rather than object of a class. - A static method can be invoked without the need for creating an instance of a class. - static method can access static data member and can change the value of it.
Q. 7	**What is created in heap**
Ans	Object
	Lesson 5: Exploring Basic Java Class Libraries
Q.8	**What are wrapper classes, list all the wrapper classes**
Ans	A wrapper class is a class whose object wraps or contain a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. Wrapper classes: 1.Character 2.Byte 3.Short 4.Integer 5.Float 6.Double

	7.Boolean
Q. 9	**What are the possible modifiers for :** **a) class/static variables** **b) instance variables** **c) local variables**
Ans	Possible modifiers for: a) class/static variables-public, private, static b) instance variables-public, private, default c) local variables-final
Q. 10	**Which type of variables must be initialized-mandatory ?**
Ans	Local variable must be initialized and it is mandatory.
Q. 11	**Which of the following are mutable/immutable ?** **a) string** **b) String buffer** **c) String builder**
Ans.	a) String – immutable & synchronized. The object created as String is stored in the Constant String Pool b) String buffer – mutable & synchronized. The object created through StringBuffer is stored in the heap c) String builder – mutable & unsynchronized. Therefore, Thus StringBuilder is faster than the StringBuffer when calling the same methods of each class.
Q. 12	**Give examples of using :** **a) Append** **b) Concat** **c) Equals** **d) ==** **e) compareTo**

Ans	**a) Append**
	`StringBuffer sb = new StringBuffer("abc");`
	`sb.append("def");`
	`System.out.println("sb = " + sb); // output is "sb = abcdef"`
	
	b) Concat :
	`String str = "Core ";`
	`System.out.println( str=str.concat(" Java") );`
	`Output -> "Core Java”`
	
	c) d) e) Equals, == , compareTo
	`String str = "Hello";`
	`String str1 = new String("Hello");`
	
	`System.out.println(str.equals(str1));         //output : True`
	`System.out.println(str == str1);              //output : False`
	`System.out.println( str.compareTo(str1) );    //output : 0`

Q. 13	**a) Give example of using Scanner object.**

	b) State all the nextxxx() methods used with scanner object

Ans	**a) Example :**
	
	`String myInput = null;`
	`Scanner myscan = new Scanner(System.in).useDelimiter("\\n");`
	`System.out.println("Enter your input: ");`
	`myInput = myscan.next();`
	`System.out.println(myInput);`
	`This will let you use Enter as a delimiter.`
	`input:   Hello world (ENTER)`
	`output : Hello World`
	
	b)
	String next()
	Boolean nextBoolean()
	byte nextByte()
	double nextDouble()
	float nextFloat()
	int nextInt()
	String nextLine()
	Long nextLong()
	Short nextShort()

Q. 14	**Give examples for each :** **a) Get current date** **b) Get tomorrows date - Add one day** **c) Get yesterdays date -Subtract one day**

Ans	a) LocalDate now = LocalDate.now();
	or
	a) SOP(“Today : “ + now);
	b) SOP(“Tomorrow : “ + now.plusDays(1));
	c) SOP(“Yesterday : “ + now.minusDays(1));

Q. 15	**State all Object class methods.**
Ans	• clone() - Creates and returns a copy of this object. • equals() - Indicates whether some other object is "equal to" this one. • finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. • getClass() - Returns the runtime class of an object. • hashCode() - Returns a hash code value for the object. • notify() - Wakes up a single thread that is waiting on this object's monitor. • notifyAll() - Wakes up all threads that are waiting on this object's monitor. • toString() - Returns a string representation of the object. • wait() - Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
Q. 16	**Discuss all Access Modifiers**

Ans :

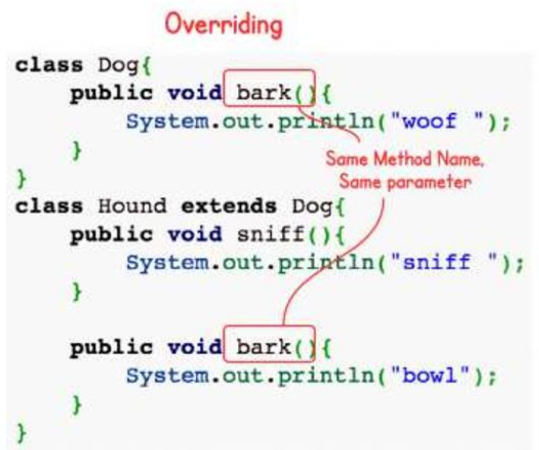
Location/Access Modifier	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Lesson 6: Inheritance and Polymorphism

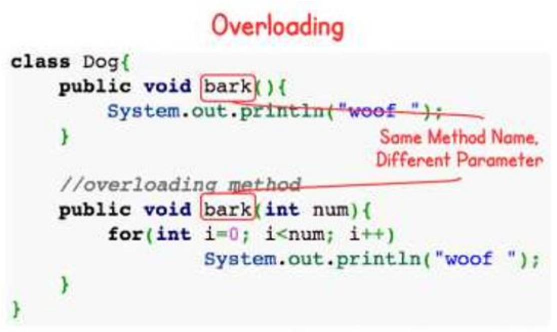
Q. 17 **Difference between overriding and overloading**

Ans

Overloading (compile time polymorphism)	Overriding (run time polymorphism)
Two or more methods within the same class share the same name but parameter declarations are different. You can overload Constructors and Normal Methods.	A method in a subclass has the same name and type signature as a method in its super class, then the subclass method overrides the super class method. Overridden methods allow Java to support run-time polymorphism



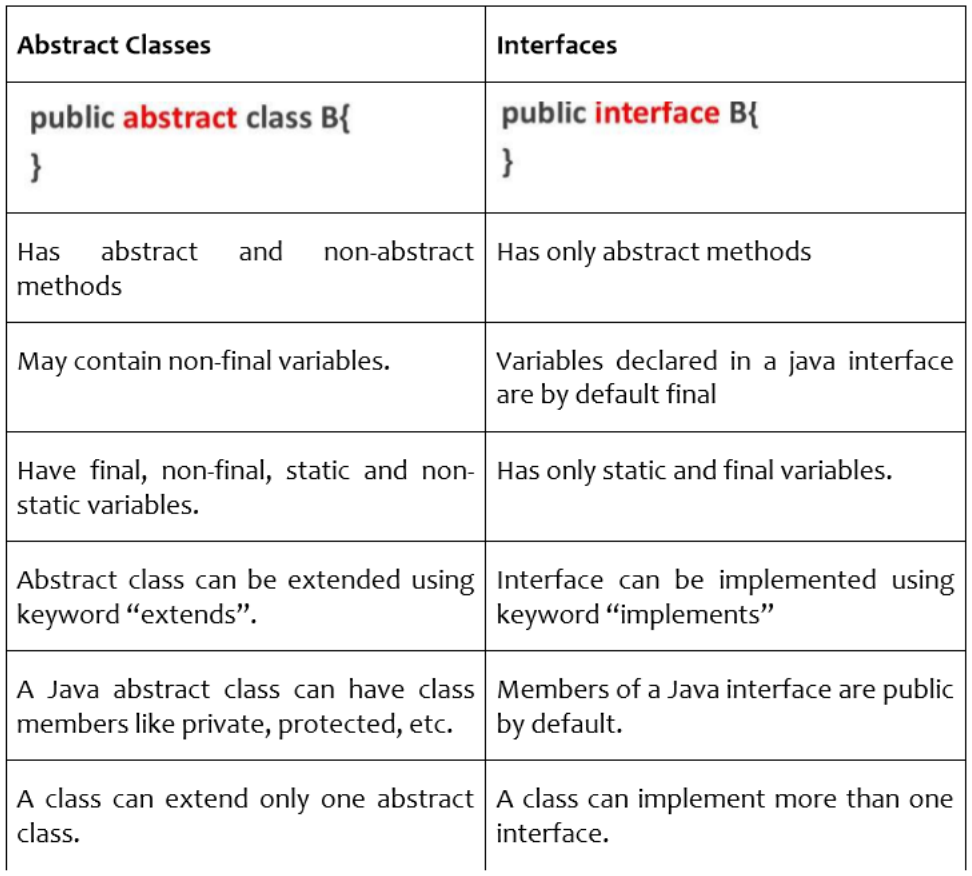
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```



```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Q. 18 **Difference between Abstract Class and Interface**

Ans



Abstract Classes	Interfaces

public **abstract** class B{
 }
	public **interface** B{
	 }

Has abstract and non-abstract methods	Has only abstract methods
May contain non-final variables.	Variables declared in a java interface are by default final
Have final, non-final, static and non-static variables.	Has only static and final variables.
Abstract class can be extended using keyword "extends".	Interface can be implemented using keyword "implements"
A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
A class can extend only one abstract class.	A class can implement more than one interface.

Q. 19 **State the modifiers of the data members in an interface**

Ans Members of an INTERFERENCE are by default PUBLIC.

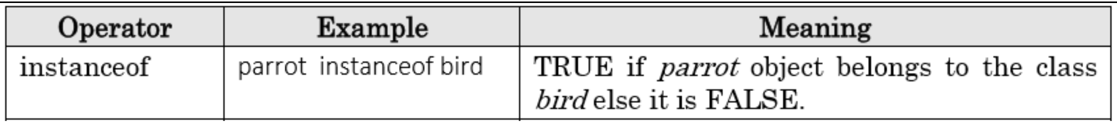
Q. 20 **Aggregation relationship – how will you implement in java**

Ans.

```
class Employee{
int id;
String name;
Address address;//Address is a class
...
 }
```

Q. 21 **InstanceOf – Use and Example**

Ans



Operator	Example	Meaning
instanceof	parrot instanceof bird	TRUE if *parrot* object belongs to the class *bird* else it is FALSE.

Example :

```
   class Simple1{
    public static void main(String args[]){
```

```
    Simple1 s=new Simple1();
    System.out.println(s instanceof Simple1);
    }
   }
```

Output : true

Q. 22 **Discuss all points about key word “this” and “super” with examples (while writing constructors)**

Ans

<u>THIS</u>

The ‘this’ keyword is used to refer the current object. As shown in the above example, the constructor parameters are shadowing the instance variables. Therefore we can use this keyword to make difference between the local variables/parameters and instance variables.

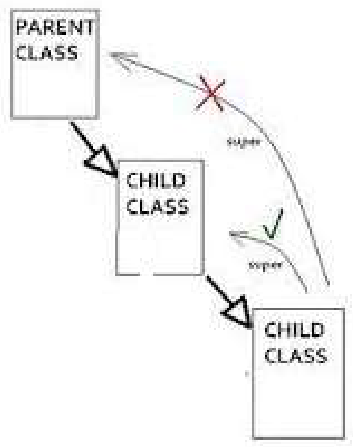
```
class Point {
        int xCord;
        int yCord;
        Point() {
            this(0, 0);      //chaining constructors using this
        }
        Point(int xCord, int yCord) {
            this.xCord = xCord;
            this.yCord = yCord;
        }
}
```

<u>SUPER</u>

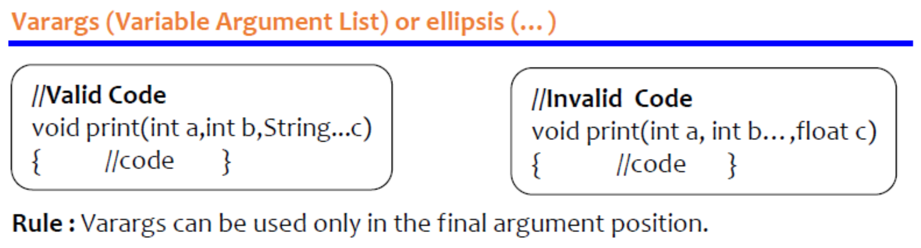
It is a reference variable used to refer the immediate parent class object
- super() invokes immediate parent class constructor
- Call member (variables & methods) of parent class

Syntax : super.baseclassMemberName

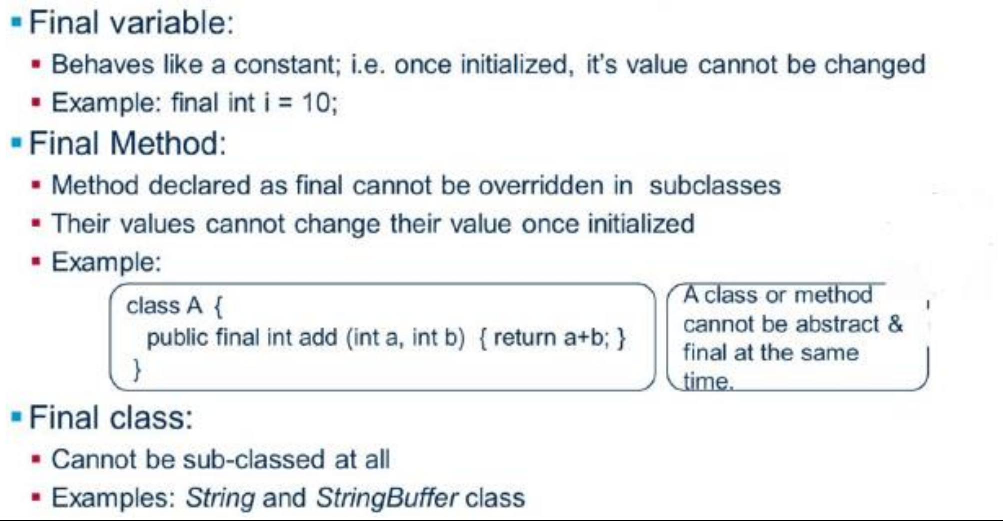
```
class Base {
  public void baseMethod() {
      System.out.println("Base");
  }
}
class Derived extends Base {
   public void derivedMethod() {
      super. baseMethod ();
      System.out.println("Derived");
   }
}
class Test {
   public static void main(String args[]){
               Derived derived=new Derived();
               derived. derivedMethod();
   }
}
```



Q. 23	**How will you write varargs (what conditions must be followed).**	

Ans	 Varargs (Variable Argument List) or ellipsis (…)
	//Valid Code
	`void print(int a,int b,String...c)`
	`{ //code }`
	//Invalid Code
	`void print(int a, int b…,float c)`
	`{ //code }`
	Rule : Varargs can be used only in the final argument position.	

Q. 24	Explain all points about : a) Final variable b) Final method c) Final class	

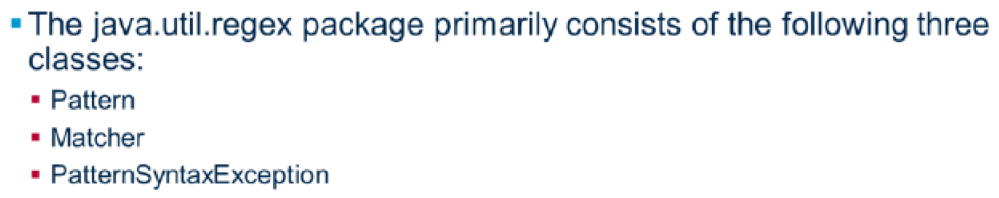
Ans	 ▪ Final variable:
	▪ Behaves like a constant; i.e. once initialized, it's value cannot be changed
	▪ Example: final int i = 10;
	▪ Final Method:
	▪ Method declared as final cannot be overridden in subclasses
	▪ Their values cannot change their value once initialized
	▪ Example:
	`class A {`
	`public final int add (int a, int b) { return a+b; }`
	`}`
	A class or method cannot be abstract & final at the same time.
	▪ Final class:
	▪ Cannot be sub-classed at all
	▪ Examples: *String* and *StringBuffer* class	

	Lesson 7: Abstract Classes and Interfaces	
Q. 25	**Difference between final and abstract class**	
Ans:	<u>Final Classes</u> • final classes are the way we can prevent class being extended • we can instantiate final class and immutable objects can be created • these cannot contain abstract methods. Must have all the method implementations in it Eg: String class	A<u>bstract Classes</u> • Abstract classes are always extended, for situation like no abstract methods, these still can have member elements that makes body and which can be inherited. • Can not be instantiated. So, cannot create immutable objects Eg: HTTP Servlet class • Can contain abstract methods and concrete methods
Q. 26	**By default interface data members are ____________**	
Ans	FINAL	

Lesson 8: Regular Expressions

Q. 27 Three classes in regex package

Ans.

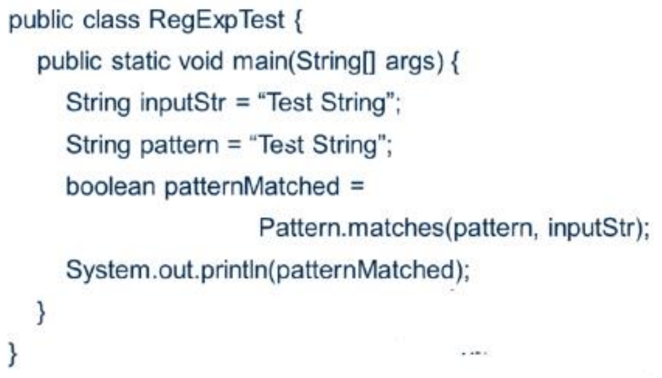


- The java.util.regex package primarily consists of the following three classes:
  - Pattern
  - Matcher
  - PatternSyntaxException

Q. 28 **Examples on pattern matching**

Ans.

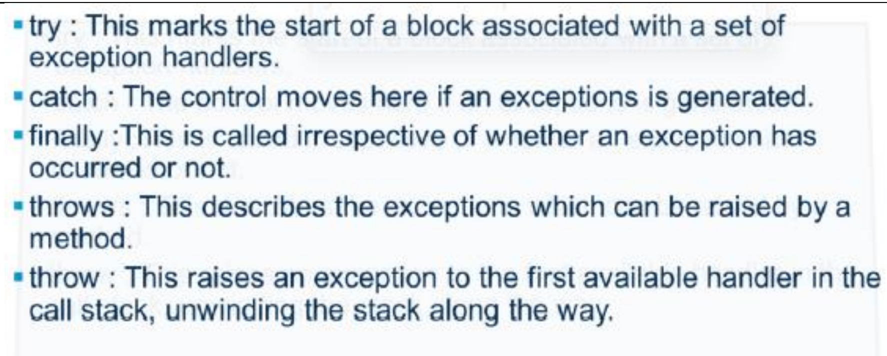
Example 1 :



```
public class RegExpTest {
   public static void main(String[] args) {
      String inputStr = "Test String";
      String pattern = "Test String";
      boolean patternMatched =
                         Pattern.matches(pattern, inputStr);
      System.out.println(patternMatched);
   }
}
```

Example 2 :

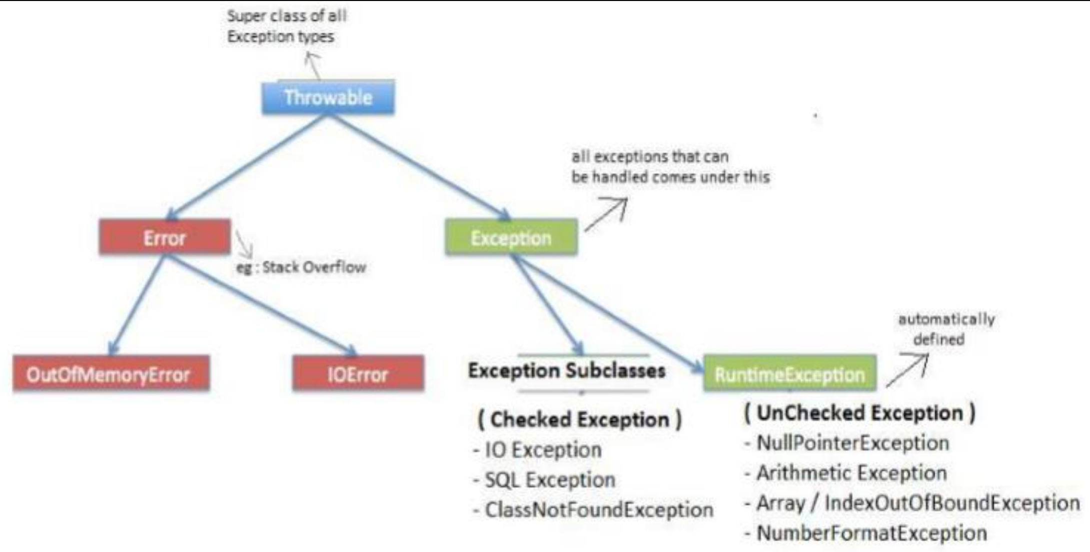
```
import java.util.regex.*;

public class RegExpTest {
		public static void main(String[] args)
		{
				String inputStr = "Test String";
				String pattern = "Test String";
				boolean patternMatched = Pattern.matches(pattern, inputStr);
				System.out.println(patternMatched);

				/*
				 * Pattern pattern1 = Pattern.compile(","); String[] str =
				 * pattern1.split("Shop,Mop,Hopping,Chopping"); for (String st : str) {
				 * System.out.println(st); }
				 */
				String input = "Shop,Mop,Hopping,Chopping";
				Pattern pattern1 = Pattern.compile("hop");
				Matcher matcher = pattern1.matcher(input);
				System.out.println(matcher.matches());
				while (matcher.find())
				{
						System.out.println(matcher.group() + ": " + matcher.start() + ": "
										+ matcher.end());
				}
		}
}
```

	Lesson 9:Exception Handling
Q. 29	**List ALL the Checked exception and UnChecked exception**
Ans.	- Checked Exceptions : SQLException, IOException, ClassNotFoundException - UnChecked Exceptions : NullPointerException, ArithmeticException, ArrayIndexOutOfBoundException, NumberFormatException
Q. 30	**Base class of all exception**
Ans.	Throwable
Q. 31	**Define :** **a) Try** **b) Catch** **c) Finally** **d) Throw** **e) Throws**
Ans.	 ▪ try : This marks the start of a block associated with a set of exception handlers. ▪ catch : The control moves here if an exceptions is generated. ▪ finally :This is called irrespective of whether an exception has occurred or not. ▪ throws : This describes the exceptions which can be raised by a method. ▪ throw : This raises an exception to the first available handler in the call stack, unwinding the stack along the way.
Q. 32	**Significance of Try-with-resource feature in exception handling**
Ans.	A try-with-resources is a new feature added in java 7, where resources are closed automatically. Any block after try (either catch or finally block) will be executed only after the resource is closed
Q. 33	**Any null reference with method invocation will create NullPointer exception – Give example**

Ans.	class TryCatchDemo{
	public static void main(String a[]) {
	String str= null;
	try {
	str.equals("Hello"); **//NullPointerException**
	} catch(NullPointerExceptionne) {
	str= new String("Hello");
	System.out.println(str.equals("Hello"));
	}
	System.out.println("Continuing in the program");
	}
	}

Q. 34	**Layered architecture of Exception handling**

Ans.



Lesson 10:Array

Q. 35 **Syntax for declaring and initializing arrays. Give example**

Ans :

Declaration	int arr1[];
	
	arr1 = new int[4];

Short form of above Declaration	int arr1[] = new int[4]; (or) int[] arr1 = new int[4];
Declaration with Initialization	int arr1[] = {2,3,4,5};
Initialization in case of Declaration 1,2	arr1[0]=2; arr1[1]=3; arr1[2]=4; arr1[3]=5;
Declaring Array Object	Student stu = new Student[3]; // creates 3 objects of Student class

2D Array Declaration	int arr1[][] ={ {2,3},{4,5},{34,56}};
	
	int arr1[][] = new int[3][2];

Lesson 11: Collection

Q. 36 **Difference between enhanced for loop and iterator with example.**

Ans. **Iterator** is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection. While **enhanced For** loop is meant for traversing items in a collection.

Example with iterator :

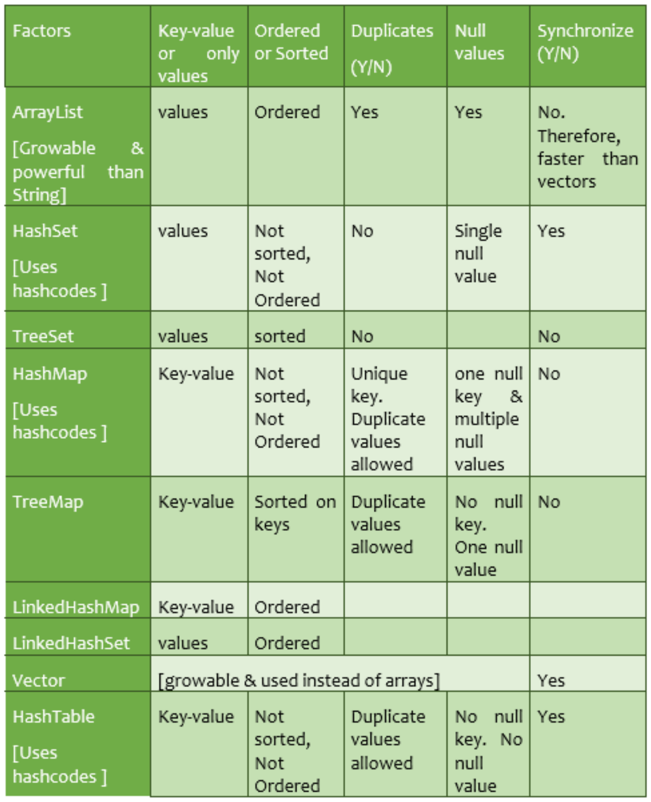
```
        void printAll(Collection<Emp> employees) {
     for (Iterator<Emp> iterator = employees.iterator();
iterator.hasNext(); )
          System.out.println(iterator.next()); } }
```

Example of Enhanced for loop :

```
void printAll(Collection<Emp> employees) {
      for (Emp empObj : employees) )
          System.out.println( empObj ); }}
```

Q. 37 All collections' comparison for ordered, sorted, duplicates, synchronization, key/value pair & allows null.

Ans.



Factors	Key-value or only values	Ordered or Sorted	Duplicates (Y/N)	Null values	Synchronize (Y/N)
ArrayList [Growable & powerful than String]	values	Ordered	Yes	Yes	No. Therefore, faster than vectors
HashSet [Uses hashcodes]	values	Not sorted, Not Ordered	No	Single null value	Yes
TreeSet	values	sorted	No		No
HashMap [Uses hashcodes]	Key-value	Not sorted, Not Ordered	Unique key. Duplicate values allowed	one null key & multiple null values	No
TreeMap	Key-value	Sorted on keys	Duplicate values allowed	No null key. One null value	No
LinkedHashMap	Key-value	Ordered			
LinkedHashSet	values	Ordered			
Vector	[growable & used instead of arrays]				Yes
HashTable [Uses hashcodes]	Key-value	Not sorted, Not Ordered	Duplicate values allowed	No null key. No null value	Yes

Q. 38 Give an Example on :

a) Arrays.sort(array)
b) Collections.sort()
c) remove()
d) removeAll()
e) isEmpty()

Ans.

a) Arrays.sort():-

```
int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};
Arrays.sort(arr);
```

b) c) d) e)

```
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " + al.size());

// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
```

	`al.add(1, "A2");`
	`System.out.println("Size of al after deletions: " + al.size());   //7`
	
	`al.remove("F");`
	`System.out.println("Size of al after deletions: " + al.size());  //6`
	`al.removeAll();   //remove all elements`
	`System.out.println("al.isEmpty());         //returns TRUE`

	Lesson 12: File IO
Q. 39	**Different types of streams in File IO, Buffered Streams.**
Ans.	• Byte Streams: Handle I/O of raw binary data. • Character Streams: Handle I/O of character data. • Buffered Streams: Optimize input and output with reduced number of calls to the native API. • Data Streams: Handle binary I/O of primitive data type and String values. • Object Streams: Handle binary I/O of objects. • Scanning & Formatting: Allows program to read and write formatted text.
Q. 40	**Need of flush() & isFile() method with Examples.**
Ans.	The **java.io.Writer.flush()** method flushes the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. Then, if that destination is another character or byte stream, flush it. Thus one flush() invocation will flush all the buffers in a chain of Writers and OutputStreams. The **java.io.File.isFile()** checks whether the file denoted by this abstract pathname is a normal file.
Q. 41	**Difference between Serialization and Deserialization**
Ans.	**Serialization:** Serialization is the process through which we can store the state of an object into any storage medium. We can store the state of the object into a file, into a database table etc. An object is serialized by writing it an ObjectOutputStream. **Deserialization:** Deserialization is the opposite process of serialization where we retrieve the object back from the storage medium. An object is deserialized by reading it from an ObjectInputStream.
	Lesson 13: Introduction to Junit 4 & Lesson 14: Advanced Testing
Q. 42	Explain @Test with all attributes like timeout, expected.

Ans.	**Example 1:**
	`@Test(expected = ArithmeticException.class)`
	`public void divideByZeroTest() {`
	`calobj.divide(15,0);`
	`}`
	
	Example 2:
	`@Test(timeout=1000)`

Q. 43	Explain methods of Assert class

Ans.

- Fail([String]) : It signals the failure of a test. This method has two formats. If the String argument is not provided, then no message is displayed. Else the String argument message is displayed.
- assertTrue(boolean) : It asserts if the condition is true. Similarly assertFalse(boolean) asserts if the condition is false.
- assertEquals([String message],expected,actual) : It asserts whether the two objects passed as arguments are equal. This method can accept any kind of values for comparison like double long, etc..
- assertNull([message],object) : It asserts that an object is null.
- assertNotNull([message],object) : It asserts that an object is not null.
- assertSame([String],expected,actual) : It asserts that two objects refer to the same object and assertNotSame([String],expected,actual) asserts that two objects do not refer to the same object.
- assertThat(String, T actual, Matcher <T> matcher) : It asserts that “actual” satisfies the condition specified by the “matcher”.

Q. 44 **Give Example : '@RunWith(Suite.class), @Suite.SuiteClasses**

Ans.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({ TestCalAdd.class, TestCalSubtract.class,
TestCalMultiply.class, TestCalDivide.class })
public class CalSuite {
// the class remains completely empty,
// being used only as a holder for the above annotations
}
```

Q. 45 **Define :**

@Before
@After
@BeforeClass
@AfterClass
@ignore

Ans.

- @Test – used to signify a method is a test method
- @Before – can do initialization task before each test run
- @After – cleanup task after each test is executed
- @BeforeClass – execute task before start of tests
- @AfterClass – execute cleanup task after all tests have completed
- @Ignore – to ignore the test method

Q. 46 **What is parameterized test?**

Ans. Allows you to run the same test with different data.
Syntax : @RunWith(Parameterized.class)