MODULE III QUESTIONS SOLUTION

1. **Explain the constructor method. How it differs from other member function.**

Constructor:

- A constructor initializes the instance variables of an object of class.
- It is called immediately after the object is created but before the new operator completes.
  1. it is syntactically similar to a method
  2. it has the same name as the name of its class
  3. it is written without return type; the default return type of a class constructor is the same class
- When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.
- Constructor can be zero argument/default or parameterized.
- Zero argument/default constructor does not take any parameter.
- Parameterized constructors take the parameters

```
class Box {
    double width;
    double height;
    double depth;
   // Zero argument or default constructor
    Box() {
          System.out.println("Constructing Box using default constructor");
          width = 10; height = 10; depth = 10;
    }

    Box(double w, double h, double d) {
          System.out.println("Constructing Box using parameterized  constructor");
          width = w; height = h; depth = d;
    }
    // Parameterized constructor
    double volume() {
          return width * height * depth;
    }
}
```

Difference between Java constructors and methods

Constructors	**Methods**
Constructors are used to initialize the state of object	M**ethod**s are used to expose the behaviour of object
Constructor must not have return type	A **method** must have a return type
Constructor name is same as the class name	M**ethod** name is usually not same as the class name
Constructor is invoke implicitly at the time of object creation	**Method** is invoked explicitly after objects are created
If constructor is not provided the compiler provides a default constructor	The compiler doesnot provide a default **method**

2. **Write short notes on: i) Final class ii) Abstract class.**

Final class

A class declared as **final** cannot be extended by other classes. Though a final class cannot be extended, it can extend other classes. In simpler words, a final class can be a sub class but not a super class. A class is declared to be final using the final keyword as shown below:

```
// The final keyword can be placed either before or after the access specifier.
final public class A {
    //code
}
```

When we attempt to extend a final class or override a final method, compilation errors occur.

```
class B extends A { // compilation error, A is final
}
```

Abstract class

- A class that is declared with abstract keyword, is known as abstract class in java as shown below:
  `abstract class A{}`
- It cannot be instantiated.
- It can have abstract and non-abstract methods (method with body).

- It needs to be extended and its method implemented.

Example:

In this example, Bike the abstract class that contains only one abstract method run.
It implementation is provided by the Honda class.

```
abstract class Bike{
        abstract void run();
}
class Honda extends Bike{
        void run(){System.out.println("running safely..");}
        public static void main(String args[]){
                Bike obj = new Honda();
                obj.run();
        }
}
```

3. Write short notes on modifiers: i) Static ii) Final iii) Abstract iv) Native

Static modifier

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- Both methods and variables can be declared static.
- The most common example of a **static** member is **main( )**. main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:
- They can only call other static methods.
- They must only access static data.
- They cannot refer to this or super in any way.

The following example shows a class that has a static method, some static variables, and a static initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
        static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        }
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
        }
        public static void main(String args[]) {
                meth(42);
        }
}
```

Final modifier

final keyword can be used along with variables, methods and classes. 1) final variable, 2) final method and 3) final class

1. **final** variable: final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Consider the following example

```
class Demo{
   //It is considered as a good practice to have constant names in UPPER CASE
   final int MAX_VALUE=99;
   void myMethod(){
      MAX_VALUE=101;  // Compilation Error modifying a final variable
   }
   public static void main(String args[]){
      Demo obj=new  Demo();
      obj.myMethod();
   }
}
```

Output:

We get a compilation error in the above program because we tried to change the value of a final variable "MAX_VALUE".

2. **final** method: A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

Example:The following fragment illustrates final:

```
class A {
   final void meth() {
          System.out.println("This is a final method.");
   }
}
class B extends A {
   void meth() { // ERROR! Can't override.
          System.out.println("Illegal!");
    }
}
```

3. final class: We cannot extend a final class. Consider the below example:

Here is an example of a final class:

```
final class A {
       // ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
       // ...
}
```

Abstract modifier: The abstract modifier is placed before classes or methods. It cannot be applied to variables.

Abstract class : Abstract class cannot be directly instantiated, but has to be subclassed.

```
public abstract class Test{
    //class implementation
}
```

Abstract method: Abstract method does not have an implementation in the class, but it has to be implemented in a subclass.

```
public abstract void test();
```

Here is a simple example of an abstract class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
   abstract void callme();
   // concrete methods are still allowed in abstract classes
   void callmetoo() {
          System.out.println("This is a concrete method.");
   }
 }
 class B extends A {
   void callme() {
          System.out.println("B's implementation of callme.");
   }
 }
 class AbstractDemo {
   public static void main(String args[]) {
          B b = new B();
          b.callme();
          b.callmetoo();
   }
}
```

Native modifier: The native keyword is applied to a method to indicate that the method is implemented in native code written in another programming language such as C, C++, FORTRAN or assembly language. The body of a native method is given as a semicolon only, indicating that the implementation is omitted. It works together with JNI (Java Native Interface).

```
// Example declaration of native method
public native int square(int i);
```

4. Explain with an example each, the effect of the keyword "final" with i) a class and ii) method(s) of a class.

1. **final** class: We cannot extend a final class. Consider the below example:

Here is an example of a final class:

```
final class A {
      // ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
      // ...
}
```

2. **final** method: A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

Example:The following fragment illustrates final:

```
class A {
   final void meth() {
         System.out.println("This is a final method.");
   }
}
class B extends A {
   void meth() { // ERROR! Can't override.
         System.out.println("Illegal!");
   }
}
```

5. Write a note on object instantiation/creation.

A class provides the blueprint for objects; Objects are created from the class. Following statements creates an object of class Point:

```
Point origin = new Point(23, 94);
```

This statements has three parts:

1. **Declaration**: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The `new` keyword is a Java operator that creates the object.
3. **Initialization**: The `new` operator is followed by a call to a constructor, which initializes the new object.

Object Instantiation: The new keyword is a Java operator that creates/instantiates the object. The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor. The new operator requires a single argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate. The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
Point originOne = new Point(23, 94);
```

6. What is super? Explain the use of super with suitable examples. OR
With example, give two uses of super.

What is super?

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
super has two general forms.

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Each use is examined here.

1. The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
   Animal(){
          System.out.println("animal is created");}
   }
class Dog extends Animal{
   Dog(){
          super();
          System.out.println("dog is created");
   }
}
class TestSuper3{
   public static void main(String args[]){
          Dog d=new Dog();
   }
}

Output:
animal is created
dog is created
```

2. The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

In the example below, Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local. To call the parent class method, we need to use super keyword.

```
class Animal{
   void eat(){
          System.out.println("eating...");
   }
}
class Dog extends Animal{
   void eat(){
          System.out.println("eating bread...");
   }
   void bark(){
          System.out.println("barking...");
   }
   void work(){
          super.eat();
          bark();  this

   }
}
class TestSuper2{
   public static void main(String args[]){
          Dog d=new Dog();
          d.work();
   }
}
Output:
eating...
barking...
```

7. **Describe the significance of final and super with examples.**

Refer final from question 3 above.
Refer super from question 4 above.

8. **What is meant by instance variable hiding? How to overcome it?**

Instance Variable Hiding: In Java, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.

Because this pointer directly refers to the object, it can be used to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
   this.width = width;
   this.height = height;
   this.depth = depth;
}
```

9. Differentiate C++ language and JAVA language with respect to inheritance, and also mention the use of super and this in JAVA inheritance.

Comparison of C++ and Java with respect to inheritance

Java	**C++**
In Java, all classes inherit from the Object class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and Object class is root of the tree.	In C++, there is forest of classes; when we create a class that doesn't inherit from anything, we create a new tree in forest.
In Java, a class cannot directly access the grandparent's members. In Java, we can access grandparent's members only through the parent class.	In C++, we can use scope resolution operator (::) to access any ancestor's member in inheritance hierarchy.
In Java, **protected members** of a class "A" are accessible in other class "B" of same package, even if B doesn't inherit from A.	In C++, protected members of a class "A" are accessible in class "B" only if B inherits from A.
Java uses *extends* keyword for inheritance. Java doesn't provide an inheritance specifier like public, protected or private.	C++, Java provides an inheritance specifier.
In Java, methods are virtual by default.	In C++, we explicitly use virtual keyword.
Java doesn't support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.	C++ supports multiple inheritance.

The this Keyword: Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class' type is permitted.

Because this pointer directly refers to the object, it can be used to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
   this.width = width;
   this.height = height;
   this.depth = depth;
}
```

Use of super

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. super has two general forms.

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

For details refer question 4.

10. Which is an alternative approach to implement multiple inheritance in Java? Explain, with an example. OR
11. Briefly explain the role of interfaces while implementing multiple inheritance in JAVA.

Multiple inheritance is the ability of a single class to inherit from multiple classes. Java does *not* have this capability. One specific problem that Java avoids by not having multiple inheritance is called the diamond problem.

Interfaces - Java's alternative to multiple inheritance

- In Java a class implements an interface if it provides a complete set of methods defined by this interface.
- What a Java class does have is the ability to implement multiple interfaces – which is considered a reasonable substitute for multiple inheritance, but without the added complexity. one class may implement any number of interfaces Each class is free to determine the details of its implementation. Implementation relation is written with the implements keyword.

- General format of a class that includes the implements clause:
  ```
  access class name
  extends super-class
  implements interface1, interface2, …, interfaceN {
         …
  }
  ```
  Access is public or default.

- If a class implements several interfaces, they are separated with a comma.
- If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.
  This is what a Java class that implements multiple interfaces would look like:

  ```
  // Wolf and Canine are interfaces
  public class Dog implements Wolf, Canine {
         /*...*/
  }
  ```

  You can also have a class that extends one class, while implementing an interface – or even multiple interfaces. So, a class like this is perfectly legal in Java:

  ```
  // Wolf is an interface
  // Dog is a base class

  // Poodle both extends from a class and implements an interface
  public class Poodle extends Dog implements Wolf {
         /*...*/
  }
  ```

12. **Discuss the following: i) inner classes ii) Overriding and overloading**
13. **What is an inner class? Write a program to demonstrate inner class.**

Inner Classes: An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner.

```
// Demonstrate an inner class.
class Outer {
       int outer_x = 100;
       void test() {
              Inner inner = new Inner();
              inner.display();
       }
       // this is an inner class
       class Inner {
              void display() {
                     System.out.println("display: outer_x = " + outer_x);
              }
       }
}
       class InnerClassDemo {
          public static void main(String args[]) {
              Outer outer = new Outer();
              outer.test();
          }
        }
}
```
Output
```
display: outer_x = 100
```

- An instance of Inner can be created only within the scope of class Outer. The Java compiler generates an error message if any code outside of class Outer attempts to instantiate class Inner.
- We can create an instance of Inner outside of Outer by qualifying its name with Outer, as in Outer.Inner.
- An inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class

ii) Discuss Overriding and Overloading

Method Overriding

- When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is overridden.
- When an overridden method is called from within the sub-class:
  1. it will always refer to the sub-class method
  2. super-class method is hidden

Example: Hiding with Overriding

```
class A {
	int i, j;
	A(int a, int b) {
		i = a; j = b;
	}
	void show() {
		System.out.println("i and j: " + i + " " + j);
	}
}

class B extends A {
	int k;
	B(int a, int b, int c) {
		super(a, b);
		k = c;
	}
	void show() {
		System.out.println("k: " + k);
	}
}
```

- **When show() is invoked on an object of type B, the version of show() defined in B is used:**

```
class Override {
	public static void main(String args[]) {
		B subOb = new B(1, 2, 3);
		subOb.show();
	}
}
```

- **The version of show() in A is hidden through overriding.**

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- Method overriding occurs only when the names and types of the two methods (super-class and sub-class methods) are identical. If the method parameters/types are not identical, the two methods are simply overloaded.

```
class A {
	int i, j;
	A(int a, int b) {
		i = a; j = b;
	}
	void show() {
		System.out.println("i and j: " + i + " " + j);
	}
}
```

- The show() method in B takes a String parameter, while the show() method in A takes no parameters:

```
class B extends A {
	int k;
	B(int a, int b, int c) {
		super(a, b); k = c;
	}
	void show(String msg) {
		System.out.println(msg + k);
	}
}
```

- The two invocations of show() are resolved through the number of arguments (zero versus one):

```
class Override {
      public static void main(String args[]) {
            B subOb = new B(1, 2, 3);
            subOb.show("This is k: ");
            subOb.show();
      }
}
```

14. Distinguish between method overloading and overriding in JAVA, with suitable examples.

Method Overloading	**Method Overriding**
Overloading happens at compile-time i.e. the binding of overloaded method call to its definition has happens at compile-time	Overriding happens at runtime i.e. binding of overridden method call to its definition happens at runtime
Method overloading can be done in the same class	For method overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
Static methods can be overloaded which means a class can have more than one static method of same name.	Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
Static binding is being used for overloaded methods and	dynamic binding is being used for overridden methods
private and final methods can be overloaded but they cannot be overridden.	private and final methods cannot be overridden.
Argument list should be different while doing method overloading.	Argument list should be same in method Overriding.

Overloading example

```
//A class for adding upto 5 numbers
class Sum {
    int add(int n1, int n2) {
        return n1+n2;
    }
    int add(int n1, int n2, int n3) {
        return n1+n2+n3;
    }
    public static void main(String args[]) {
       Sum obj = new Sum();
       System.out.println("Sum of two numbers: "+obj.add(20, 21));
       System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));
    }
}
Output:

Sum of two numbers: 41
Sum of three numbers: 63
```

Overriding example

```
// Here speedLimit() method of class Ford is overriding the speedLimit() method of class CarClass.
class CarClass {
    public int speedLimit() {
        return 100;
    }
}
class Ford extends CarClass {
    public int speedLimit() {
        return 150;
    }
    public static void main(String args[]) {
       CarClass obj = new Ford();
       int num= obj.speedLimit();
       System.out.println("Speed Limit is: "+num);
    }
}
```

```
Output:

Speed Limit is: 150
```

15. **Create a class figure in JAVA, with following members: dim1, dim2 and abstract method area. Create subclasses Triangle and Rectangle with implementation of area.**

```
// Abstract class figure with two dimensions and abstract method area
abstract class Figure {
       double dim1;
       double dim2;
       Figure(double a, double b) {
              dim1 = a; dim2 = b;
       }
       abstract double area();
}
// Triangle is a sub-class of Figure:
class Triangle extends Figure {
       Triangle(double a, double b) {
              super(a, b);
       }
       double area() {
              System.out.println("Inside Area for Triangle.");
              return dim1 * dim2 / 2;
       }
}
// Rectangle is a sub-class of Figure:
class Rectangle extends Figure {
       Rectangle(double a, double b) {
              super(a, b);
       }
       double area() {
              System.out.println("Inside Area for Rectangle.");
              return dim1 * dim2;
       }
}

// Invoked through the Figure variable and overridden in their respective subclasses, the area()
method returns the area of the invoking object:
class FindAreas {
       public static void main(String args[]) {
              Triangle t = new Triangle(10, 8);
              Rectangle r = new Rectangle(9, 5);
              Figure figref;
              figref = t; System.out.println(figref.area());
              figref = r; System.out.println(figref.area());
       }
}
```

16. **Explain dynamic method dispatch in Java with example program.**

Dynamic method dispatch: Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.

- Method overriding allows for dynamic method invocation:
- an overridden method is called through the super-class variable
- Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
- when different types of objects are referred, different versions of the overridden method will be called.
-

```
// A is super class
class A {
   void callme() {
          System.out.println("Inside A's callme method");
   }
}
// Two sub-classes B and C verride the A's callme() method.
```

```
class B extends A {
   void callme() {
          System.out.println("Inside B's callme method");
   }
}
class C extends A {
   void callme() {
          System.out.println("Inside C's callme method");
   }
}
```

- **Overridden method is invoked through the variable of the super-class type. Each time, the version of the callme() method executed depends on the type of the object being referred to at the time of the call:**

```
class Dispatch {
   public static void main(String args[]) {
          A a = new A();
          B b = new B();
          C c = new C();
          A r;
          r = a; r.callme();
          r = b; r.callme();
          r = c; r.callme();
   }
}
```

17. WAP in JAVA to implement a stack that can hold 10 integer values.
18. Write a JAVA program to implement stack operations.

```
class Stack {
  int stck[] = new int[10];
  int tos;
  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }
  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());
  }
}
```

```
Output
9
8
```

```
7
6
5
4
3
2
1
0
```

19. What is a package? What are the steps involved in creating user defined packages.

A **package** is both a naming and a visibility control mechanism:

1. divides the name space into disjoint subsets
   - It is possible to define classes within a package that are not accessible by code outside the package.
2. controls the visibility of classes and their members
   - It is possible to define class members that are only exposed to other members of the same package.
   - Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages.

Defining Package

- A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- It means that all classes in this file belong to the myPackage package. The package statement creates a name space where such classes are stored.
- When the package statement is omitted, class names are put into the default package which has no name.
- Other files may include the same package instruction:
- A package may be distributed through several source files.

```
// A Source File
package myPackage;
class MyClass1 { … }
class MyClass2 { … }

   // Another Source File
package myPackage;
class MyClass3{ … }
```

Packages and directories: Java uses file system directories to store packages. Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The bytecode files MyClass1.class and MyClass2.class must be stored in a directory myPackage.
- Case is significant! Directory names must match package names exactly.
- To create a package hierarchy, separate each package name with a dot:
  `package myPackage1.myPackage2.myPackage3;`
- A package hierarchy must be stored accordingly in the file system: On Windows
  `myPackage1\myPackage2\myPackage3`

Finding Packages

- As packages are stored in directories, how does the Java run-time system know where to look for packages?
- Two ways:
  1) The current directory is the default start point – if packages are stored in the current directory or sub-directories, they will be found.
  2) Specify a directory path or paths by setting the CLASSPATH environment variable.

Package Example:

```
package MyPack;
class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
            name = n; bal = b;
    }
    void show() {
            if (bal<0) System.out.print("-->> ");
            System.out.println(name + ": $" + bal);
    }
}
class AccountBalance {
    public static void main(String args[]) {
            Balance current[] = new Balance[3];
            current[0] = new Balance("K. J. Fielding", 123.23);
            current[1] = new Balance("Will Tell", 157.02);
            current[2] = new Balance("Tom Jackson", -12.33);
            for (int i=0; i<3; i++) current[i].show();
    }
}
```

Save, compile and execute:

1) call the file AccountBalance.java
2) save the file in the directory MyPack
3) compile; AccountBalance.class should be also in MyPack
4) set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
5) run: **java MyPack.AccountBalance**

Make sure to use the package-qualified class name.

Importing of Packages

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.
- The **import** statement allows to use classes or whole packages directly.
- Importing of a concrete class:
  import myPackage1.myPackage2.myClass;
- Importing of all classes within a package:
  import myPackage1.myPackage2.*;

20. What is an exception'? Explain the different exception handling mechanisms with an example. OR
What is a Java exception'? Explain the exception handling mechanism with an example

What is an exception'?

Exception is an abnormal condition that arises when executing a program. Java provides syntactic mechanisms to signal, detect and handle errors. An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.

Exception handling involves the following:

1) When an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
2) that method may choose to handle the exception itself or pass it on
3) either way, at some point, the exception is caught and processed
Five constructs are used in exception handling:

1. **try** – a block surrounding program statements to monitor for exceptions
2. **catch** – together with try, catches specific kinds of exceptions and handles them in some way
3. **finally** – specifies any code that absolutely must be executed whether or not an exception occurs
4. **throw** – used to throw a specific exception from the program
5. **throws** – specifies which exceptions a given method can throw

General Form of Exception handling is given below:

```
try {
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}
```
Here, *ExceptionType* is the type of exception that has occurred.

21. Define exception. Demonstrate the working of nested try blocks, with suitable examples. OR What is an exception'? Give an example for nested try statements.

Nested try Statements: When a try block is nested within another try block, it's known as nested try block.

1) if an inner try does not catch a particular exception
2) the exception is inspected by the outer try block
3) this continues until:
    - a) one of the catch statements succeeds or
    - b) all the nested try statements are exhausted
4) in the latter case, the Java run-time system will handle the exception

An example with two nested try statements is given below:

- Catch statement for the inner try statement, catches the array index out of bound exception:
- Catch statement for the outer try statement, catches both division-by-zero exceptions for the inner and outer try statements

```
class NestTry {
    public static void main(String args[]) {
            try { // Outer try statement
                    int a = args.length;
                    int b = 42 / a; // Division by zero when no argument
                    System.out.println("a = " + a);
                    try { // Inner try statement
                            // Division by zero when single argument
                            if (a==1) a = a/(a-a);
                            // Array index out of bound when two arguments
                            if (a==2) {
                                    int c[] = { 1 };
                                    c[42] = 99
                            }
                    } catch(ArrayIndexOutOfBoundsException e) {
                            System.out.println(e);
                    }
            } catch(ArithmeticException e) {
                    System.out.println("Divide by 0: " + e);
            }
    }
}
```

22. Write a program which contains one method which will throw IllegalAccessException and use proper exception handlers so that exception should be printed.

Consider the class ThrowsDemo which defines the method `throwOne()` which throws IllegalAccessException.
The main calls `throwOne()` in its try block and handles the IllegalAccessException in its catch block prints the exception.

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
            System.out.println("Inside throwOne.");
            throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
            try {
                    throwOne();
            } catch (IllegalAccessException e) {
                  System.out.println("Caught " + e);
            }
      }
}
```

23. Explain the java's built in exceptions.

Java Built-In Exceptions

The default java.lang package provides several exception classes, all sub-classing the RuntimeException class.

There are two sets of build-in exception classes:

1. **unchecked exceptions –** the compiler does not check if a method handles or throws these exceptions
2. **checked exceptions –** must be included in the method's throws clause if the method generates but does not handle them

Following table provides a list of Java's **Unchecked** RuntimeException Subclasses.

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type
ClassCastException	Invalid cast
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security
StringIndexOutOfBounds	Attempt to index outside the bounds of a string
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Following table provides a list of Java's **Checked** Exceptions

Exception	**Meaning**
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist

24. Create a try block that is likely to generate three types of exception and incorporate necessary catch blocks to catch and handle them.
OR
Write the syntax of try and catch block to handle the multiple exceptions, Explain.

When more than one exception can be raised by a single piece of code, several catch clauses can be used with one try block:

1. each catch catches a different kind of exception
2. when an exception is thrown, the first one whose type matches that of the exception is executed
3. after one catch executes, the other are bypassed and the execution continues after the try/catch block

Three different exception types are possible in the following code: incorrect argument type, division by zero and array index out of bound:

```
class MultiCatch {
   public static void main(String args[]) {
         try {
               int a = args.length;
               int a= Integer.parse.Int(args[0]);
               int b = 42 / a;
               int c[] = { 1 };
               c[42] = 99;
         }
         catch (NumberFormatException e) {
               System.out.println ("Incorrect argument type: " + e);
         }
         catch(ArithmeticException e) {
               System.out.println("Divide by 0: " + e);
         } catch(ArrayIndexOutOfBoundsException e) {
               System.out.println("Array index oob: " + e);
         }
         System.out.println("After try/catch blocks.");
   }
}
```

25. Consider the following code:

```
   class nested-try {
      public static void main (string args[] ) {
         try {
                  int a= Integer.parse.Int (args [0]);
                  int b = Integer.parse.Int (args[1]);
                  int quot = 0;
                  try  {
                        quot = a/b;
                        system.out.println(quot);
                        try {
                              system.out.println("a * b = " + (a * b));
                              if (a * b < 0)
                                    throw new array index out of bounds exception( )
                           }
                           finally {
                              system.out.println ("in finally block");
                           }
                        catch (Arithmetic Exception e)  {
                              system.out.println ("Divide by zero");
                           }
                        catch (Number Format Exception e) {
                              system.out.println ("Incorrect argument type");
                           }
                     }
               }
         }
   }
```

Indicate the output of the above code for following runs: i) Java Nested-Try 24 6 ii) Java Nested-Try 24 aa
iii) Java Nested-Try 24 0 iv) Java Nested-Try -1 5

Corrected Code:

```
class NestedTry {
        public static void main (String args[] ) {
                try {
                        int a= Integer.parseInt (args [0]);
                        int b = Integer.parseInt (args[1]);
                        int quot = 0;
                        try  {
                                quot = a/b;
                                System.out.println(quot);
                                try {
                                        System.out.println("a * b = " + (a * b));
                                        if (a * b < 0)
                                           throw new ArrayIndexOutOfBoundsException();
                                }
                                catch (ArrayIndexOutOfBoundsException e) {
                                         System.out.println("Array is out of Bounds");
                                }
                        }
                        catch (ArithmeticException e)  {
                                System.out.println ("Divide by zero");
                        }
                }
                catch (NumberFormatException e) {
                        System.out.println ("Incorrect argument type");
                }
                finally {
                        System.out.println ("in finally block");
                }
        }
}
```

i)

```
NestedTry 24 6
Output:
4
a * b = 144
in finally block
```

ii)

```
NestedTry 24 aa
Output:
Incorrect argument type
in finally block
```

iii)

```
NestedTry 24 0
Output:
Divide by zero
in finally block
```

iv)

```
NestedTry -1 5
Output:
0
a * b =-5
Array is out of Bounds
in finally block
```

26. Differentiate the usage of access specifiers in java and their score.
27. Explain different access specifiers in Java, with examples.

Java provides several access specifiers to control the access of classes as well as the variables, methods and constructors in your classes. These Access Specifiers are: i) **private** ii) **protected** iii) **default** iv) **public**

A member (variable, method) has a default accessibility when no accessibility modifier is specified.

private access modifier: Variables and methods declared private cannot be accesses by anywhere outside the enclosing class. They are accessible only within the class.

```
private int variable;
private int method(…) { … }
```

protected access modifier: Variables and methods declared protected are accessible within the same package as the current class, or within its sub-classes. The sub-class may be located inside or outside the current package.

```
protected int variable;
protected int method(…) { … }
```

default access modifier: Java provides a default specifier which is used when no access modifier is present. Variables and methods with no declared access modifier are accessible only by classes in the same package.

```
int variable;
int method(…) { … }
```

public access modifier:

Variables and methods declared public within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

```
public int variable;
public int method(…) { … }
```

Following table differentiates/compares the access control provided by various access modifier:

	private	default	protected	public
same class	**yes**	**yes**	**yes**	**yes**
same package subclass	**no**	**yes**	**yes**	**yes**
same package non-sub-class	**no**	**yes**	**yes**	**yes**
different package sub-class	**no**	**no**	**yes**	**yes**
different package non-sub-class	**no**	**no**	**no**	**yes**

Following example explain access control with two packages p1 and p2 and five classes.

```
// A public Protection class is in the package p1. It has four variables with four possible access
rights:

package p1;
public class Protection {
       int n = 1;
       private int n_pri = 2;
       protected int n_pro = 3;
       public int n_pub = 4;
       public Protection() {
              System.out.println("base constructor");
              System.out.println("n = " + n);
              System.out.println("n_pri = " + n_pri);
              System.out.println("n_pro = " + n_pro);
              System.out.println("n_pub = " + n_pub);
       }
}

// Derived class is in the same p1 package and is the sub-class of Protection.
// It has access to all variables of Protection except the private n_pri:
package p1;
class Derived extends Protection {
       Derived() {
              System.out.println("derived constructor");
              System.out.println("n = " + n);
              System.out.println("n_pro = " + n_pro);
              System.out.println("n_pub = " + n_pub);
       }
}

// SamePackage is in the p1 package but is not a sub-class of Protection.
// It has access to all variables of Protection except the private n_pri: package p1;
class SamePackage {
       SamePackage() {
              Protection p = new Protection();
              System.out.println("same package constructor");
              System.out.println("n = " + p.n);
              System.out.println("n_pro = " + p.n_pro);
              System.out.println("n_pub = " + p.n_pub);
       }
}

// Protection2 is a sub-class of p1.Protection, but is located in a different package p2.
// Protection2 has access to the public and protected variables of Protection.
// It has no access to its private and default-access variables:
package p2;
```

```
class Protection2 extends p1.Protection {
	Protection2() {
		System.out.println("derived other package");
		System.out.println("n_pro = " + n_pro);
		System.out.println("n_pub = " + n_pub);
	}
}

// OtherPackage is in the p2 package and is not a sub-class of p1.Protection.
// OtherPackage has access to the public variable of Protection only.
// It has no access to its private, protected or default-access variables:
class OtherPackage {
	OtherPackage() {
		p1.Protection p = new p1.Protection();
		System.out.println("other package constructor");
		System.out.println("n_pub = " + p.n_pub);
	}
}

// A demonstration to use classes of the p1 package:
package p1;
public class Demo {
	public static void main(String args[]) {
		Protection ob1 = new Protection();
		Derived ob2 = new Derived();
		SamePackage ob3 = new SamePackage();
	}
}

// A demonstration to use classes of the p2 package:
package p2;
public class Demo {
	public static void main(String args[]) {
		Protection2 ob1 = new Protection2();
		OtherPackage ob2 = new OtherPackage();
	}
}
```

Java Classes have two levels of access:
public access: A class available in the whole program:

```
public class MyClass { … }
```

default access: When no access specifier used, a class is available within the same package only:

```
class MyClass { … }
```