NJC LABS ACCELERATOR PROGRAM

SESSION 4 NOTES

Object Oriented Programming Fundamentals

1. **What is the main difference between a class and an object?**
   Classes and objects are essential part of Object-oriented Programming, where class is a construct that encapsulates a group of variables and methods while an object is an instance of the class. A class is like a blueprint or template to create an instance, referred to as an object.

2. **What is Encapsulation? Explain with a use-case.**
   Encapsulation is one of the fundamental OOP concepts. It is a mechanism of wrapping data (variables) and code that works on the data (methods) as one single unit. Encapsulation implements data hiding, that hides the data and implementation to other classes. For example, a car has many sub-parts like brakes, engine, steering, clutches (like variables) and the there is a lot of mechanism that involves all these parts (similar to methods). Here, we can say a car encapsulates all its parts and mechanism, similar to classes that implement encapsulation.

3. **What is Polymorphism? Explain with a use-case.**
   Polymorphism is one of the fundamental concepts of OOP. IT is the ability of an object to take many forms. When a single object passes the IS-A relationship test for more than one class, it can be said to be polymorphic. An object can be accessed through a reference variable. A reference variable can only be of one type and once assigned it can not be changed unless it is reassigned. A reference variable can point to any object of its declared type or any of its sub-type. Consider the following declarations in JAVA:

   ```
   public interface Vegetarian{}
   public class Animal{}
   public class Deer extends Animal implements Vegetarian{}
   ```

   Here, the class Deer can be considered to polymorphic as it has multiple inheritance, that is follows multiple IS-A test:
   A Deer IS-A Vegetarian
   A Deer IS-A Animal
   A Deer IS-A Deer
   A Deer IS-A Object
   This holds true in real-time as well.

   When we add reference variable concept to the above classes, it is legal to declare the following:

   ```
   Deer d = new Deer();
   Animal a = d;
   Vegetarian v = d;
   Object o = d;
   ```

   All the reference variable, d, a, v and o refer to the same Deer object in the storage heap.

   JAVA has two types of polymorphism:
   - Static polymorphism or compile-time polymorphism

- Dynamic or run-time polymorphism

The above use-case is an example of run-time polymorphism.

4. **Explain Overriding & Overloading and its advantages.**
Overriding and Overloading are ways to implement polymorphism.

Overriding means having a method with the same name and same parameters (i.e. method signature) in both the parent class and the child class. This allows to have a specific implementation to a method in a child class while it is already defined in the parent class.
For example, consider a parent class Animal and two child classes Monkey and Dog. We can have an activity like “eat” declared as a method in Animal. The same method maybe declared with the same name “eat” in both child classes Monkey and Dog but what happens inside that method may be different.
The benefit of overriding is the ability to define a behaviour that is specific to the subclass type, which means that the subclass type can implement a parent class method based on its requirement.

Overloading means having the more than method with the same name but different parameters in the same class. Overloading can be implemented by changing number of parameters or datatype of the parameters or changing both.
Consider a class Adder for adding numbers, with method “add”. There are the following methods in the class Adder:

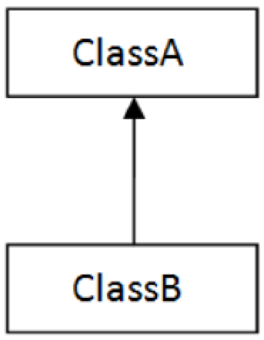
```
class Adder{
static int add(int a,int b){return a+b;} // method 1
static int add(float a,float b){return a+b;} //method 2
static int add(int a,float b){return a+b;} //method 3
static int add(int a,int b,int c){return a+b+c;}  //method 4
}
```

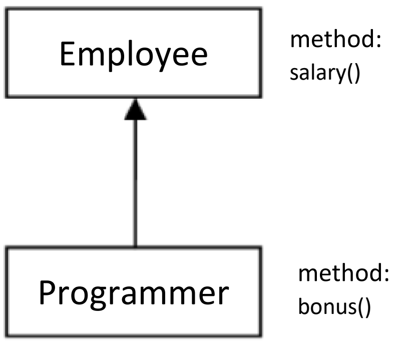
There may be many more methods with the same name and different parameters based on the requirement. When the method is called with any arguments, the compiler will call the correct method based on the arguments. If the program calls add method with two integer arguments, the method 1 from the Adder class is called. Overriding allows to declare methods with similar behaviour but different arguments to be implemented with same name to avoid creating many methods with different names and also avoid to remember so many names, making it easy for programmers to use the program.
Overloading increases the readability of the program.

5. **What is Inheritance and different types of inheritance? Explain with a use-case.**
Inheritance is an important concept of OOP. It is a mechanism in which one class acquires the properties and behaviour of another class. The class which inherits the properties of another class is referred to as child class or subclass, while the class whose properties are inherited is called parent class or super class.
Inheritance is used:
   - for method overriding (to implement runtime polymorphism)
   - for code reusability

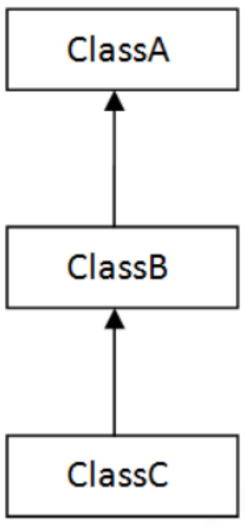
There are different types of inheritance that are realized in OOP. Based on the class, there are three types of inheritance – single, multilevel, hierarchical.

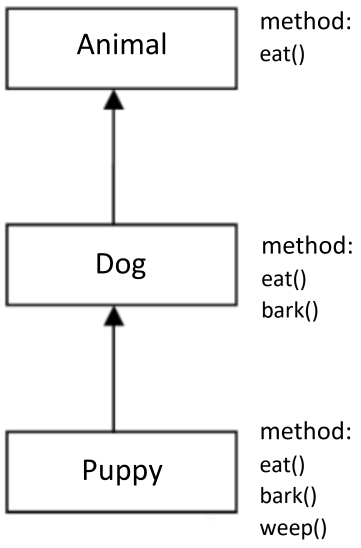
When only one class inherits the properties of another class, it is called **Single** inheritance. For example, a class Employee can be a super-class and a Programmer is a subclass as it has the same properties and behaviour as an employee as well as few specific properties and behaviour of its own. Consider a subclass Programmer, it inherits all the properties and behaviour of super class Employee. The subclass Programmer will have fields and parameters of its own.



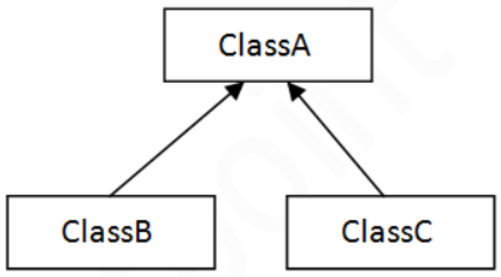


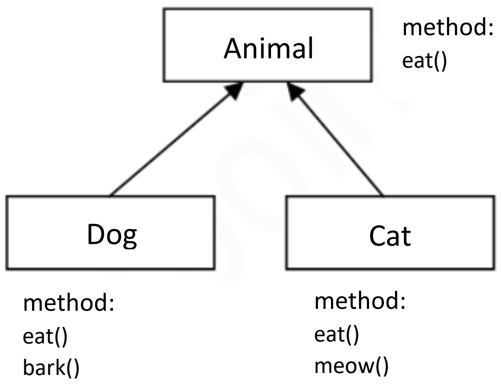
When a class inherits from a derived class(subclass) making it the super class for the newly derived class, it is called **Multilevel** inheritance. Consider a subclass Dog which inherits all the properties of the super class Animal. A new derived class Puppy would inherit the properties of both the classes Dog and Animal.





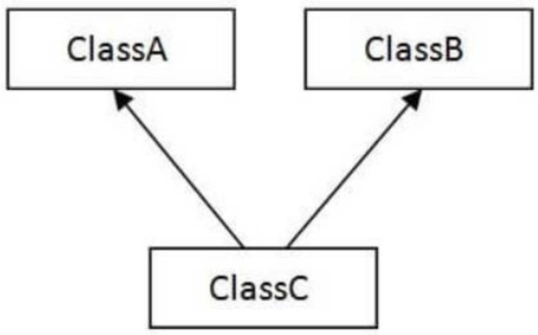
When one class is inherited by two or more subclasses, it is called **Hierarchical** inheritance. Consider a super-class Animal, many subclasses like Dog and Cat both inherit the properties of super class Animal while each have their own specific methods.

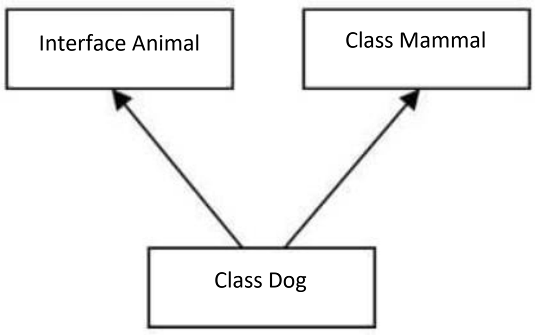




In JAVA, multiple and hybrid inheritance are two types of inheritances that can be realised using interfaces only.

When one class inherits multiple parent classes, it is known as **Multiple** inheritance.





6. **What is an abstract class?**
   Abstraction is the process of hiding the implementation and showing only the functionality to the user.
   A class which is declared with ‘abstract’ keyword is called an abstract class in JAVA. It can have abstract and non-abstract methods. It needs to be extended and its methods implemented. An abstract class can not be instantiated.

7. **What is an interface and how multiple inheritance is achieved with this?**
   An interface is a reference type similar to abstract class. It is a collection of abstract methods. A class implements an interface inheriting all the abstract methods of the interface. Along with abstract methods, interface may also contain constants, default methods, static methods, and nested types.
   Interfaces can not be instantiated as it does not have actual implementation at all. But classes can implement the interface. An interface can extend another JAVA interface only.
   When one class inherits more than one class, it is called multiple inheritance. Multiple inheritance is realized when a class implements one or more interfaces. A class can not extend more than one parent class but can implement more than one interfaces.

8. **What are the access modifiers?**
   The access modifiers in JAVA specifies the accessibility or scope of a field, method, class or constructor. The access level of a field, method, class or constructor can be changed by applying the access modifier to it while declaration.
   There are four types of JAVA access modifiers:
   - Private – The access level of private modifier is only within the class. It can not be accessed from outside the class.
   - Default – The access level of a default modifier is only within the package. It can not be accessed from outside the package. If no access modifier is specified, it will automatically be default.
   - Protected – The access level of a protected modifier is within the package and outside package through child classes. Only a child class outside package can enable to access it from outside the package.
   - Public – The access level of public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

9. **What are the various types of constructors?**

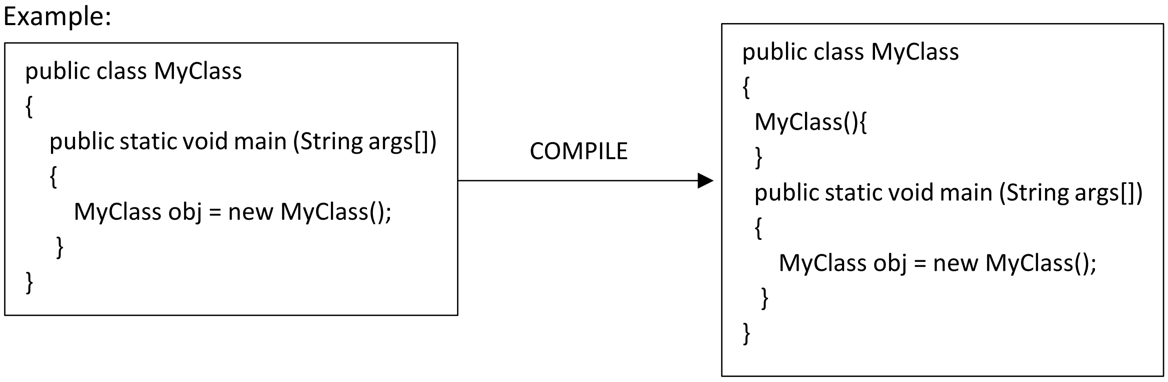
Constructor is a block of code similar to a method. It is called when an instance of a class is created. A constructor resembles an instance method but its not a method as it does not have a return type.

There are three types of constructors: Default, No-arg, Parameterized

Default Constructor

If a constructor is not implemented in a class, the JAVA compiler inserts a constructor into the code at compile time which is known as default constructor. A default constructor does not necessarily do anything and is not found in the source file. Since it is inserted during compilation, the default constructor exists in '.class' file.

Example:



```
public class MyClass
{
  public static void main (String args[])
  {
    MyClass obj = new MyClass();
   }
}
```

```
public class MyClass
{
 MyClass(){
 }
 public static void main (String args[])
 {
   MyClass obj = new MyClass();
  }
}
```

No-Arg Constructor

Constructor with no arguments is known as No-arg Constructor. The signature of no-arg constructor is same as default constructor. However, no-arg constructor has code in its method body, unlike default constructor which is empty.

Example:

```
public class MyClass
{
  public MyClass()
  {
   System.out.println("This is a no argument constructor.");
  }
  public static void main (String args[])
  {
    MyClass obj = new MyClass();
  }
}
```

Output:

```
This is a no argument constructor.
```

Parameterized Constructor

Constructor with argument(s) is known as parameterized constructor.

Example:

```
public class MyClass
{
  String myName;
  public MyClass(String name)
  {
   myName = name;
  }
   public void display()
   {
     System.out.println("My name is "+ myName);
   }

  public static void main (String args[])
  {
    MyClass obj = new MyClass("John");
  }
}
```

Output:

```
My name is John
```

10. What is 'this' pointer?

'this' keyword in JAVA is a reference variable that refers to the current object of the instance method or instance class. Any member of the current object can be referred from within an instance method or instance constructor by using 'this' keyword.

Using 'this' with a Field

'this' keyword can be used to refer to current class instance variable. If there is an ambiguity between instance variables and parameters, 'this' keyword resolves the ambiguity.

The most common reason for using 'this' keyword is because a field is shadowed by a method or constructor parameter. In a method, a field variable with a 'this' keyword refers to the instance variable and not the parameters.

For example:

```
public class Point
{
  public int x;
  public int y;

  //constructor
  public Point (int a, int b)
   {
      x = a;
      y = b;
    }
}
```

Can also be written as →

```
public class Point
{
  public int a;
  public int b;

  //constructor
  public Point (int a, int b)
   {
      this.a = a;
      this.b = b;
    }
}
```

Using 'this' with a constructor

From within a constructor, 'this' keyword can be used to invoke another constructor of the same class. This is called explicit constructor invocation.

```
public class Rectangle {
   private int x, y;
   private int width, height;

   public Rectangle() {
      this(0, 0, 1, 1);
```

```
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

11. **What is static and dynamic Binding?**

Connecting a method call to a method body is known as 'Binding'. There are two types of binding: Static Binding and Dynamic Binding.

Static Binding

The binding which can be resolved during compile time by compiler is known as static binding or early binding. The binding of static, final and protected methods is compile-time as these methods can not be overridden, and the class type of the object can be determined at compile time.

For example:

```
public class Shape
{
    public static void show()
    {
        System.out.println("This is a Shape.");
    }
}
public class Square extends Shape
{
    public static void show()
    {
        System.out.println("This is a Square.");
    }
    public static void main(String args[])
    {
        Shape obj = new Square();
        Shape obj2 = new Shape();

        obj.show();
        obj2.show();
    }
}
```

```
Output:

This is a Shape.

This is a Shape.
```

Here, the method show() declared in class Shape is static which means that method show() can not be overridden. Even though a Square object is used during the creation of object obj, parent class Shape is called because reference is of Shape type (parent type).

Dynamic Binding

When the biding happens at runtime, it is known as dynamic binding or late binding. Method overriding is the best example of dynamic binding. Both parent and child class have the same methods due to method overriding, and only the type of object created can resolve which method is to be executed. The type of object can be determined only at runtime.
For example:

```
public class Shape
{
    public void show()
    {
       System.out.println("This is a Shape.");
    }
}
public class Square extends Shape
{
    public void show()
    {
       System.out.println("This is a Square.");
    }
    public static void main(String args[])
    {
       Shape obj = new Square();
       Shape obj2 = new Shape();

       obj.show();
       obj2.show();
    }
}
```

```
Output:

This is a Square.

This is a Shape.
```

12. How many instances can be created for an abstract class and why?

No instances can be created for an abstract class as an abstract class is partially or completely empty structure, that is, it does not have any implementation. An abstract class is like a template which can be extended by other classes to implement its abstract methods. Any number of instances can be created for the classes that extend the abstract classes though.
For example:

```
//abstract class
abstract class Shape
{
    //abstract method
    public abstract void show()

}
public class Square extends Shape
{
    public void show()
    {
       System.out.println("This is a Square.");
    }
    public static void main(String args[])
    {
      Shape obj = new Square();

      obj.show();
    }
}
```

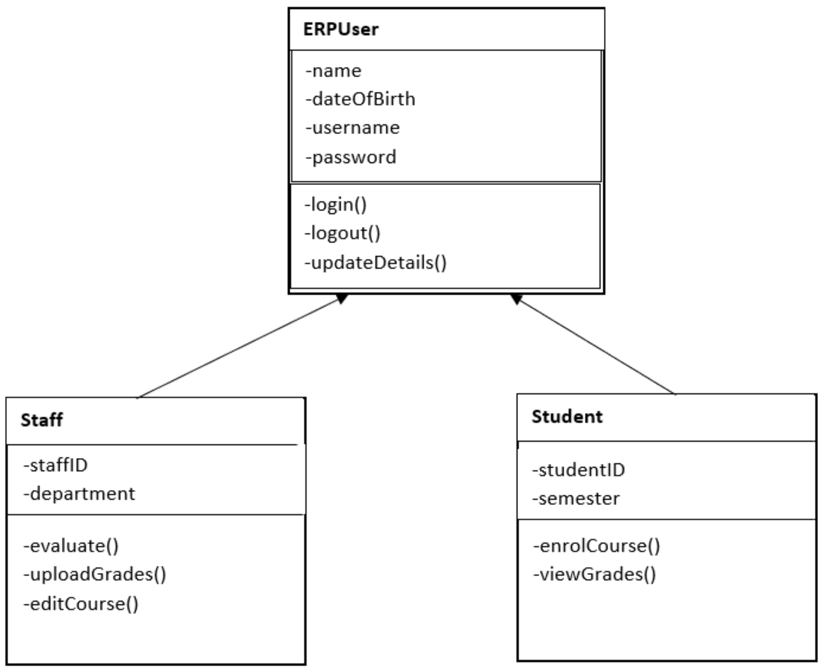
```
Output:

This is a Square.
```

13. Which OOPS concept is used as a reuse mechanism and explain with a use case?

Inheritance is the OOP concept that is used as reuse mechanism. An inheritance allows a child class to inherit the attributes(field variables) and behaviours(methods) of the parent class.
In regard to reuse mechanism, inheritance allows to add additional features to a child class without modifying the parent class altogether. The child class will have the features of both classes, the parent class as well as the new additional features of the child class.

Consider a University ERP system. There are several users of the ERP System, consider each of them is known as ERPUser. An ERPUser will have their name, date of birth, sex, address, username, password and other personal details stored in the system that the University needs. A ERPUser will be able to perform many functions like login, logout and update details in the ERP System.
There will be several users of ERP system like staff of the university and the students. The ERP system needs same kind of information from both the users as well as additional information and functions related only to specific kind of users. So, staff and students can be considered subclasses of the parent class ERPUser in the following way:



The above class diagram shows the inheritance concept of OOP where Staff and Student classes inherit the properties and behaviour of ERPUser class. Each class then has their own properties and behaviours. This enables to reuse the properties and behaviours created for ERPUser to be used by its subclasses to avoid repetitive coding. This saves time and ensures to maintain consistency across the classes.

14. **Please identify one practical scenario for each pillar of OOPs.**

The four pillars of OOP are – Encapsulation, Polymorphism, Abstraction and Inheritance.
To briefly explain the four pillars of OOP, let us consider a mobile phone.

Encapsulation

Encapsulation can be defined as wrapping of data in one single unit. Consider a mobile phone, it can perform many functions like call, message, click pictures, store contacts, and has a name, OS version, a unique device ID. All these are encapsulated into one single physical device called mobile phone such that it all function together properly. It is secure from other mobile devices, that is, other mobile devices can not access its data unless given access to. A mobile phone can give access to other phones using bluetooth but the other phones can access the only limited data given permissions to and can not access the functions or private information stored in the phone. This is due to level of permission given to the other phone, similar to level of abstraction in OOP.

Polymorphism

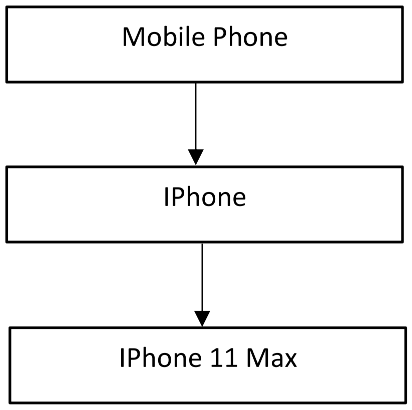
Polymorphism can be defined as one entity taking many forms. Latest mobile phones have the basic function of clicking photos. A phone can click a normal photo, panorama photo or night mode photo based on the mode we select. The same camera of the same one phone is able to click different kinds of photos. One phone is acting as different kinds of cameras, that is, it is taking different forms. The same functionality of clicking pictures works differently with different modes.

Abstraction

Abstraction can be defined as the mechanism of data hiding. This mechanism shows only relevant data and hides the unnecessary information. A mobile phone can be used to call another number. We enter the mobile number on our phone and press the dial button to make a call. However, we do not know what functions actually runs in the background to actually place a call, that is, the implementation is hidden. Only the number keypad and the dial button are visible to the user whereas the call request functionality that takes place in the mobile phone is hidden.

Inheritance

Inheritance is referred to as a mechanism to reuse or share information. The latest mobile phones have the basic functions of calling, messaging, clicking pictures and manage contacts. Consider a mobile phone of the brand IPhone. It can perform the same basic functionality of a mobile phone with additional features like more modes of camera, facetime, etc. An Iphone inherits the functionality of mobile phone and develop its own additional features. An Iphone 11 Max is like a child of Iphone which inherits the functionalities of both mobile phone and Iphone as well as has its own advanced features. The idea behind the functionalities is shared and improved with every advanced device.



Unit Testing & JUnit

1. **What is unit testing?**
   Unit Testing is the first level of testing where individual units or components of an application or system are tested. The purpose is to validate each individual unit as soon as it is implemented so that the bugs can be found in initial stages and the bigger components are easily implemented at later stages. A unit is the smallest testable parts of an application or system with fewer inputs and usually a single output.
   Unit testing may sometimes be performed by the developer after implementation or a single software tester. Only after every individual unit is tested, they are integrated together to form a bigger component which is then tested.

2. **What is the difference between manual testing and automated testing?**

Manual Testing	Automated Testing
Tests are executed manually by QA Analysts.	Testers write code/test scripts to automate tests execution.
Time-consuming.	Faster than manual approach.
Not accurate at all time due to human errors. Less reliable.	Performed by tools and/or scripts. More reliable.
Invested required for human resources.	Investment required on tools.
Exploratory testing is possible.	Pre-scripted tests which runs to compare actual result with expected result.
Allows human observation, which may be useful for user-friendly.	Does not consider human involvement. Can never give assurance of user-friendliness and positive customer experience.
Can be executed in parallel but would need to increase human resources which is expensive.	Testing can be executed on different platforms in parallel. Reduces test execution time.
Batch testing not possible.	Can batch multiple tests for nightly execution.
Practical only when test cases are run once or twice, and frequent repetition is not required.	Practical when test cases are run repeatedly over long period of time.

3. **Is it necessary to write the test case for every logic? If yes, why?**
   It is necessary to validate or test every logic of a code to detect a bug at early stages of development. It may not be required to test simple declaration statements or assignment statements, but it is helpful to check every logic in a code. With better coverage of testing, there are low chances of unexpected or unidentified bugs to be encountered when the code is tested as a whole. Also, it is very common for the requirements to be changed and code be updated frequently so by testing every logic, it ensures to have lower bug detection in future, maintain standards and deliver best quality code.

4. **What are the features of JUnit?**
   JUnit is an open source framework, which is used for writing and running tests. It plays a crucial role in test-driven development and is a family of unit testing frameworks collectively known as xUnit.
   Salient features of JUnit:
   - Provides annotations to identify test methods.

   - Provides assertions for testing expected results.

- Provides test runners for running tests.
- Allows to write codes faster, which increases quality.
- It is simple: Less complex and takes less time.
- JUnit tests can be run automatically, and they check their own test results and provide immediate feedback. There is no need to manually check through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

5. **What are the important JUnit annotations? And its usage in coding**
   In programming paradigm, there are two approaches: imperative programming and declarative programming. Imperative programming specifies the algorithm to achieve a goal (what has to be done) while the declarative programming specifies how to achieve that goal (how it has to be done).
   Declarative programming is done using metadata, that is, annotation and/or deployment descriptors. JUnit annotations are special form of syntactic metadata that can be added to Java code for better code readability and structure for users. The annotations tell the compiler how it has to run the code. Annotations are generally placed on methods, which tell the compiler when to run that method. The methods in test classes do not take any parameters and are generally declared *public*.
   The important and most used JUnit4 annotations are:
   - @BeforeClass – Run once before any of the methods in the class. It is generally declared to be '*public static* void' method, that is, it is loaded before an instance of the class is created. The purpose of method annotated with @BeforeClass can be to allocate external computational resources for all the methods in the class, establish a database connection, log information, etc.
   - @AfterClass – Run once after all methods in the class are executed. These methods are also declared '*public static void*' and are run even if @BeforeClass throws an exception. The purpose of @AfterClass annotated method is to clean up memory, free external computational resources, close database connection or to log information, etc.
   - @Test – This is the actual test method to be run. It is generally declared '*public void*'.
   - @Before – Run before each test method in the class is run. Declared '*public void*'. It is common that every test method needs similar objects created before they are run. These methods are used to create necessary objects before test methods are run or log information.
   - @After – Run after each test method in the class is run. Declared '*public void*'. It is used to free any resources allocated in @Before annotated methods or log information.

6. **What does Assert class?**
   Test methods need to confirm (or assert) that the test results conform to expected results. To help the test methods to do so, Assert class provides with static assert methods which help in

writing test codes. Assertions are like utility methods to support in writing asserting conditions in tests. The assert methods can be used directly in the tests as 'Assert.assertEquals(….)' or for better readability be imported in the class using the statement 'import static org.junit.Assert.*'.
Example:

```
import org.junit.Test;
import org.junit.Assert.*;

public class TestAssertions {
    @Test
    public void testAssertions(){
      //Test data
      String testString = new String("testData");
      String expectedString = new String("testData");

      //Check if two objects are equal
      assertEquals(expectedString, testString);
    }
}
```

Following are the most commonly used assert methods:

S.No.	Method and Description
1	**void assertEquals(boolean expected, boolean actual)** Checks that two primitives/objects are equal.
2	**void assertTrue(boolean condition)** Checks that a condition is true.
3	**void assertFalse(boolean condition)** Checks that a condition is false.
4	**void assertNotNull(Object object)** Checks that an object isn't null.
5	**void assertNull(Object object)** Checks that an object is null.
6	**void assertSame(object1, object2)** The assertSame() method tests if two object references point to the same object.
7	**void assertNotSame(object1, object2)** The assertNotSame() method tests if two object references do not point to the same object.
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.

7. What is Code Coverage?

Code Coverage is a software metric that helps to understand how much of the code was tested during the execution of test suites. It is helpful to assess the quality of test suites created. There are several tools that can give a detailed report of the code coverage of test suites.

Most of the tools take into consideration few critical coverage criteria while generating code coverage reports. Following are few critical coverage criteria:

i. **Function Coverage** – Number of functions in the source code that are called and executed at least once.
ii. **Statement Coverage** - Number of statements that have been successfully validated in the source code.
iii. **Branch or Decision Coverage** – Number of decision control structures (loops, if statements for example) that have executed fine for at least once.
iv. **Condition Coverage** – Number of boolean expressions that are validated and that executes both TRUE and FALSE as per the test runs.
v. **Line Coverage** – Number of lines of code that are executed.

Code coverage is generally measured in percentage. A code coverage percentage is returned for each criterion. For example, line coverage can be calculated as follows:

Code Coverage Percentage = (Number of lines of code executed by a testing algorithm/Total number of lines of code in a system component) * 100

8. What are the best practices to perform Unit Testing?

Following are the best practices to perform unit testing:

- Unit Tests should be trustworthy – The unit test must fail if and only if the code is broken lest the results of the test cannot be trusted.
- Unit Tests should be maintainable and readable – Tests should be easy to read and understand what it is testing or how it is testing. It should be understandable to any other tester or developer as well apart from the person who wrote it to enable them to make changes or update them if necessary.
- Unit tests should verify a single use-case – Good tests validate one and only one use-case. If a test verifies multiple use-cases, it would be ambiguous or difficult to identify the cause of the bug when the test fails. It would be easier to identify the location of possible bug if a test verifies only one use-case at a time.
- Unit Test should be isolated – Tests should be runnable on any machine and in any order without affecting each other. Unit tests should not have any dependencies on other environmental or global states as it would influence the result of the test and a change in state may lead to unexpected result. Tests that have dependencies can be difficult to run and usually be unstable, making them harder to debug or fix.

9. What is Mocking?

In simple English, mocking is making a replica or imitation of something.

The best practice to perform unit testing is to run unit tests on a class in isolation without any dependencies. However, it is common for a class to use other services or methods from other classes during execution. To perform unit testing in isolation, these services or methods on which a class depends on are replicated and replaced with fake objects that simulate the behaviour of the real services/methods. This helps the test suites to test a class in isolation without any external dependencies.

Mocking is a framework that generate replacement objects for dependencies, like Stubs or Mocks.

Mock is a replica of the service or a method and mimics the real one. A stub is like mocking but it only mocks the behaviour of the real object. It can not be used to interact with all its properties or functions similar to a real one.