85 Core Java Interview Questions and Answers

(For Beginners)

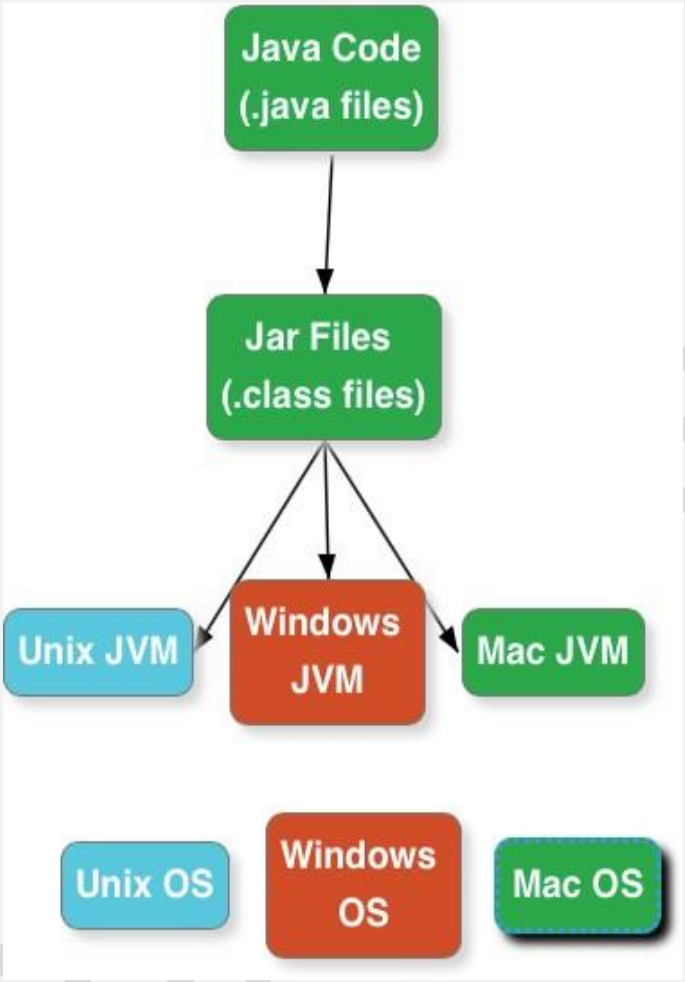
1) Why is Java so Popular?

Two main reasons for popularity of Java are

1. Platform Independence
2. Object Oriented Language

We will look at these in detail in later sections.

2) What is Platform Independence?



Platform Independence is also called build once, run anywhere. Java is one of the most popular platform independent languages. Once we compile a java program and build a jar, we can run the jar (compiled java program) in any Operating System - where a JVM is installed.

Java achieves Platform Independence in a beautiful way. On compiling a java file the output is a class file - which contains an internal java representation called bytecode. JVM converts bytecode to executable

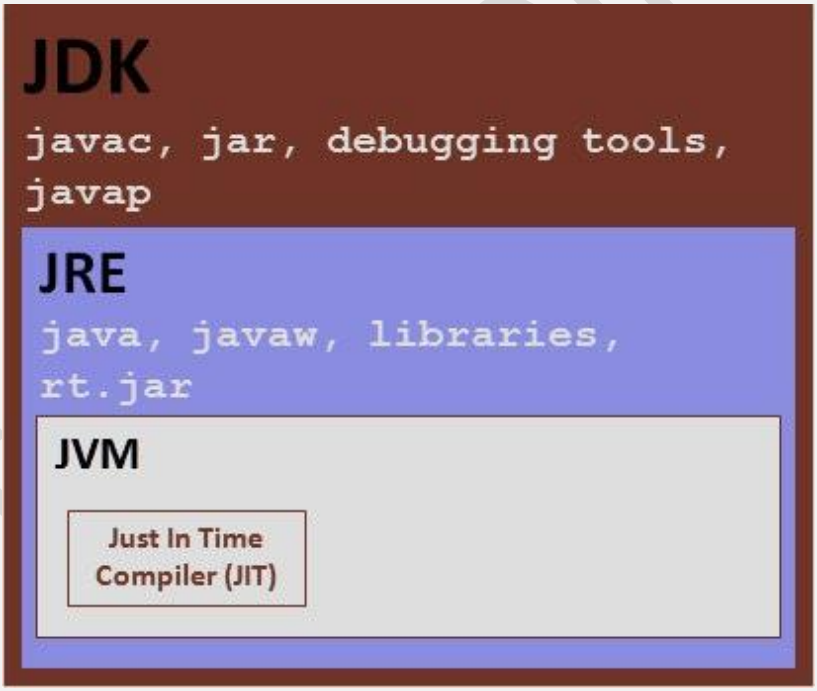
Instructions. The executable instructions are different in different operating systems. So, there are different JVM's for different operating systems. A JVM for windows is different from a JVM for mac. However, both the JVM's understand the bytecode and convert it to the executable code for the respective operating system.

3) What is ByteCode?

Java bytecode is the instruction set of the Java virtual machine. Each bytecode is composed of one, or in some cases two bytes that represent the instruction (opcode), along with zero or more bytes for passing parameters.

4) Compare JDK vs JVM VS JRE.

1. JVM
    a. Virtual machine that run the Java bytecode.
    b. Makes java portable.
2. JRE
    a. JVM + Libraries + Other Components (to run applets and other java applications)
3. JDK
    a. JRE + Compilers + Debuggers



5) What are the important differences between C++ and Java?

1. Java is platform independent. C++ is not platform independent.
2. C++ has pointers (access to internal memory). Java has no concept called pointers.
3. In C++, programmer has to handle memory management. A programmer has to write code to remove an object from memory. In Java, JVM takes care of removing objects from memory using a process called Garbage Collection.
4. C++ supports Multiple Inheritance. Java does not support Multiple Inheritance.

6) What is the role for a ClassLoader in Java?

A Java program is made up of a number of custom classes (written by programmers like us) and core classes (which come pre-packaged with Java). When a program is executed, JVM needs to load the content of all the needed class. JVM uses a ClassLoader to find the classes.

Three Class Loaders are shown in the picture

- System Class Loader - Loads all classes from CLASSPATH
- Extension Class Loader - Loads all classes from extension directory
- Bootstrap Class Loader - Loads all the Java core files

7) Why do we need Wrapper Classes in Java?

A wrapper class wraps (encloses) around a data type and gives it an object appearance.

Reasons why we need Wrapper Classes

- null is a possible value
- use it in a Collection

8) What are the different ways of creating Wrapper Class Instances?

Using a Wrapper Class Constructor

```
Integer number = new Integer(55);//int
Integer number2 = new Integer("55");//String
```

valueOf Static Methods

Provide another way of creating a Wrapper Object

```
Integer hundred =
    Integer.valueOf("100");//100 is stored in variable
```

9) What are differences in the two ways of creating Wrapper Classes?

The difference is that using the Constructor you will always create a new object, while using valueOf() static method, it may return you a cached value with-in a range.

For example : The cached values for long are between [-128 to 127].

We should prefer static valueOf method, because it may save you some memory. To understand it further, here is an implementation of valueOf method in the Long class

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

10) What is Auto Boxing?

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

Example 1

```
Integer nineC = 9;
```

Example 2

```
Integer ten = new Integer(10);
ten++;//allowed. Java does had work behind the screen for us
```

11) What are the advantages of Auto Boxing?

Auto Boxing helps in saving memory by reusing already created Wrapper objects. Auto Boxing uses the static valueOf methods. However wrapper classes created using new are not reused.

12) What is Casting?

Casting is used when we want to convert on data type to another.
There are two types of Casting

- Implicit Casting
- Explicit Casting

13) What is Implicit Casting?

Implicit Casting is done by the compiler. Good examples of implicit casting are all the automatic widening conversions i.e. storing smaller values in larger variable types.

```
int value = 100;
long number = value; //Implicit Casting
float f = 100; //Implicit Casting
```

14) What is Explicit Casting?

Explicit Casting is done through code. Good examples of explicit casting are the narrowing conversions. Storing larger values into smaller variable types;

```
long number1 = 25678;
int number2 = (int)number1;//Explicit Casting
//int x = 35.35;//COMPILER ERROR
int x = (int)35.35;//Explicit Casting
```

Explicit casting would cause truncation of value if the value stored is greater than the size of the variable.

```
int bigValue = 280;
byte small = (byte) bigValue;
System.out.println(small);//output 24. Only 8 bits remain.
```

15) Are all String's immutable?

Value of a String Object once created cannot be modified. Any modification on a String object creates a new String object.

```
String str3 = "value1";
str3.concat("value2");
System.out.println(str3); //value1
```

Note that the value of str3 is not modified in the above example. The result should be assigned to a new reference variable (or same variable can be reused). All wrapper class instances are immutable too!

```
String concat = str3.concat("value2");
System.out.println(concat); //value1value2
```

16) Where are string values stored in memory?

The location where the string values are stored in memory depends on how we create them.

Approach 1

In the example below we are directly referencing a String literal.

```
String str1 = "value";
```

This value will be stored in a "String constant pool" – which is inside the Heap memory. If compiler finds a String literal,it checks if it exists in the pool. If it exists, it is reused.

```
String str5 = "value";
```

In above example, when str5 is created - the existing value from String Constant Pool is reused.

Approach 2

However, if new operator is used to create string object, the new object is created on the heap. There will not be any reuse of values.

```
//String Object - created on the heap
String str2 = new String("value");
```

17) What are differences between String and StringBuffer?

- Objects of type String are immutable. StringBuffer is used to represent values that can be modified.
- In situations where values are modified a number to times, StringBuffer yields significant performance benefits.

18) What are differences between StringBuilder and StringBuffer?

StringBuilder is not thread safe. So, it performs better in situations where thread safety is not required

19) What is a Class?

A class is a Template. In below example, Class CricketScorer is the template for creating multiple objects. A class defines state and behavior that an object can exhibit

```
public class CricketScorer {
    //Instance Variables - constitute the state of an object
    private int score;

    //Behavior - all the methods that are part of the class
    //An object of this type has behavior based on the
    //methods four, six and getScore
    public void four(){
        score = score + 4;
    }

    public void six(){
        score = score + 6;
    }

    public int getScore() {
        return score;
    }

    public static void main(String[] args) {
        CricketScorer scorer = new CricketScorer();
        scorer.six();
        //State of scorer is (score => 6)
        scorer.four();
        //State of scorer is (score => 10)
        System.out.println(scorer.getScore());
    }
}
```

20) What is an Object?

An instance of a class. In the above example, we create an object using `new CricketScorer()`. The reference of the created object is stored in scorer variable. We can create multiple objects of the same class.

21) What is state of an Object?

Values assigned to instance variables of an object. Consider following code snippets from the above example. The value in score variable is initially 0. It changes to 6 and then 10. State of an object might change with time.

22) What is behavior of an Object?

Methods supported by an object. Above example the behavior supported is six(), four() and getScore().

23) What is the super class of every class in Java?

Every class in java is a sub class of the class Object.

24) Explain about toString method?

toString() method is used to print the content of an Object. If the toString method is not overridden in a class, the default toString method from Object class is invoked. This would print some hashcode as shown in the example below. However, if toString method is overridden, the content returned by the toString method is printed.

Consider the class given below:

```
class Animal {

    public Animal(String name, String type) {
        this.name = name;
        this.type = type;
    }

    String name;
    String type;

}
```

Run this piece of code:

```
Animal animal = new Animal("Tommy", "Dog");
System.out.println(animal);//com.rithus.Animal@f7e6a96
```

Output does NOT show the content of animal (what name? and what type?). To show the content of the animal object, we can override the default implementation of toString method provided by Object class.

Adding toString to Animal class

```
class Animal {

    public Animal(String name, String type) {
        this.name = name;
        this.type = type;
    }

    String name;
    String type;

    public String toString() {
        return "Animal [name=" + name + ", type=" + type
                + "]";
    }

}
```

Run this piece of code:

```
Animal animal = new Animal("Tommy","Dog");
System.out.println(animal);//Animal [name=Tommy, type=Dog]
```

Output now shows the content of the animal object

25) What is the use of equals method in Java?

Equals method is used when we compare two objects. Default implementation of equals method is defined in Object class. The implementation is similar to == operator. Two object references are equal only if they are pointing to the same object.

We need to override equals method, if we would want to compare the contents of an object.

Consider the example Client class provided below.

```
class Client {

    private int id;

    public Client(int id) {
        this.id = id;
    }
}
```

== comparison operator checks if the object references are pointing to the same object. It does NOT look at the content of the object.

```
Client client1 = new Client(25);
Client client2 = new Client(25);
Client client3 = client1;

//client1 and client2 are pointing to different client objects.
System.out.println(client1 == client2);//false
```

```
//client3 and client1 refer to the same client objects.
System.out.println(client1 == client3);//true

//similar output to ==
System.out.println(client1.equals(client2));//false
System.out.println(client1.equals(client3));//true
```

We can override the equals method in the Client class to check the content of the objects. Consider the example below: The implementation of equals method checks if the id's of both objects are equal. If so, it returns true. Note that this is a basic implementation of equals and more needs to be done to make it fool-proof.

```
class Client {

    private int id;

    public Client(int id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object obj) {
        Client other = (Client) obj;
        if (id != other.id)
   return false;return true;
    }
}
```

Consider running the code below:

```
Client client1 = new Client(25);
Client client2 = new Client(25);
Client client3 = client1;

//both id's are 25
System.out.println(client1.equals(client2));//true

//both id's are 25
System.out.println(client1.equals(client3));//true
```

Above code compares the values (id's) of the objects.

26) What is the hashCode method used for in Java?

HashCode's are used in hashing to decide which group (or bucket) an object should be placed into. A group of object's might share the same hashcode.

The implementation of hash code decides effectiveness of Hashing. A good hashing function evenly distributes object's into different groups (or buckets).

27) What is Method Overloading?

A method having the same name as another method (in same class or a sub class) but having different parameters is called an Overloaded Method.

Example 1

doIt method is overloaded in the below example:

```
class Foo{
    public void doIt(int number){

    }
    public void doIt(String string){

    }
}
```

Example 2

Overloading can also be done from a sub class.

```
class Bar extends Foo{
    public void doIt(float number){

    }
}
```

28) What is Method Overriding?

Creating a Sub Class Method with same signature as that of a method in SuperClass is called Method Overriding.

Let's define an Animal class with a method shout.

```
public class Animal {
    public String bark() {
        return "Don't Know!";
    }
}
```

Let's create a sub class of Animal – Cat - overriding the existing shout method in Animal.

```
class Cat extends Animal {
    public String bark() {
        return "Meow Meow";
    }
}
```

bark method in Cat class is overriding the bark method in Animal class.

Java Example : HashMap public int size() overrides AbstractMap public int size()

29) Can super class reference variable can hold an object of sub class?

Yes. Look at the example below:

```
Object object = new Hero();

public class Actor {
    public void act(){
        System.out.println("Act");
    };
}

//IS-A relationship. Hero is-a Actor
public class Hero extends Actor {
    public void fight(){
        System.out.println("fight");
    };
}

//IS-A relationship. Comedian is-a Actor
public class Comedian extends Actor {
    public void performComedy(){
        System.out.println("Comedy");
    };
}

Actor actor1 = new Comedian();
Actor actor2 = new Hero();
```

30) Is Multiple Inheritance allowed in Java?

Multiple Inheritance results in a number of complexities. Java does not support Multiple Inheritance.

```
class Dog extends Animal, Pet { //COMPILER ERROR
}
```

However, we can create an Inheritance Chain

```
class Pet extends Animal {
}

class Dog extends Pet {
}
```

31) What is an Interface?

An interface defines a contract for responsibilities (methods) of a class.

32) How do you define an Interface?

An interface is declared by using the keyword interface. Look at the example below: Flyable is an interface.

```
//public abstract are not necessary
public abstract interface Flyable {
    //public abstract are not necessary
    public abstract void fly();
}
```

33) How do you implement an interface?

We can define a class implementing the interface by using the implements keyword. Let us look at a couple of examples:

Example 1

Class Aeroplane implements Flyable and implements the abstract method fly().

```
public class Aeroplane implements Flyable{
    @Override
    public void fly() {
        System.out.println("Aeroplane is flying");
    }
}
```

Example 2

```
public class Bird implements Flyable{
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}
```

Note : - Variables in an interface are always public, static, final. Variables in an interface cannot be declaredprivate.

```
interface ExampleInterface1 {
    //By default - public static final. No other modifier allowed
    //value1,value2,value3,value4 all are - public static final
    int value1 = 10;
    public int value2 = 15;
    public static int value3 = 20;
    public static final int value4 = 25;
    //private int value5 = 10;//COMPILER ERROR
}
```

Interface methods are by default public and abstract. Before Java 8, A concrete method (fully defined method) cannot be created in an interface. Consider the example below:

```
interface ExampleInterface1 {
    //By default - public abstract. No other modifier allowed
    void method1();//method1 is public and abstract
    //private void method6();//COMPILER ERROR!

//This method, uncommented, would have given COMPILER ERROR!

    //in Java 7. Allowed from Java 8.
    default void method5() {
        System.out.println("Method5");
```

```
        }
    }
```

34) Can you extend an interface?

An interface can extend another interface. Consider the example below:

```
interface SubInterface1 extends ExampleInterface1{
    void method3();
}
```

35) Can a class extend multiple interfaces?

A class can implement multiple interfaces. It should implement all the method declared in all Interfaces being implemented.

36) What is an Abstract Class?

An abstract class is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby encapsulating common functionality for inherited classes.

```
public abstract class AbstractClassExample {
    public static void main(String[] args) {
        //An abstract class cannot be instantiated
        //Below line gives compilation error if uncommented
        //AbstractClassExample ex = new AbstractClassExample();
    }
}
```

37) When do you use an Abstract Class?

If you want to provide common, implemented functionality among all implementations of your component, use an abstract class.

38) How do you define an abstract method?

An Abstract method does not contain body. An abstract method does not have any implementation. The implementation of an abstract method should be provided in an over-riding method in a sub class.

```
    //Abstract Class can contain 0 or more abstract methods
    //Abstract method does not have a body
    abstract void abstractMethod1();
    abstract void abstractMethod2();
```

Abstract method can be declared only in Abstract Class. In the example below, abstractMethod() gives a compiler error because NormalClass is not abstract.

```
class NormalClass{
    abstract void abstractMethod();//COMPILER ERROR
}
```

39) Compare Abstract Class vs Interface?

Syntactical Differences

- Methods and members of an abstract class can have any visibility. All methods of an interface must be public.
- A concrete child class of an Abstract Class must define all the abstract methods. An Abstract child class can have abstract methods. An interface extending another interface need not provide default implementation for methods inherited from the parent interface.
- A child class can only extend a single class. An interface can extend multiple interfaces. A class can implement multiple interfaces.
- A child class can define abstract methods with the same or less restrictive visibility, whereas a class implementing an interface must define all interface methods as public.

40) What is a Constructor?

Constructor is invoked whenever we create an instance(object) of a Class. We cannot create an object without a constructor.

```
class Animal {
    String name;

    // This is called a one argument constructor.
    public Animal(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        // Since we provided a constructor, compiler does not
        // provide a default constructor.
        // Animal animal = new Animal();//COMPILER ERROR!

        // The only way we can create Animal1 object is by using
        Animal animal = new Animal("Tommy");
    }
}
```

41) What is a Default Constructor?

Default Constructor is the constructor that is provided by the compiler. It has no arguments. In the example below, there are no Constructors defined in the Animal class. Compiler provides us with a default constructor, which helps us create an instance of animal class.

```
public class Animal {
    String name;

    public static void main(String[] args) {
        // Compiler provides this class with a default no-argument constructor.
        // This allows us to create an instance of Animal class.
        Animal animal = new Animal();
    }
}
```

42) How do you call a Super Class Constructor from a Constructor?

A constructor can call the constructor of a super class using the super() method call. Only constraint is that it should be the first statement

43) What is the use of this()?

Another constructor in the same class can be invoked from a constructor, using this({parameters}) method call.

```
public Animal() {
    this("Default Name");
}

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

44) Is a super class constructor called even when there is no explicit call from a sub class constructor?

If a super class constructor is not explicitly called from a sub class constructor, super class (no argument) constructor is automatically invoked (as first line) from a sub class constructor.

Consider the example below:

```
class Animal {
    public Animal() {
        System.out.println("Animal Constructor");
    }
}

class Dog extends Animal {
    public Dog() {
        System.out.println("Dog Constructor");
    }
}

class Labrador extends Dog {
    public Labrador() {
        System.out.println("Labrador Constructor");
    }
}

public class ConstructorExamples {
    public static void main(String[] args) {
        Labrador labrador = new Labrador();
    }
}
```

Program Output

Animal Constructor
Dog Constructor
Labrador Constructor

45) What is Polymorphism?

Polymorphism is defined as “Same Code” giving “Different Behavior”. Let’s look at an example.

Let’s define an Animal class with a method shout.

```
public class Animal {
    public String shout() {
        return "Don't Know!";
    }
}
```

Let’s create two new sub classes of Animal overriding the existing shout method in Animal.

```
class Cat extends Animal {
    public String shout() {
        return "Meow Meow";
    }
}

class Dog extends Animal {
    public String shout() {
        return "BOW BOW";
    }

    public void run(){

    }
}
```

Look at the code below. An instance of Animal class is created. shout method is called.

```
Animal animal1 = new Animal();
System.out.println(
        animal1.shout()); //Don't Know!
```

Look at the code below. An instance of Dog class is created and store in a reference variable of type Animal.

```
Animal animal2 = new Dog();

//Reference variable type => Animal
//Object referred to => Dog
//Dog's bark method is called.
System.out.println(
        animal2.shout()); //BOW BOW
```

When shout method is called on animal2, it invokes the shout method in Dog class (type of the object pointed to by reference variable animal2).

46) What is the use of instanceof Operator in Java?

instanceof operator checks if an object is of a particular type.

47) What is Coupling?

Coupling is a measure of how much a class is dependent on other classes. There should minimal dependencies between classes. So, we should always aim for low coupling between classes.

48) What is Cohesion?

Cohesion is a measure of how related the responsibilities of a class are. A class must be highly cohesive i.e. its responsibilities (methods) should be highly related to one another.

49) What is Encapsulation?

Encapsulation is “hiding the implementation of a Class behind a well defined interface”.

Approach 1

In this approach we create a public variable score. The main method directly accesses the score variable, updates it.

```
public class CricketScorer {
    public int score
}

Let’s use the CricketScorer class.
public static void main(String[] args) {
       CricketScorer scorer = new CricketScorer();
       scorer.score = scorer.score + 4;
}
```

Approach 2

In this approach, we make score as private and access value through get and set methods. However, the logic of adding 4 to the score is performed in the main method.

```
public class CricketScorer {
    private int score;

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
        this.score = score;
    }
}

Let’s use the CricketScorer class.

public static void main(String[] args) {
       CricketScorer scorer = new CricketScorer();

       int score = scorer.getScore();
       scorer.setScore(score + 4);
}
```

Approach 3

In this approach - For better encapsulation, the logic of doing the four operation also is moved to the CricketScorer class.

```
public class CricketScorer {
    private int score;

    public void four() {
         score += 4;
    }

}
```

```
Let's use the CricketScorer class.
public static void main(String[] args) {
       CricketScorer scorer = new CricketScorer();
       scorer.four();
}
```

Description

In terms of encapsulation Approach 3 > Approach 2 > Approach 1. In Approach 3, the user of scorer class does not even know that there is a variable called score. Implementation of Scorer can change without changing other classes using Scorer.

50) What is an Inner Class?

Inner Classes are classes which are declared inside other classes. Consider the following example:

```
class OuterClass {

    public class InnerClass {
    }

    public static class StaticNestedClass {
    }

}
```

51) What is a Static Inner Class?

A class declared directly inside another class and declared as static. In the example above, class name StaticNestedClass is a static inner class.

Can you create an inner class inside a method?

Yes. An inner class can be declared directly inside a method. In the example below, class name MethodLocalInnerClass is a method inner class.

```
class OuterClass {

    public void exampleMethod() {
        class MethodLocalInnerClass {
        };
    }

}
```

52) What is an Anonymous Class?

Anonymous Class does not have a name. Below examples shows various ways to create Anonymous classes.

```
class Animal {
      void bark() {
            System.out.println("Animal  Bark");
      }
};

public class AnonymousClass {

      private static String[] reverseSort(String[] array) {
```

```
            Comparator<String> reverseComparator = new Comparator<String>() {
                  /* Anonymous Class */
                  @Override
                  public int compare(String string1,
                              String string2) {
                        return string2.compareTo(string1);
                  }

            };

            Arrays.sort(array,  reverseComparator);

            return array;
      }

      public static void main(String[] args) {

            String[] array = { "Apple", "Cat", "Boy" };

            System.out .println(Arrays
                        .toString(reverseSort(array)));//[Cat,  Boy,  Apple]

            /* Second Anonymous Class - SubClass of Animal*/
            Animal animal = new Animal() {
                  void bark() {
                        System.out.println("Subclass  bark");
                  }
            };

            animal.bark();//Subclass bark

      }

}
```

52) What is default class modifier?

- A class is called a Default Class is when there is no access modifier specified on a class.
- Default classes are visible inside the same package only.
- Default access is also called Package access.

53) What is private access modifier?

- Private variables and methods can be accessed only in the class they are declared.
- Private variables and methods from SuperClass are NOT available in SubClass.

54) What is default or package access modifier?

- Default variables and methods can be accessed in the same package Classes.
- Default variables and methods from SuperClass are available only to SubClasses in same package.

55) What is protected access modifier?

- Protected variables and methods can be accessed in the same package Classes.
- Protected variables and methods from SuperClass are available to SubClass in any package

56) What is public access modifier?

- Public variables and methods can be accessed from every other Java classes.
- Public variables and methods from SuperClass are all available directly in the SubClass

57) What is the use of a final modifier on a class?

Final class cannot be extended. Example of Final class in Java is the String class. Final is used very rarely as it prevents re-use of the class.Consider the class below which is declared as final.

Final Class examples : String, Integer, Double and other wrapper classes

```
final public class FinalClass {
}
```

Below class will not compile if uncommented. FinalClass cannot be extended.

```
/*
class ExtendingFinalClass extends FinalClass{ //COMPILER ERROR

}
*/
```

58) What is the use of a final modifier on a method?

Final methods cannot be overridden. Consider the class FinalMemberModifiersExample with method finalMethod which is declared as final.

```
public class FinalMemberModifiersExample {
    final void finalMethod(){
    }
}
```

Any SubClass extending above class cannot override the finalMethod().

```
class SubClass extends FinalMemberModifiersExample {
    //final method cannot be over-riddent
    //Below method, uncommented, causes compilation Error
    /*
    final void finalMethod(){

    }
    */
}
```

59) What is a Final variable?

Once initialized, the value of a final variable cannot be changed.

```
final int finalValue = 5;
 //finalValue = 10; //COMPILER ERROR
```

Final Variable example : java.lang.Math.PI

60) What is a final argument?

Final arguments value cannot be modified. Consider the example below:

```
void testMethod(final int finalArgument){
    //final argument cannot be modified
    //Below line, uncommented, causes compilation Error
    //finalArgument = 5;//COMPILER ERROR
}
```

61) What is a Static Variable?

Static variables and methods are class level variables and methods. There is only one copy of the static variable for the entire Class. Each instance of the Class (object) will NOT have a unique copy of a static variable. Let's start with a real world example of a Class with static variable and methods.

Static Variable/Method – Example

count variable in Cricketer class is static. The method to get the count value getCount() is also a static method.

```
public class Cricketer {
    private static int count;

    public Cricketer() {
        count++;
    }

    static int getCount() {
        return count;
    }

    public static void main(String[] args) {

        Cricketer cricketer1 = new Cricketer();
        Cricketer cricketer2 = new Cricketer();
        Cricketer cricketer3 = new Cricketer();
        Cricketer cricketer4 = new Cricketer();

        System.out.println(Cricketer.getCount());//4
    }
}
```

4 instances of the Cricketer class are created. Variable count is incremented with every instance createdin the constructor.

62) What is an Enhanced For Loop?

Enhanced for loop can be used to loop around array's or List's.

```
int[] numbers = {1,2,3,4,5};

for(int number:numbers){
    System.out.print(number);
}
```

Example Output

```
12345
```

What is the output of the for loop below?

Any of 3 parts in a for loop can be empty.

```
for (;;) {
    System.out.print("I will be looping for ever..");
}
```

Result:

```
Infinite loop => Loop executes until the program is terminated.
```

63) Why is Exception Handling important?

Most applications are large and complex. I've not seen an application without defects in my 15 year experience. It is not that bad programmers create defects. Even good programmers write code that has defects and throws exceptions. There are two things that are important when exceptions are thrown.

- A friendly message to the user: You do not want a windows blue screen. When something goes wrong and an exception occurs, it would be great to let the user know that something went wrong and tech support has been notified. Additional thing we can do is to give the user a unique exception identifier and information on how to reach the tech support.
- Enough Information for the Support Team/Support Developer to debug the problem : When writing code, always think about what information would I need to debug a problem in this piece of code. Make sure that information is made available, mostly in the logs, if there are exceptions. It would be great to tie the information with the unique exception identifier given to the user.

64) In what scenarios is code in finally not executed?

- Code in finally is NOT executed only in two situations.If exception is thrown in finally.
- If JVM Crashes in between (for example, System.exit()).

65) Is try without a catch is allowed?

Yes. It is.

66) Is try without catch and finally allowed?

No.

67) Can you explain about try with resources?

Consider the example below. When the try block ends the resources are automatically released. We do not need to create a separate finally block.

```
try (BufferedReader br = new BufferedReader(new FileReader("FILE_PATH")))
{

	String line;
	while ((line = br.readLine()) != null) {
		System.out.println(line);
	}
} catch (IOException e) {
	e.printStackTrace();
}
```

68) How does try with resources work?

try-with-resources is available to any class that implements the AutoCloseable interface. In the above example BufferedReader implements AutoCloseable interface.

```
public interface AutoCloseable { void
	close() throws Exception;
}
```

69) How do you print the content of an array?

Let's look at different methods in java to print the content of an array.

Printing a 1D Array

```
int marks5[] = { 25, 30, 50, 10, 5 };
System.out.println(marks5); //[I@6db3f829
System.out.println(
    Arrays.toString(marks5));//[25, 30, 50, 10, 5]
```

Printing a 2D Array

```
int[][] matrix3 = { { 1, 2, 3 }, { 4, 5, 6 } };
System.out.println(matrix3); //[[I@1d5a0305
System.out.println(
        Arrays.toString(matrix3));
//[[I@6db3f829, [I@42698403]
System.out.println(
        Arrays.deepToString(matrix3));
```

70) What are Variable Arguments or varargs?

Variable Arguments allow calling a method with different number of parameters. Consider the example method sum below. This sum method can be called with 1 int parameter or 2 int parameters or more int parameters.

```
    //int(type) followed ... (three dot's) is syntax of a variable argument.
    public int sum(int... numbers) {
        //inside the method a variable argument is similar to an array.
```

```
        //number can be treated as if it is declared as int[] numbers;
        int sum = 0;
        for (int number: numbers) {
            sum += number;
        }
        return sum;
    }


    public static void main(String[] args) {
        VariableArgumentExamples example = new VariableArgumentExamples();
        //3 Arguments
        System.out.println(example.sum(1, 4, 5));//10
        //4 Arguments
        System.out.println(example.sum(1, 4, 5, 20));//30
        //0 Arguments
        System.out.println(example.sum());//0
    }
```

71) What is Garbage Collection?

Garbage Collection is a name given to automatic memory management in Java. Aim of Garbage Collection is to Keep as much of heap available (free) for the program as possible. JVM removes objects on the heap which no longer have references from the heap

72) When is Garbage Collection run?

Garbage Collection runs at the whims and fancies of the JVM (it isn't as bad as that). Possible situations when Garbage Collection might run are

- when available memory on the heap is low
- when cpu is free

73) What are best practices on Garbage Collection?

Programmatically, we can request (remember it's just a request - Not an order) JVM to run Garbage Collection by calling System.gc() method.

JVM might throw an OutOfMemoryException when memory is full and no objects on the heap are eligible for garbage collection.

finalize() method on the objected is run before the object is removed from the heap from the garbage collector. We recommend not to write any code in finalize();

74) What are Initialization Blocks?

Initialization Blocks - Code which runs when an object is created or a class is loaded

There are two types of Initialization Blocks
Static Initializer: Code that runs when a class is loaded.
Instance Initializer: Code that runs when a new object is created.

75) What is a Static Initializer?

Look at the example below:

```
public class InitializerExamples {
static int count;
int i;
```

```
static{
    //This is a static initializers. Run only when Class is first loaded.
    //Only static variables can be accessed
    System.out.println("Static Initializer");
    //i = 6;//COMPILER ERROR
    System.out.println("Count when Static Initializer is run is " + count);
}

public static void main(String[] args) {
    InitializerExamples example = new InitializerExamples();
       InitializerExamples example2 = new InitializerExamples();
       InitializerExamples example3 = new InitializerExamples();
}
}
```

Code within static{ and } is called a static initializer. This is run only when class is first loaded. Only static variables can be accessed in a static initializer.

Example Output

```
Static Initializer
Count when Static Initializer is run is 0
```

Even though three instances are created static initializer is run only once.

76) What is an Instance Initializer Block?

Let's look at an example

```
public class InitializerExamples {
    static int count;

    int i;
    {
        //This is an instance initializers. Run every time an object is created.
        //static and instance variables can be accessed
        System.out.println("Instance Initializer");
        i = 6;
        count = count + 1;
        System.out.println("Count when Instance Initializer is run is " + count);
    }

    public static void main(String[] args) {
        InitializerExamples example = new InitializerExamples();
        InitializerExamples example1 = new InitializerExamples();
        InitializerExamples example2 = new InitializerExamples();
    }

}
```

Code within instance initializer is run every time an instance of the class is created.

Example Output

```
Instance Initializer
Count when Instance Initializer is run is 1
Instance Initializer
Count when Instance Initializer is run is 2
Instance Initializer
Count when Instance Initializer is run is 3
```

77) What is Tokenizing?

Tokenizing means splitting a string into several sub strings based on delimiters. For example, delimiter ; splits the string ac;bd;def;e into four sub strings ac, bd, def and e.

Delimiter can in itself be any of the regular expression(s) we looked at earlier.

String.split(regex) function takes regex as an argument.

78) Can you give an example of Tokenizing?

```
private static void tokenize(String string,String regex) {
    String[] tokens = string.split(regex);
    System.out.println(Arrays.toString(tokens));
}
```

Example:

```
tokenize("ac;bd;def;e",";");//[ac, bd, def, e]
```

79) What is Serialization?

Serialization helps us to save and retrieve the state of an object.

- Serialization => Convert object state to some internal object representation.
- De-Serialization => The reverse. Convert internal representation to object.

Two important methods:

- ObjectOutputStream.writeObject() // serialize and write to file
- ObjectInputStream.readObject() // read from file and deserialize

80) How do you serialize an object using Serializable interface?

To serialize an object it should implement Serializable interface. In the example below, Rectangle class implements Serializable interface. Note that Serializable interface does not declare any methods to be implemented.

Below example shows how an instance of an object can be serialized. We are creating a new Rectangle object and serializing it to a file Rectangle.ser.

```
class Rectangle implements Serializable {
    public Rectangle(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
        area = length * breadth;
    }

    int length;
    int breadth;
    int area;
}

FileOutputStream fileStream = new FileOutputStream("Rectangle.ser");
ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);
objectStream.writeObject(new Rectangle(5, 6));
objectStream.close();
```

81) How do you de-serialize in Java?

Below example show how a object can be deserialized from a serialized file. A rectangle object is deserialized from the file Rectangle.ser

```
FileInputStream fileInputStream = new FileInputStream("Rectangle.ser");
ObjectInputStream objectInputStream = new ObjectInputStream(
        fileInputStream);
Rectangle rectangle = (Rectangle) objectInputStream.readObject();
objectInputStream.close();
System.out.println(rectangle.length);// 5
System.out.println(rectangle.breadth);// 6
System.out.println(rectangle.area);// 30
```

82) What do you do if only parts of the object have to be serialized?

We mark all the properties of the object which should not be serialized as transient. Transient attributes in an object are not serialized. Area in the previous example is a calculated value. It is unnecessary to serialize and deserialize. We can calculate it when needed. In this situation, we can make the variable transient. Transient variables are not serialized. (`transient int area;`)

//Modified Rectangle class

```
class Rectangle implements Serializable {
    public Rectangle(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
        area = length * breadth;
    }

    int length;
    int breadth;
    transient int area;
}
```

If you run the program again, you would get following output

```
System.out.println(rectangle.length);// 5
System.out.println(rectangle.breadth);// 6
System.out.println(rectangle.area);// 0
```

Note that the value of rectangle.area is set to 0. Variable area is marked transient. So, it is not stored into the serialized file. And when de-serialization happens area value is set to default value i.e. 0.

83) How do you serialize a hierarchy of objects?

Objects of one class might contain objects of other classes. When serializing and de-serializing, we might need to serialize and de-serialize entire object chain. All classes that need to be serialized have to implement the Serializable interface. Otherwise, an exception is thrown. Look at the class below. An object of class House contains an object of class Wall.

```
class House implements Serializable {
    public House(int number) {
        super();
        this.number = number;
    }
```

```
    Wall wall;
    int number;
}

class Wall{
    int length;
    int breadth;
    int color;
}
```

House implements Serializable. However, Wall doesn't implement Serializable. When we try to serialize an instance of House class, we get the following exception.

Output:

```
Exception in thread "main" java.io.NotSerializableException:
com.rithus.serialization.Wall
    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
    at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)
```

This is because Wall is not serializable. Two solutions are possible.

1. Make Wall transient. Wall object will not be serialized. This causes the wall object state to be lost.
2. Make Wall implement Serializable. Wall object will also be serialized and the state of wall object along with the house will be stored.

```
class House implements Serializable {

    public House(int number) {
        super();
        this.number = number;
    }

    transient Wall wall;
    int number;
}

class Wall implements Serializable {
    int length;
    int breadth;
    int color;
}
```

With both these programs, earlier main method would run without throwing an exception.

If you try de-serializing, In Example2, state of wall object is retained whereas in Example1, state of wall object is lost.

84) Are the constructors in an object invoked when it is de-serialized?

No. When a class is De-serialized, initialization (constructor's, initializer's) does not take place. The state of the object is retained as it is.

85) Are the values of static variables stored when an object is serialized?

Static Variables are not part of the object. They are not serialized.