www.in28minutes.com

Java Interview Questions and Answers

Java Interview Companion - Books & Videos

Copyright © 2016 by in28Minutes

Lead Editor: Ranga Karanam

For information on translations, please contact Ranga at http://www.in28minutes.com

The source code for this book is available to readers at https://github.com/in28minutes/JavaInterviewQuestionsAndAnswers/

TABLE OF CONTENTS

Java Platform

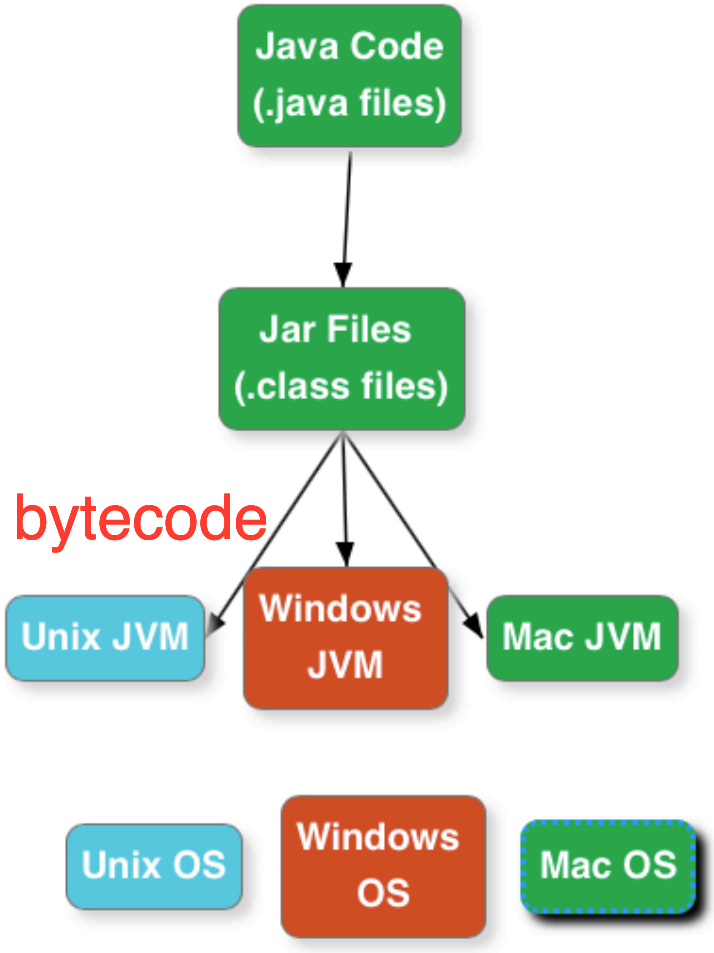
Why is Java so Popular?

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.

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.

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.

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

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

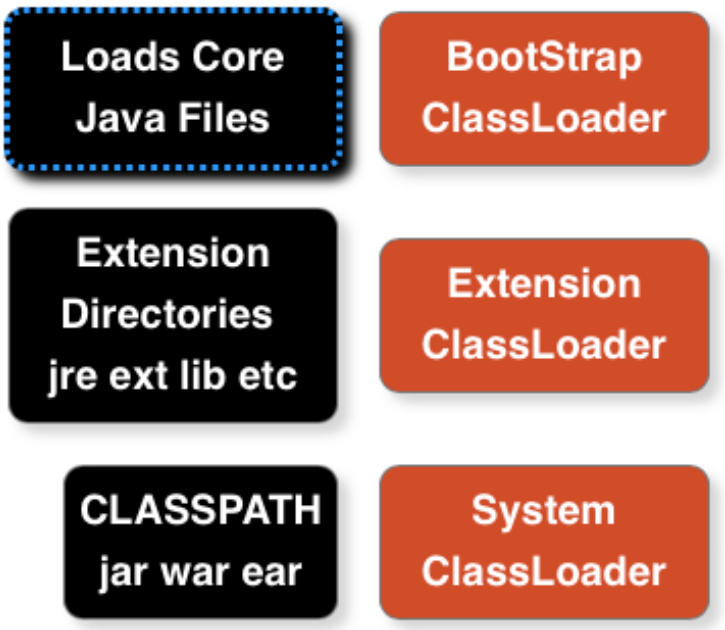
1. Java is platform independent. C++ is not platform independent.
2. Java & C++ are both NOT pure Object Oriented Languages. However, Java is more purer Object Oriented Language (except for primitive variables). In C++, one can write structural programs without using objects.
3. C++ has pointers (access to internal memory). Java has no concept called pointers.
4. 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.
5. C++ supports Multiple Inheritance. Java does not support Multiple Inheritance.

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

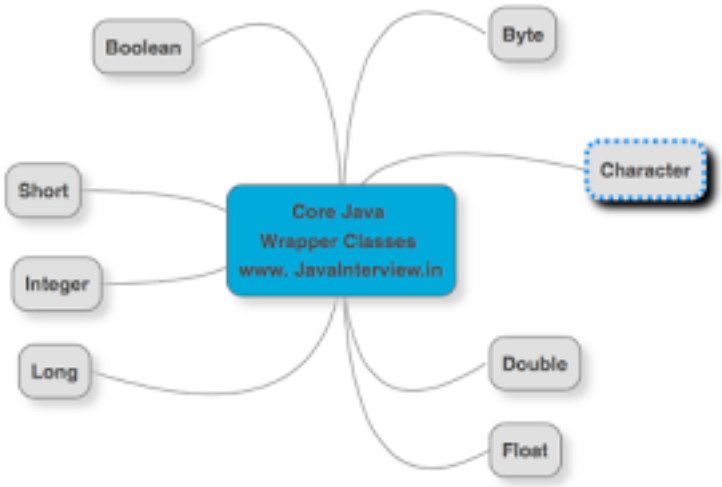


When JVM needs to find a class, it starts with System Class Loader. If it is not found, it checks with Extension Class Loader. If it not found, it goes to the Bootstrap Class Loader. If a class is still not found, a ClassNotFoundException is thrown.

Wrapper Classes

What are wrapper classes?

A brief description is provided below.



A primitive wrapper class in the Java programming language is one of eight classes provided in the java.lang package to provide object methods for the eight primitive types. All of the primitive wrapper classes in Java are immutable.

Wrapper: Boolean,Byte,Character,Double,Float,Integer,Long,Short
Primitive: boolean,byte,char ,double, float, int , long,short

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
- Methods that support Object like creation from other types.. like String
    - `Integer number2 = new Integer("55");//String`

What are the different ways of creating Wrapper Class Instances?

Two ways of creating Wrapper Class Instances are described below.

Using a Wrapper Class Constructor

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

Float number3 = new Float(55.0);//double argument
Float number4 = new Float(55.0f);//float argument
Float number5 = new Float("55.0f");//String
```

```
Character c1 = new Character('C');//Only char constructor
//Character c2 = new Character(124);//COMPILER ERROR

Boolean b = new Boolean(true);

//"true" "True" "tRUe" - all String Values give True
//Anything else gives false
Boolean b1 = new Boolean("true");//value stored - true
Boolean b2 = new Boolean("True");//value stored - true
Boolean b3 = new Boolean("False");//value stored - false
Boolean b4 = new Boolean("SomeString");//value stored - false
```

valueOf Static Methods

Provide another way of creating a Wrapper Object

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

Integer seven =
    Integer.valueOf("111", 2);//binary 111 is converted to 7
```

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

```
/**
 * Returns an {@code Integer} instance representing the specified
 * {@code int} value.  If a new {@code Integer} instance is not
 * required, this method should generally be used in preference to
 * the constructor {@link #Integer(int)}, as this method is likely
 * to yield significantly better space and time performance by
 * caching frequently requested values.
 *
 * This method will always cache values in the range -128 to 127,
 * inclusive, and may cache other values outside of this range.
 *
 * @param  i an {@code int} value.
 * @return an {@code Integer} instance representing {@code i}.
 * @since  1.5
 */
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
```

```
        return new Integer(i);
    }
```

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
```

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.

Two wrapper objects created using new are not same object.

```
Integer nineA = new Integer(9);
Integer nineB = new Integer(9);
System.out.println(nineA == nineB);//false
System.out.println(nineA.equals(nineB));//true
```

Two wrapper objects created using boxing are same object.

```
Integer nineC = 9;
Integer nineD = 9;
System.out.println(nineC == nineD);//true
System.out.println(nineC.equals(nineD));//true
```

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

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
```

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.
```

Strings

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
```

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");
```

Why should you be careful about String Concatenation(+) operator in Loops?

Consider the code below:

```
String s3 = "Value1";
String s2 = "Value2";
for (int i = 0; i < 100000; ++i) {
      s3 = s3 + s2;
}
```

How many objects are created in memory? More than 100000 Strings are created. This will have a huge performance impact.

How do you solve above problem?

The easiest way to solve above problem is using StringBuffer. On my machine StringBuffer version took 0.5 seconds. String version took 25 Seconds. That's a 50 fold increase in performance.

```
StringBuffer s3 = new StringBuffer("Value1");
String s2 = "Value2";
for (int i = 0; i < 100000; ++i) {
      s3.append(s2);
}
```

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.
- Both String and StringBuffer are thread-safe.
- StringBuffer is implemented by using synchronized keyword on all methods.

What are differences between StringBuilder and StringBuffer?

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

Can you give examples of different utility methods in String class?

String class defines a number of methods to get information about the string content.

```
String str = "abcdefghijk";
```

Get information from String

```
Following methods help to get information from a String.
//char charAt(int paramInt)
System.out.println(str.charAt(2)); //prints a char - c
System.out.println("ABCDEFGH".length());//8
System.out.println("abcdefghij".toString()); //abcdefghij
System.out.println("ABC".equalsIgnoreCase("abc"));//true

//Get All characters from index paramInt
//String substring(int paramInt)
```

```
System.out.println("abcdefghij".substring(3)); //cdefghij

//All characters from index 3 to 6
System.out.println("abcdefghij".substring(3,7)); //defg
```

Object Oriented Programming Basics

What is a Class?

Let's look at an example:

```
package com.rithus;

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());
    }
}
```

Class

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

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.

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.

```
        scorer.six();
        //State of scorer is (score => 6)

        scorer.four();
        //State of scorer is (score => 10)
```

What is behavior of an Object?

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

What is the super class of every class in Java?

Every class in java is a sub class of the class Object. When we create a class we inherit all the methods and properties of Object class. Let's look at a simple example:

```
String str = "Testing";
System.out.println(str.toString());
System.out.println(str.hashCode());
System.out.println(str.clone());

if(str instanceof Object){
    System.out.println("I extend Object");//Will be printed
}
```

In the above example, toString, hashCode and clone methods for String class are inherited from Object class and overridden.

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.

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.

What are the important things to consider when implementing equals method?

Any equals implementation should satisfy these properties:

1. Reflexive. For any reference value x, x.equals(x) returns true.
2. Symmetric. For any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
3. Transitive. For any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
4. Consistent. For any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, if no information used in equals is modified.
5. For any non-null reference value x, x.equals(null) should return false.

Our earlier implementation of equals method will not satisfy condition 5. It would throw an exception if an object of different class (other than Client) is used for comparison.

Let's now provide an implementation of equals which satisfy these properties:

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

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).

A good hashCode method should have the following properties

- If obj1.equals(obj2) is true, then obj1.hashCode() should be equal to obj2.hashCode()
- obj.hashCode() should return the same value when run multiple times, if values of obj used in equals() have not changed.
- If obj1.equals(obj2) is false, it is NOT required that obj1.hashCode() is not equal to obj2.hashCode(). Two unequal objects MIGHT have the same hashCode.

A sample hashcode implementation of Client class which meets above constraints is given below:

```
//Client class
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}
```

Explain inheritance with Examples.

Consider the example class Actor below:

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

```
}
```

We can extend this class by using the keyword extends. Hero class extends Actor.

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

We can now create an instance of Hero class. Since Hero extends Animal, the methods defined in Animal are also available through an instance of Hero class. In the example below, we invoke the act method on hero object.

```
Hero hero = new Hero();
//act method inherited from Actor
hero.act();//Act
hero.fight();//fight
```

Let's look at another class extending Actor class - Comedian.

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

We can now reuse Actor methods from an instance of Comedian class as well.

```
Comedian comedian = new Comedian();
//act method inherited from Actor
comedian.act();//Act
comedian.performComedy();//Comedy
```

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){

    }
}
```

Java Example

- Constructors
  - public HashMap(int initialCapacity, float loadFactor)
  - public HashMap() {
  - public HashMap(int initialCapacity)
- Methods
  - public boolean addAll(Collection<? extends E> c)
  - public boolean addAll(int index, Collection<? extends E> c)

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()

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

Yes. Look at the example below:

Actor reference variables actor1, actor2 hold the reference of objects of sub classes of Animal, Comedian and Hero.

Since object is super class of all classes, an Object reference variable can also hold an instance of any class.

```
//Object is super class of all java classes
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();
```

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 {
}
```

What is an Interface?

- An interface defines a contract for responsibilities (methods) of a class.
- An interface is a contract: the guy writing the interface says, "hey, I accept things looking that way"
- Interface represents common actions between Multiple Classes.
- **Example in Java api** : Map interface, Collection interface.

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();
}
```

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");
    }
}
```

Can you explain a few tricky things about interfaces?

Variables in an interface are always public, static, final. Variables in an interface cannot be declared private.

```
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");
        }
    }
```

Can you extend an interface?

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

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

Class implementing SubInterface1 should implement both methods - method3 and method1(from ExampleInterface1)

An interface cannot extend a class.

```
/* //COMPILE ERROR IF UnCommented
   //Interface cannot extend a Class
interface SubInterface2 extends Integer{
    void method3();
}
*/
```

Can a class extend multiple interfaces?

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

An example of a class in the JDK that implements several interfaces is HashMap, which implements the interfaces Serializable, Cloneable, and Map. By reading this list of interfaces, you can infer that an instance of HashMap (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map.

```
interface ExampleInterface2 {
    void method2();
}

class SampleImpl implements ExampleInterface1,ExampleInterface2{
    /* A class should implement all the methods in an interface.
       If either of method1 or method2 is commented, it would
       result in compilation error.
     */
    public void method2() {
        System.out.println("Sample Implementation for Method2");
    }

    public void method1() {
        System.out.println("Sample Implementation for Method1");
```

```
    }

}
```

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();
    }
}
```

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. Abstract classes allow you to partially implement your class.

- An example of an abstract class in the JDK is AbstractMap, which is part of the Collections Framework. Its subclasses (which include HashMap, TreeMap, and ConcurrentHashMap) share many methods (including get, put, isEmpty, containsKey, and containsValue) that AbstractMap defines.
    - example abstract method : public abstract Set> entrySet();

- Another Example - Spring AbstractController

In code below "AbstractClassExample ex = new AbstractClassExample();" gives a compilation error because AbstractClassExample is declared with keyword abstract.

Example in Java : HashMap & TreeMap extend AbstractMap.

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
}
```

Compare Abstract Class vs Interface?

Real Difference - Apple vs Orange

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

What is a Constructor?

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

Constructor has the same name as the class and no return type. It can accept any number of parameters.

```
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");
    }
}
```

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();
    }
}
```

Will this code compile?

```
class Animal {
    String name;

    public Animal() {
        this.name = "Default Name";
    }

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

    public static void main(String[] args) {
       Animal animal = new Animal();
    }
}
```

Answer is no. Since we provided a constructor, compiler does not provide a default constructor

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 i

Both example constructors below can replaces the no argument "public Animal() " constructor in Example 3.

```
public Animal() {
    super();
    this.name = "Default Name";
}
```

Will this code Compile?

```
public Animal() {
    this.name = "Default Name";
    super();
}
```

Answer is NO. super should be always called on the first line of the constructor.

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;
}
```

Can a constructor be called directly from a method?

A constructor cannot be explicitly called from any method except another constructor.

```
class Animal {
    String name;

    public Animal() {
    }

    public method() {
        Animal();// Compiler error
    }
}
```

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
```

Advanced Object Oriented Concepts

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).

Even though dog has a method run, it cannot be invoked using super class reference variable.

```
//animal2.run();//COMPILE ERROR
```

What is the use of instanceof Operator in Java?

instanceof operator checks if an object is of a particular type. Let us consider the following class and interface declarations:

```
class SuperClass {
};

class SubClass extends SuperClass {
};

interface Interface {
};

class SuperClassImplementingInteface implements Interface {
};

class SubClass2 extends SuperClassImplementingInteface {
};

class SomeOtherClass {
};
```

Let's consider the code below. We create a few instances of the classes declared above.

```
SubClass subClass = new SubClass();
Object subClassObj = new SubClass();

SubClass2 subClass2 = new SubClass2();
SomeOtherClass someOtherClass = new SomeOtherClass();
```

Let's now run instanceof operator on the different instances created earlier.

```
System.out.println(subClass instanceof SubClass);//true
System.out.println(subClass instanceof SuperClass);//true
System.out.println(subClassObj instanceof SuperClass);//true

System.out.println(subClass2
        instanceof SuperClassImplementingInteface);//true
```

instanceof can be used with interfaces as well. Since Super Class implements the interface, below code prints true.

```
System.out.println(subClass2
        instanceof Interface);//true
```

If the type compared is unrelated to the object, a compilation error occurs.

```
//System.out.println(subClass
```

```
//            instanceof SomeOtherClass);//Compiler Error
```

Object referred by subClassObj(SubClass)- NOT of type SomeOtherClass

```
System.out.println(subClassObj instanceof SomeOtherClass);//false
```

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.

Coupling Example Problem

Consider the example below:

```
class ShoppingCartEntry {
    public float price;
    public int quantity;
}

class ShoppingCart {
    public ShoppingCartEntry[] items;
}

class Order {
    private ShoppingCart cart;
    private float salesTax;

    public Order(ShoppingCart cart, float salesTax) {
        this.cart = cart;
        this.salesTax = salesTax;
    }

    // This method know the internal details of ShoppingCartEntry and
    // ShoppingCart classes. If there is any change in any of those
    // classes, this method also needs to change.
    public float orderTotalPrice() {
        float cartTotalPrice = 0;
        for (int i = 0; i < cart.items.length; i++) {
            cartTotalPrice += cart.items[i].price
                    * cart.items[i].quantity;
        }
        cartTotalPrice += cartTotalPrice * salesTax;
        return cartTotalPrice;
    }
}
```

Method orderTotalPrice in Order class is coupled heavily with `ShoppingCartEntry` and `ShoppingCart` classes. It uses different properties (items, price, quantity) from these classes. If any of these properties change, orderTotalPrice will also change. This is not good for Maintenance.

Solution

Consider a better implementation with lesser coupling between classes below: In this implementation, changes in ShoppingCartEntry or CartContents might not affect Order class at all.

```
class ShoppingCartEntry
{
    float price;
    int quantity;

    public float getTotalPrice()
    {
        return price * quantity;
    }
}

class CartContents
{
    ShoppingCartEntry[] items;

    public float getTotalPrice()
    {
        float totalPrice = 0;
        for (ShoppingCartEntry item:items)
        {
            totalPrice += item.getTotalPrice();
        }
        return totalPrice;
    }
}

class Order
{
    private CartContents cart;
    private float salesTax;

    public Order(CartContents cart, float salesTax)
    {
        this.cart = cart;
        this.salesTax = salesTax;
    }

    public float totalPrice()
    {
        return cart.getTotalPrice() * (1.0f + salesTax);
    }
}
```

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.

Example Problem

Example class below is downloading from internet, parsing data and storing data to database. The responsibilities of this class are not really related. This is not cohesive class.

```
class DownloadAndStore{
    void downloadFromInternet(){
    }

    void parseData(){
    }

    void storeIntoDatabase(){
    }

    void doEverything(){
        downloadFromInternet();
        parseData();
        storeIntoDatabase();
    }
}
```

Solution

This is a better way of approaching the problem. Different classes have their own responsibilities.

```
class InternetDownloader {
    public void downloadFromInternet() {
    }
}

class DataParser {
    public void parseData() {
    }
}

class DatabaseStorer {
    public void storeIntoDatabase() {
    }
}

class DownloadAndStore {
    void doEverything() {
        new InternetDownloader().downloadFromInternet();
        new DataParser().parseData();
        new DatabaseStorer().storeIntoDatabase();
    }
}
```

What is Encapsulation?

Encapsulation is “hiding the implementation of a Class behind a well defined interface”. Encapsulation helps us to change implementation of a class without breaking other code.

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.

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 {
    }

}
```

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 {
        };
    }

}
```

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

	}

}
```

Modifiers

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.

Example

```
package com.rithus.classmodifiers.defaultaccess.a;

/* No public before class. So this class has default access*/
class DefaultAccessClass {
//Default access is also called package access
}
```

Another Class in Same Package: Has access to default class

```
package com.rithus.classmodifiers.defaultaccess.a;

public class AnotherClassInSamePackage {
    //DefaultAccessClass and AnotherClassInSamePackage
    //are in same package.
    //So, DefaultAccessClass is visible.
    //An instance of the class can be created.
    DefaultAccessClass defaultAccess;
}
```

Class in Different Package: NO access to default class

```
package com.rithus.classmodifiers.defaultaccess.b;

public class ClassInDifferentPackage {
    //Class DefaultAccessClass and Class ClassInDifferentPackage
    //are in different packages (*.a and *.b)
    //So, DefaultAccessClass is not visible to ClassInDifferentPackage

    //Below line of code will cause compilation error if uncommented
    //DefaultAccessClass defaultAccess; //COMPILE ERROR!!
}
```

What is private access modifier?

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

What is default or package access modifier?

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

What is protected access modifier?

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

What is public access modifier?

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

What access types of variables can be accessed from a Class in Same Package?

Look at the code below to understand what can be accessed and what cannot be.

```
package com.rithus.membermodifiers.access;

public class TestClassInSamePackage {
    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();

        example.publicVariable = 5;
        example.publicMethod();

        //privateVariable is not visible
        //Below Line, uncommented, would give compiler error
        //example.privateVariable=5; //COMPILE ERROR
        //example.privateMethod();

        example.protectedVariable = 5;
        example.protectedMethod();

        example.defaultVariable = 5;
        example.defaultMethod();
    }
}
```

What access types of variables can be accessed from a Class in Different Package?

Look at the code below to understand what can be accessed and what cannot be.

```
package com.rithus.membermodifiers.access.different;

import com.rithus.membermodifiers.access.ExampleClass;

public class TestClassInDifferentPackage {
    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();

        example.publicVariable = 5;
        example.publicMethod();

        //privateVariable,privateMethod are not visible
        //Below Lines, uncommented, would give compiler error
        //example.privateVariable=5; //COMPILE ERROR
        //example.privateMethod();//COMPILE ERROR

        //protectedVariable,protectedMethod are not visible
        //Below Lines, uncommented, would give compiler error
        //example.protectedVariable = 5; //COMPILE ERROR
        //example.protectedMethod();//COMPILE ERROR
```

```
        //defaultVariable,defaultMethod are not visible
        //Below Lines, uncommented, would give compiler error
        //example.defaultVariable = 5;//COMPILE ERROR
        //example.defaultMethod();//COMPILE ERROR
    }
}
```

What access types of variables can be accessed from a Sub Class in Same Package?

Look at the code below to understand what can be accessed and what cannot be.

```
package com.rithus.membermodifiers.access;

public class SubClassInSamePackage extends ExampleClass {

    void subClassMethod(){
        publicVariable = 5;
        publicMethod();

        //privateVariable is not visible to SubClass
        //Below Line, uncommented, would give compiler error
        //privateVariable=5; //COMPILE ERROR
        //privateMethod();

        protectedVariable = 5;
        protectedMethod();

        defaultVariable = 5;
        defaultMethod();
    }
}
```

What access types of variables can be accessed from a Sub Class in Different Package?

Look at the code below to understand what can be accessed and what cannot be.

```
package com.rithus.membermodifiers.access.different;

import com.rithus.membermodifiers.access.ExampleClass;

public class SubClassInDifferentPackage extends ExampleClass {

    void subClassMethod(){
        publicVariable = 5;
        publicMethod();

        //privateVariable is not visible to SubClass
        //Below Line, uncommented, would give compiler error
        //privateVariable=5; //COMPILE ERROR
        //privateMethod();
```

```
        protectedVariable = 5;
        protectedMethod();

        //privateVariable is not visible to SubClass
        //Below Line, uncommented, would give compiler error
        //defaultVariable = 5; //COMPILE ERROR
        //defaultMethod();
    }
}
```

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

}
*/
```

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(){

    }
    */
}
```

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

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
}
```

What happens when a variable is marked as volatile?

- Volatile can only be applied to instance variables.
- A volatile variable is one whose value is always written to and read from "main memory". Each thread has its own cache in Java. The volatile variable will not be stored on a Thread cache.

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 created in the constructor.

Conditions & Loops

Why should you always use blocks around if statement?

If blocks (code between { and }) are not used, only the first statement after the if is considered to be part of the if statement.

```
int number = 5;
if(number < 0) //condn is false. So the line in if is not executed.
    number = number + 10; //Not executed
    number++; //This statement is not part of if. Executed.
System.out.println(number);//prints 6
```

Guess the output

```
int m = 15;

if(m>20)
if(m<20)
    System.out.println("m>20");
else
    System.out.println("Who am I?");
```

Nothing is printed to output. Above code is similar to code below

```
if(m>20) {//Condn is false. So, code in if is not executed
    if(m<20)
        System.out.println("m>20");
    else
        System.out.println("Who am I?");
}
```

Guess the output

```
boolean isTrue = false;
if(isTrue==true){
    System.out.println("TRUE TRUE");//Will not be printed
}
if(isTrue=true){
    System.out.println("TRUE");//Will be printed.
}
```

Condition is isTrue=true. This is assignment. Returns true. So, code in if is executed.

Guess the output of this switch block.

```
int number = 2;
switch (number) {
case 1:
    System.out.println(1);
case 2:
    System.out.println(2);
case 3:
    System.out.println(3);
default:
```

```
    System.out.println("Default");
}

Output of above switch
2
3
Default
```

In this example, there is no break statement in every case. If there is no break, then all the case's until we find break are executed.

Since there is no break after case 2, execution falls through to case 3. There is no break in case 3 as well. So, execution falls through to default.

Rule: Code in switch is executed from a matching case until a break or end of switch statement is encountered.

Guess the output of this switch block?

In below example, we have break statements in case 1, 3 and default. There is no break in case 2.

```
number = 2;
switch (number) {
case 1:
    System.out.println(1);
    break;
case 2:
case 3:
    System.out.println("Number is 2 or 3");
    break;
default:
    System.out.println("Default");
    break;
}
```

Program Output

Number is 2 or 3.

Case 2 matches. Since there is no code in case 2, execution falls through to case 3, executes the println. Break statement takes execution out of the switch

Should default be the last case in a switch statement?

default doesn't need to be the last case in an switch. In the example below default is the first case.

```
number = 10;
switch (number) {
default:
    System.out.println("Default");
    break;
case 1:
    System.out.println(1);
    break;
case 2:
```

```
    System.out.println(2);
    break;
case 3:
    System.out.println(3);
    break;
}
```

Example Output

```
Default
```

Can a Switch statement be used around a String

Switch can be used only with Stringm char, byte, short, int or enum

Guess the output of this for loop

There can be multiple statements in Initialization or Operation separated by commas

```
for (int i = 0,j = 0; i < 10; i++,j--) {
    System.out.print(j);
}
```

Code Output

```
0123456789
```

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.
```

What is the output of the program below?

Break statement takes execution out of inner most loop.

```
for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 10; k++) {
        System.out.print(j + "" + k);
        if (k == 5) {
            break;//Takes out of loop using k
        }
    }
}
```

Program Output

```
000102030405101112131415
```

Each time the value of k is 5 the break statement is executed. The break statement takes execution out of the k loop and proceeds to the next value of j.

What is the output of the program below?

To get out of an outer for loop, labels need to be used.

```
outer:
    for (int j = 0; j < 2; j++) {
        for (int k = 0; k < 10; k++) {
            System.out.print(j + "" + k);
            if (k == 5) {
                break outer;//Takes out of loop using j
            }
        }
    }
```

Program Output

```
000102030405
```

Exception Handling

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.

What design pattern is used to implement Exception handling Features in most languages?

When an exception is thrown from a method with no exception handling, it is thrown to the calling method. If there is no exception handling in that method too, it is further thrown up to its calling method and so on. This happens until an appropriate exception handler is found.This is an example of Chain of Responsibility Pattern defined as "a way of passing a request between a chain of objects".

A good real time example is the Loan or Leave Approval Process. When a loan approval is needed, it first goes to the clerk. If he cannot handle it (large amount), it goes to his manager and so on until it is approved or rejected.

```
public static void main(String[] args) {
        method1();
    }

    private static void method1() {
        method2();
    }

    private static void method2() {
        String str = null;
        str.toString();
    }
```

Program Output

```
Exception in thread "main" java.lang.NullPointerException at
com.rithus.exceptionhandling.ExceptionHandlingExample1.method2(ExceptionHandlingExample1.java:1
5)
at
com.rithus.exceptionhandling.ExceptionHandlingExample1.method1(ExceptionHandlingExample1.java:1
0)
at com.rithus.exceptionhandling.ExceptionHandlingExample1.main(ExceptionHandlingExample1.java:6)
```

Look at the stack trace. Exception which is thrown in method2 is propagating to method1 and then to main. This is because there is no exception handling in all 3 methods - main, method1 and method2

What is the need for finally block?

Consider the example below: In method2, a connection is opened. However, because of the exception thrown, connection is not closed. This results in unclosed connections.

```
package com.rithus.exceptionhandling;

class Connection {
    void open() {
        System.out.println("Connection Opened");
    }

    void close() {
        System.out.println("Connection Closed");
    }
}

public class ExceptionHandlingExample1 {

    // Exception Handling Example 1
    // Let's add a try catch block in method2
    public static void main(String[] args) {
        method1();
        System.out.println("Line after Exception - Main");
    }

    private static void method1() {
        method2();
        System.out.println("Line after Exception - Method 1");
    }

    private static void method2() {
        try {
            Connection connection = new Connection();
            connection.open();

            // LOGIC
            String str = null;
            str.toString();

            connection.close();
        } catch (Exception e) {
            // NOT PRINTING EXCEPTION TRACE- BAD PRACTICE
```

```
            System.out.println("Exception Handled - Method 2");
        }
    }
}
```

Output
Connection Opened
Exception Handled - Method 2
Line after Exception - Method 1
Line after Exception - Main

Connection that is opened is not closed. Because an exception has occurred in method2, connection.close() is not run. This results in a dangling (un-closed) connection.

Code with Finally

Finally block is used when code needs to be executed irrespective of whether an exception is thrown. Let us now move connection.close(); into a finally block. Also connection declaration is moved out of the try block to make it visible in the finally block.

```
    private static void method2() {
        Connection connection = new Connection();
        connection.open();
        try {
            // LOGIC
            String str = null;
            str.toString();

        } catch (Exception e) {
            // NOT PRINTING EXCEPTION TRACE - BAD PRACTICE
            System.out.println("Exception Handled - Method 2");
        } finally {
            connection.close();
        }
    }
```

Output
Connection Opened
Exception Handled - Method 2
Connection Closed
Line after Exception - Method 1
Line after Exception - Main

Connection is closed even when exception is thrown. This is because connection.close() is called in the finally block.
Finally block is always executed (even when an exception is thrown). So, if we want some code to be always executed we can move it to finally block.

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()).

Will finally be executed in the program below?

```
private static void method2() {

        Connection connection = new Connection();
        connection.open();
        try {
            // LOGIC
            String str = null;
            str.toString();
            return;
        } catch (Exception e) {
            // NOT PRINTING EXCEPTION TRACE - BAD PRACTICE
            System.out.println("Exception Handled - Method 2");
            return;
        } finally {
            connection.close();
        }
    }
```

Yes. It will be. Finally will be executed even when there is a return statement in try or catch.

Is try without a catch is allowed?

Yes. It is.

```
        private static void method2() {

          Connection connection = new Connection();
          connection.open();
          try {
              // LOGIC
              String str = null;
              str.toString();
          } finally {
              connection.close();
          }
    }
```

Output:
Connection Opened
Connection Closed
Exception in thread "main" java.lang.NullPointerException at com.rithus.exceptionhandling.ExceptionHandlingExample1.method2(ExceptionHandlingExample1.java:33) at com.rithus.exceptionhandling.ExceptionHandlingExample1.method1(ExceptionHandlingExample1.java:22) at com.rithus.exceptionhandling.ExceptionHandlingExample1.main(ExceptionHandlingExample1.java:17)

Try without a catch is useful when you would want to do something (close a connection) even if an exception occurred without handling the exception.

Is try without catch and finally allowed?

No. Below method would give a Compilation Error!! (End of try block)

```
    private static void method2() {
        Connection connection = new Connection();
        connection.open();
        try {
            // LOGIC
            String str = null;
            str.toString();
        }//COMPILER ERROR!!
    }
```

Can you explain the hierarchy of Exception Handling classes?

Throwable is the highest level of Error Handling classes.

Below class definitions show the pre-defined exception hierarchy in Java.

```
//Pre-defined Java Classes
class Error extends Throwable{}
class Exception extends Throwable{}
class RuntimeException extends Exception{}
```

Below class definitions show creation of a programmer defined exception in Java.

```
//Programmer defined classes
class CheckedException1 extends Exception{}
class CheckedException2 extends CheckedException1{}

class UnCheckedException extends RuntimeException{}
class UnCheckedException2 extends UnCheckedException{}
```

What is the difference between Error and Exception?

Error

Error is used in situations when there is nothing a programmer can do about an error. Ex: StackOverflowError, OutOfMemoryError.

Exception

Exception is used when a programmer can handle the exception.

What is the difference between Checked Exceptions and Unchecked Exceptions?

Un-Checked Exception

RuntimeException and classes that extend RuntimeException are called unchecked exceptions. For Example: RuntimeException,UnCheckedException,UnCheckedException2 are unchecked or RunTime Exceptions. There are subclasses of RuntimeException (which means they are subclasses of Exception also.)

Checked Exception

Other Exception Classes (which don't fit the earlier definition). These are also called Checked Exceptions. Exception, CheckedException1,CheckedException2 are checked exceptions. They are subclasses of Exception which are not subclasses of RuntimeException.

How do you throw an exception from a method?

Method addAmounts in Class AmountAdder adds amounts. If amounts are of different currencies it throws an exception.

```
class Amount {
    public Amount(String currency, int amount) {
        this.currency = currency;
        this.amount = amount;
    }

    String currency;// Should be an Enum
    int amount;// Should ideally use BigDecimal
}

// AmountAdder class has method addAmounts which is throwing a RuntimeException
class AmountAdder {
    static Amount addAmounts(Amount amount1, Amount amount2) {
        if (!amount1.currency.equals(amount2.currency)) {
            throw new RuntimeException("Currencies don't match");
        }
        return new Amount(amount1.currency, amount1.amount + amount2.amount);
    }
}

public class ExceptionHandlingExample2 {

    public static void main(String[] args) {
        AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR", 5));
    }

}
```

Output

Exception in thread "main" java.lang.RuntimeException: Currencies don't match
at com.rithus.exceptionhandling.AmountAdder.addAmounts(ExceptionHandlingExample2.java:17)
at com.rithus.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHandlingExample2.java:28)

Exception message shows the type of exception(java.lang.RuntimeException) and the string message passed to the RuntimeException constructor("Currencies don't match");

What happens when you throw a Checked Exception from a method?

Let us now try to change the method addAmounts to throw an Exception instead of RuntimeException. It gives us a compilation error.

```
class AmountAdder {
    static Amount addAmounts(Amount amount1, Amount amount2) {
        if (!amount1.currency.equals(amount2.currency)) {
            throw new Exception("Currencies don't match");// COMPILER ERROR!
// Unhandled exception type Exception
        }
        return new Amount(amount1.currency, amount1.amount + amount2.amount);
    }
}
```

What are the options you have to eliminate compilation errors when handling checked exceptions?

All classes that are not RuntimeException or subclasses of RuntimeException but extend Exception are called CheckedExceptions. The rule for CheckedExceptions is that they should either be handled or thrown. Handled means it should be completed handled - i.e. not throw out of the method. Thrown means the method should declare that it throws the exception

Option 1 : Declaring that a method would throw an exception

Let's look at how to declare throwing an exception from a method.

```
class AmountAdder {
    static Amount addAmounts(Amount amount1, Amount amount2) throws Exception {
        if (!amount1.currency.equals(amount2.currency)) {
            throw new Exception("Currencies don't match");
        }
        return new Amount(amount1.currency, amount1.amount + amount2.amount);
    }
}
```

Look at the line "static Amount addAmounts(Amount amount1, Amount amount2) throws Exception". This is how we declare that a method throws Exception. This results in compilation error in main method. This is because Main method is calling a method which is declaring that it might throw Exception. Main method again has two options a. Throw b. Handle

Code with main method throwing the exception below

```
    public static void main(String[] args) throws Exception {
        AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR", 5));
    }
```

Output

```
Exception in thread "main" java.lang.Exception: Currencies don't match
at com.rithus.exceptionhandling.AmountAdder.addAmounts(ExceptionHandlingExample2.java:17)
at com.rithus.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHandlingExample2.java:28)
```

Option 2 : Handling the Check Exception with a try catch block

main can also handle the exception instead of declaring throws. Code for it below.

```
    public static void main(String[] args) {
      try {
          AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",5));
      } catch (Exception e) {
          System.out.println("Exception Handled in Main");
      }
    }
```

Output

```
Exception Handled in Main
```

How do you create a Custom Exception?

For the scenario above we can create a customized exception, CurrenciesDoNotMatchException. If we want to make it a Checked Exception, we can make it extend Exception class. Otherwise, we can extend RuntimeException class.

Option 1 : Extending Exception or subclass of Exception : Creating Checked Exception

```
class CurrenciesDoNotMatchException extends Exception{
}
```

No we can change the method addAmounts to throw CurrenciesDoNotMatchException - even the declaration of the method changed.

```
class AmountAdder {
    static Amount addAmounts(Amount amount1, Amount amount2)
            throws CurrenciesDoNotMatchException {
        if (!amount1.currency.equals(amount2.currency)) {
            throw new CurrenciesDoNotMatchException();
        }
        return new Amount(amount1.currency, amount1.amount + amount2.amount);
    }
}
```

main method needs to be changed to catch: CurrenciesDoNotMatchException

```
public class ExceptionHandlingExample2 {
    public static void main(String[] args) {
        try {
            AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",
                    5));
        } catch (CurrenciesDoNotMatchException e) {
            System.out.println("Exception Handled in Main" + e.getClass());
        }
    }
}
```

Output:
Exception Handled in Mainclass com.rithus.exceptionhandling.CurrenciesDoNotMatchException

Let's change main method to handle Exception instead of CurrenciesDoNotMatchException

```
public class ExceptionHandlingExample2 {
    public static void main(String[] args) {
        try {
            AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",5));
        } catch (Exception e) {
            System.out.println("Exception Handled in Main" + e.getClass());
        }
    }
}
```

Output:
Exception Handled in Mainclass com.rithus.exceptionhandling.CurrenciesDoNotMatchException

There is no change in output from the previous example. This is because Exception catch block can catch Exception and all subclasses of Exception.

Option 2 : Extend RuntimeException

Let's change the class CurrenciesDoNotMatchException to extend RuntimeException instead of Exception

```
class CurrenciesDoNotMatchException extends RuntimeException{
}
```

Output:
Exception Handled in Mainclass com.rithus.exceptionhandling.CurrenciesDoNotMatchException

Change methods addAmounts in AmountAdder to remove the declaration " throws CurrenciesDoNotMatchException"

No compilation error occurs since RuntimeException and subclasses of RuntimeException are not Checked Exception's. So, they don't need to be handled or declared. If you are interested in handling them, go ahead and handle them. But, java does not require you to handle them.

Remove try catch from main method. It is not necessary since CurrenciesDoNotMatchException is now a RuntimeException.

```
public class ExceptionHandlingExample2 {
    public static void main(String[] args) {
        AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR", 5));
    }
}
```

Output:
Exception in thread "main" com.rithus.exceptionhandling.CurrenciesDoNotMatchException at com.rithus.exceptionhandling.AmountAdder.addAmounts(ExceptionHandlingExample2.java:21)
at com.rithus.exceptionhandling.ExceptionHandlingExample2.main(ExceptionHandlingExample2.java:30)

What is the output of the program below?

```
    public static void main(String[] args) {
        try {
            AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("DOLLAR",
                    5));
        } catch (Exception e) {
            System.out.println("Handled Exception");
        } catch (CurrenciesDoNotMatchException e) {
            System.out.println("Handled CurrenciesDoNotMatchException");
        }
    }
```

Compilation Error. Specific Exception catch blocks should be before the catch block for a Generic Exception. For example, CurrenciesDoNotMatchException should be before Exception.

How do you handle multiple exception types with same exception handling block?

This is a new feature in Java 7.

```
try {
  ...
} catch( IOException | SQLException ex ) {
  ...
}
```

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();
}
```

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;
}
```

Can you explain a few Exception Handling Best Practices?

First of all : In all above examples we have not followed an Exception Handling good practice(s). We were trying to give quick examples. So here is a list of best practices.

- Never Hide Exceptions. At the least log them. printStactTrace method prints the entire stack trace when an exception occurs. If you handle an exception, it is always a good practice to log the trace.
- Do not use exception handling for flow control in a program. They have a significant performance impact.
- Think about the user. What does the user want to see if there is an exception?
- Think about the support developer. What does the support developer need to debug the exception?
- Think about the calling method. Can the calling method do something about the exception being thrown? If not, create un checked exceptions. For example, Spring Framework chooses to make most of the jdbc exceptions as unchecked exceptions because , in most cases, there is nothing that a caller of the method can do about a jdbc exception.
- Have global exception handling.

Miscellaneous Topics

What are the default values in an array?

New Arrays are always initialized with default values.

```
int marks2[] = new int[5];
System.out.println(marks2[0]);//0
```

Default Values

byte,short,int,long 0

float,double 0.0

boolean false

object null

How do you loop around an array using enhanced for loop?

Name of the variable is mark and the array we want to loop around is marks.

```
for (int mark: marks) {
    System.out.println(mark);
}
```

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));
//[[1, 2, 3], [4, 5, 6]]
```

matrix3[0] is a 1D Array

```
System.out.println(matrix3[0]);//[I@86c347
System.out.println(Arrays.toString(matrix3[0]));//[1, 2, 3]
```

How do you compare two arrays?

Arrays can be compared using static method equals defined in Arrays class. Two arrays are equal only if they have the same numbers in all positions and have the same size.

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 4, 5, 6 };

System.out.println(Arrays
        .equals(numbers1, numbers2)); //false

int[] numbers3 = { 1, 2, 3 };
System.out.println(Arrays.equals(numbers1, numbers3)); //true
```

What is an Enum?

Enum allows specifying a list of values for a Type. Consider the example below. It declares an enum Season with 4 possible values.

```
enum Season {
    WINTER, SPRING, SUMMER, FALL
};
```

Can you use a Switch Statement around an Enum?

Example below shows how we can use a switch around an enum.

```
        //Using switch statement on an enum
        public int getExpectedMaxTemperature() {
            switch (this) {
            case WINTER:
                return 5;
            case SPRING:
            case FALL:
                return 10;
            case SUMMER:
                return 20;
            }
                   return -1;// Dummy since Java does not recognize this is possible
      }
```

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
    }
```

What are Asserts used for?

Assertions are introduced in Java 1.4. They enable you to validate assumptions. If an assert fails (i.e. returns false), AssertionError is thrown (if assertions are enabled). Basic assert is shown in the example below

```
private int computerSimpleInterest(int principal,float interest,int years){
    assert(principal>0);
    return 100;
}
```

When should Asserts be used?

Assertions should not be used to validate input data to a public method or command line argument. IllegalArgumentException would be a better option. In public method, only use assertions to check for cases which are never supposed to happen.

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.

Can you explain Garbage Collection with an example?

Let's say the below method is called from a function.

```
void method(){
    Calendar calendar = new GregorianCalendar(2000,10,30);
    System.out.println(calendar);
}
```

An object of the class GregorianCalendar is created on the heap by the first line of the function with one reference variable calendar.

After the function ends execution, the reference variable calendar is no longer valid. Hence, there are no references to the object created in the method.

JVM recognizes this and removes the object from the heap. This is called Garbage Collection.

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

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();

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.

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.

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
```

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.

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]
```

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

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();
```

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
```

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.

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.

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.

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

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

Collections

Why do we need Collections in Java?

Arrays are not dynamic. Once an array of a particular size is declared, the size cannot be modified. To add a new element to the array, a new array has to be created with bigger size and all the elements from the old array copied to new array.

Collections are used in situations where data is dynamic. Collections allow adding an element, deleting an element and host of other operations. There are a number of Collections in Java allowing to choose the right Collection for the right context.

What are the important interfaces in the Collection Hierarchy?

The most important interfaces and their relationships are highlighted below.

```
interface Collection<E> extends Iterable<E> {
}

// Unique things only - Does not allow duplication.
// If obj1.equals(obj2) then only one of them can be in the Set.
interface Set<E> extends Collection<E> {

}

// LIST OF THINGS
// Cares about which position each object is in
// Elements can be added in by specifying position - where should it be added
in
// If element is added without specifying position - it is added at the end
interface List<E> extends Collection<E> {
}

// Arranged in order of processing - A to-do list for example
// Queue interface extends Collection. So, it supports all Collection
Methods.
interface Queue<E> extends Collection<E> {
}


// A,C,A,C,E,C,M,D,H,A => {("A",5),("C",2)}
// Key - Value Pair {["key1",value1],["key2",value2],["key3",value3]}
// Things with unique identifier
interface Map<K, V> {
}
```

What are the important methods that are declared in the Collection Interface?

Most important methods declared in the collection interface are the methods to add and remove an element. add method allows adding an element to a collection and delete method allows deleting an element from a collection.

size() methods returns number of elements in the collection. Other important methods defined as part of collection interface are shown below.

```
interface Collection<E> extends Iterable<E>
{
  boolean add(E paramE);
  boolean remove(Object paramObject);

  int size();
  boolean isEmpty();
  void clear();

  boolean contains(Object paramObject);
  boolean containsAll(Collection<?> paramCollection);

  boolean addAll(Collection<? extends E> paramCollection);
  boolean removeAll(Collection<?> paramCollection);
  boolean retainAll(Collection<?> paramCollection);


  Iterator<E> iterator();

  //A NUMBER OF OTHER METHODS AS WELL..
}
```

Can you explain briefly about the List Interface?

List interface extends Collection interface. So, it contains all methods defined in the Collection interface. In addition, List interface allows operation specifying the position of the element in the Collection.

Most important thing to remember about a List interface - any implementation of the List interface would maintain the insertion order. When an element A is inserted into a List (without specifying position) and then another element B is inserted, A is stored before B in the List.
When a new element is inserted without specifying a position, it is inserted at the end of the list of elements.

However, We can also use the void add(int position, E paramE); method to insert an element at a specific position.

Listed below are some of the important methods in the List interface (other than those inherited from Collection interface):

```
interface List<E> extends Collection<E>
{
  boolean addAll(int paramInt, Collection<? extends E> paramCollection);

  E get(int paramInt);
```

```
  E set(int paramInt, E paramE);

  void add(int paramInt, E paramE);
  E remove(int paramInt);

  int indexOf(Object paramObject);
  int lastIndexOf(Object paramObject);

  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int paramInt);
  List<E> subList(int paramInt1, int paramInt2);
}
```

Explain about ArrayList with an example?

ArrayList implements the list interface. So, ArrayList stores the elements in insertion order (by default). Element's can be inserted into and removed from ArrayList based on their position.

Let's look at how to instantiate an ArrayList of integers.

```
List<Integer> integers = new ArrayList<Integer>();
```

Code like below is permitted because of auto boxing. 5 is auto boxed into Integer object and stored in ArrayList.

```
integers.add(5);//new Integer(5)
```

Add method (by default) adds the element at the end of the list.

Can an ArrayList have Duplicate elements?

ArrayList can have duplicates (since List can have duplicates).

```
List<String> arraylist = new ArrayList<String>();

//adds at the end of list
arraylist.add("Sachin");//[Sachin]

//adds at the end of list
arraylist.add("Dravid");//[Sachin, Dravid]

//adds at the index 0
arraylist.add(0, "Ganguly");//[Ganguly, Sachin, Dravid]

//List allows duplicates - Sachin is present in the list twice
arraylist.add("Sachin");//[ Ganguly, Sachin, Dravid, Sachin]

System.out.println(arraylist.size());//4
System.out.println(arraylist.contains("Dravid"));//true
```

How do you iterate around an ArrayList using Iterator?

Example below shows how to iterate around an ArrayList.

```
Iterator<String> arraylistIterator = arraylist
        .iterator();
while (arraylistIterator.hasNext()) {
    String str = arraylistIterator.next();
    System.out.println(str);//Prints the 4 names in the list on separate lines.
}
```

How do you sort an ArrayList?

Example below shows how to sort an ArrayList. It uses the Collections.sort method.

```
List<String> numbers = new ArrayList<String>();
numbers.add("one");
numbers.add("two");
numbers.add("three");
numbers.add("four");
System.out.println(numbers);//[one, two, three, four]

//Strings - By Default - are sorted alphabetically
Collections.sort(numbers);

System.out.println(numbers);//[four, one, three, two]
```

How do you sort elements in an ArrayList using Comparable interface?

Consider the following class Cricketer.

```
class Cricketer implements Comparable<Cricketer> {
    int runs;
    String name;

    public Cricketer(String name, int runs) {
        super();
        this.name = name;
        this.runs = runs;
    }

    @Override
    public String toString() {
        return name + " " + runs;
    }

    @Override
    public int compareTo(Cricketer that) {
        if (this.runs > that.runs) {
            return 1;
        }
        if (this.runs < that.runs) {
            return -1;
        }
        return 0;
    }
}
```

Let's now try to sort a list containing objects of Cricketer class.

```
List<Cricketer> cricketers = new ArrayList<Cricketer>();
cricketers.add(new Cricketer("Bradman", 9996));
cricketers.add(new Cricketer("Sachin", 14000));
cricketers.add(new Cricketer("Dravid", 12000));
cricketers.add(new Cricketer("Ponting", 11000));
System.out.println(cricketers);
//[Bradman 9996, Sachin 14000, Dravid 12000, Ponting 11000]
```

Now let's try to sort the cricketers.

```
Collections.sort(cricketers);
System.out.println(cricketers);
//[Bradman 9996, Ponting 11000, Dravid 12000, Sachin 14000]
```

How do you sort elements in an ArrayList using Comparator interface?

Other option to sort collections is by creating a separate class which implements Comparator interface. Example below:

```
class DescendingSorter implements Comparator<Cricketer> {

    //compareTo returns -1 if cricketer1 < cricketer2
    //                   1 if cricketer1 > cricketer2
    //                   0 if cricketer1 = cricketer2

    //Since we want to sort in descending order,
    //we should return -1 when runs are more
    @Override
    public int compare(Cricketer cricketer1,
            Cricketer cricketer2) {
        if (cricketer1.runs > cricketer2.runs) {
            return -1;
        }
        if (cricketer1.runs < cricketer2.runs) {
            return 1;
        }
        return 0;
    }

}
```

Let's now try to sort the previous defined collection:

```
Collections
        .sort(cricketers, new DescendingSorter());

System.out.println(cricketers);
//[Sachin 14000, Dravid 12000, Ponting 11000, Bradman 9996]
```

What is Vector class? How is it different from an ArrayList?

```
class Vector /* implements List<E>, RandomAccess */{
```

```
	// Thread Safe - Synchronized Methods
	// implements RandomAccess, a marker interface, meaning it support fast
	// almost constant time - access
}
```

Vector has the same operations as an ArrayList. However, all methods in Vector are synchronized. So, we can use Vector if we share a list between two threads and we would want to them synchronized.

What is LinkedList? What interfaces does it implement? How is it different from an ArrayList?

```
class LinkedList /* implements List<E>, Queue */{
	// Elements are doubly linked - forward and backword - to one another
	// Ideal choice to implement Stack or Queue
	// Iteration is slower than ArrayList
	// Fast Insertion and Deletion
	// Implements Queue interface. Supports methods like peek(), poll()
	// and remove()
}
```

Linked List extends List and Queue.Other than operations exposed by the Queue interface, LinkedList has the same operations as an ArrayList. However, the underlying implementation of Linked List is different from that of an ArrayList.

ArrayList uses an Array kind of structure to store elements. So, inserting and deleting from an ArrayList are expensive operations. However, search of an ArrayList is faster than LinkedList.

LinkedList uses a linked representation. Each object holds a link to the next element. Hence, insertion and deletion are faster than ArrayList. But searching is slower.

Can you briefly explain about the Set Interface?

There are hardly any new methods in the Set interface other than those in the Collection interface. The major difference is that Set interface does not allow duplication. Set interface represents a collection that contains no duplicate elements.

```
// Unique things only - Does not allow duplication.
// If obj1.equals(obj2) then only one of them can be in the Set.
interface Set<E> extends Collection<E> {

}
```

What are the important interfaces related to the Set Interface?

```
// Unique things only - Does not allow duplication.
// If obj1.equals(obj2) then only one of them can be in the Set.
interface Set<E> extends Collection<E> {
```

```
}

//Main difference between Set and SortedSet is - an implementation of
//SortedSet interface maintains its elements in a sorted order.  Set
//interface does not guarantee any Order.
interface SortedSet<E> extends Set<E> {

	SortedSet<E> subSet(E fromElement, E toElement);

	SortedSet<E> headSet(E toElement);

	SortedSet<E> tailSet(E fromElement);

	E first();

	E last();

}

//A SortedSet extended with navigation methods reporting closest matches for
//given search targets.
interface NavigableSet<E> extends SortedSet<E> {
	E lower(E e);

	E floor(E e);

	E ceiling(E e);

	E higher(E e);

	E pollFirst();

	E pollLast();
}
```

What is the difference between Set and SortedSet interfaces?

SortedSet Interface extends the Set Interface. Both Set and SortedSet do not allow duplicate elements.

Main difference between Set and SortedSet is - an implementation of SortedSet interface maintains its elements in a sorted order. Set interface does not guarantee any Order. For example, If elements 4,5,3 are inserted into an implementation of Set interface, it might store the elements in any order. However, if we use SortedSet, the elements are sorted. The SortedSet implementation would give an output 3,4,5.

Can you give examples of classes that implement the Set Interface?

HashSet, LinkedHashSet and TreeSet implement the Set interface.

```
// Order of Insertion : A, X , B
// Possible Order of Storing : X, A ,B
class HashSet /* implements Set */{
	// unordered, unsorted - iterates in random order
	// uses hashCode()
}

// Order of Insertion :A, X, B
// Order of Storing : A, X, B
class LinkedHashSet /* implements Set */{
	// ordered - iterates in order of insertion
	// unsorted
	// uses hashCode()
}

// Order of Insertion :A,C,B
// Order of Storing : A,B,C
class TreeSet /* implements Set,NavigableSet */{
	// 3,5,7
	// sorted - natural order
	// implements NavigableSet
}
```

What is a HashSet?

HashSet implements set interface. So, HashSet does not allow duplicates. However, HashSet does not support ordering. The order in which elements are inserted is not maintained.

HashSet Example

```
Set<String> hashset = new HashSet<String>();

hashset.add("Sachin");
System.out.println(hashset);//[Sachin]

hashset.add("Dravid");
System.out.println(hashset);//[Sachin, Dravid]
```

Let's try to add Sachin to the Set now. Sachin is Duplicate. So will not be added. returns false.

```
hashset.add("Sachin");//returns false since element is not added
System.out.println(hashset);//[Sachin, Dravid]
```

What is a LinkedHashSet? How is different from a HashSet?

LinkedHashSet implements set interface and exposes similar operations to a HashSet. Difference is that LinkedHashSet maintains insertion order. When we iterate a LinkedHashSet, we would get the elements back in the order in which they were inserted.

What is a TreeSet? How is different from a HashSet?

TreeSet implements Set, SortedSet and NavigableSet interfaces.TreeSet is similar to HashSet except that it stores element's in Sorted Order.

```
Set<String> treeSet = new TreeSet<String>();

treeSet.add("Sachin");
System.out.println(treeSet);//[Sachin]
```

Notice that the list is sorted after inserting Dravid.

```
//Alphabetical order
treeSet.add("Dravid");
System.out.println(treeSet);//[Dravid, Sachin]
```

Notice that the list is sorted after inserting Ganguly.

```
treeSet.add("Ganguly");
System.out.println(treeSet);//[Dravid, Ganguly, Sachin]

//Sachin is Duplicate. So will not be added. returns false.
treeSet.add("Sachin");//returns false since element is not added
System.out.println(treeSet);//[Dravid, Ganguly, Sachin]
```

Can you give examples of implementations of NavigableSet?

TreeSet implements this interface. Let's look at an example with TreeSet. Note that elements in TreeSet are sorted.

```
TreeSet<Integer> numbersTreeSet = new TreeSet<Integer>();
numbersTreeSet.add(55);
numbersTreeSet.add(25);
numbersTreeSet.add(35);
numbersTreeSet.add(5);
numbersTreeSet.add(45);
```

NavigableSet interface has following methods.

Lower method finds the highest element lower than specified element. Floor method finds the highest element lower than or equal to specified element. Corresponding methods for finding lowest number higher than specified element are higher and ceiling. A few examples using the Set created earlier below.

```
//Find the highest number which is lower than 25
System.out.println(numbersTreeSet.lower(25));//5

//Find the highest number which is lower than or equal to 25
System.out.println(numbersTreeSet.floor(25));//25

//Find the lowest number higher than 25
System.out.println(numbersTreeSet.higher(25));//35
```

```
//Find the lowest number higher than or equal to 25
System.out.println(numbersTreeSet.ceiling(25));//25
```

Explain briefly about Queue Interface?

Queue Interface extends Collection interface. Queue Interface is typically used for implementation holding elements in order for some processing.

Queue interface offers methods peek() and poll() which get the element at head of the queue. The difference is that poll() method removes the head from queue also. peek() would keep head of the queue unchanged.

```
interface Queue<E> extends Collection<E> {

	//Inserts the specified element into this queue
	//Throws exception in case of failure
	boolean add(E paramE);

	//Inserts the specified element into this queue
	//Returns false in case of failure
	boolean offer(E paramE);

	//Retrieves and removes the head of this queue.
	//Throws Exception if Queue is empty
	E remove();

	//Retrieves and removes the head of this queue.
	//returns null if Queue is empty
	E poll();

	E element();

	E peek();
}
```

What are the important interfaces related to the Queue Interface?

Two important interfaces are Deque and BlockingQueue.

Explain about the Deque interface?

```
//A linear collection that supports element insertion and removal at both
ends
interface Deque<E> extends Queue<E> {
	void addFirst(E e);

	void addLast(E e);

	boolean offerFirst(E e);
```

```
	boolean offerLast(E e);

	E removeFirst();

	E removeLast();

	E pollFirst();

	E pollLast();

	E getFirst();

	E getLast();

	E peekFirst();

	E peekLast();

	boolean removeFirstOccurrence(Object o);

	boolean removeLastOccurrence(Object o);

}
```

Explain the BlockingQueue interface?

```
//A Queue that additionally supports operations that wait for
//the queue to become non-empty when retrieving an
//element, and wait for space to become available in the queue when
//storing an element.
interface BlockingQueue<E> extends Queue<E> {
	//Same as in Queue Interface
	//Inserts the specified element into queue IMMEDIATELY
	//Throws exception in case of failure
	boolean add(E e);

	//Same as in Queue Interface
	//Inserts the specified element into queue IMMEDIATELY
	//Returns false in case of failure
	boolean offer(E e); //Same as in Queue Interface

	//Inserts the specified element into this queue, waiting
	//if necessary for space to become available.
	void put(E e) throws InterruptedException;

	//waiting up to the specified wait time
	boolean offer(E e, long timeout, TimeUnit unit) throws
InterruptedException;
```

```
	//waits until element becomes available
	E take() throws InterruptedException;

	//waits for specified time and returns null if time expires
	E poll(long timeout, TimeUnit unit) throws InterruptedException;

	int remainingCapacity();

	boolean remove(Object o);

	public boolean contains(Object o);

	int drainTo(Collection<? super E> c);

	int drainTo(Collection<? super E> c, int maxElements);
}
```

What is a PriorityQueue?

PriorityQueue implements the Queue interface.

```
//The elements of the priority queue are ordered according to their natural
ordering
class PriorityQueue /* implements Queue */{
	// sorted - natural order

}

//Using default constructor - uses natural ordering of numbers
//Smaller numbers have higher priority
PriorityQueue<Integer> priorityQueue = new PriorityQueue<Integer>();
```

Adding an element into priority queue - offer method

```
priorityQueue.offer(24);
priorityQueue.offer(15);
priorityQueue.offer(9);
priorityQueue.offer(45);

System.out.println(priorityQueue);//[9, 24, 15, 45]
```

Peek method examples

```
//peek method get the element with highest priority.
System.out.println(priorityQueue.peek());//9
//peek method does not change the queue
System.out.println(priorityQueue);//[9, 24, 15, 45]

//poll method gets the element with highest priority.
System.out.println(priorityQueue.poll());//9
//peek method removes the highest priority element from the queue.
System.out.println(priorityQueue);//[24, 15, 45]
```

```
//This comparator gives high priority to the biggest number.
Comparator reverseComparator = new Comparator<Integer>() {
    public int compare(Integer paramT1,
            Integer paramT2) {
        return paramT2 - paramT1;
    }

};
```

Can you give example implementations of the BlockingQueue interface?

```
class ArrayBlockingQueue /*implements BlockingQueue*/{
      //uses Array - optionally-bounded
}

class LinkedBlockingQueue /*implements BlockingQueue*/{
      //uses Linked List - optionally-bounded
      //Linked queues typically have higher throughput than array-based
queues but
      //less predictable performance in most concurrent applications.
}
```

Can you briefly explain about the Map Interface?

First and foremost, Map interface does not extend Collection interface. So, it does not inherit any of the methods from the Collection interface.

A Map interface supports Collections that use a key value pair. A key-value pair is a set of linked data items: a key, which is a unique identifier for some item of data, and the value, which is either the data or a pointer to the data. Key-value pairs are used in lookup tables, hash tables and configuration files. A key value pair in a Map interface is called an Entry.

Put method allows to add a key, value pair to the Map.

```
  V put(K paramK, V paramV);
```

Get method allows to get a value from the Map based on the key.

```
  V get(Object paramObject);
```

Other important methods in Map Inteface are shown below:

```
interface Map<K, V>
{
  int size();
  boolean isEmpty();

  boolean containsKey(Object paramObject);
  boolean containsValue(Object paramObject);

  V get(Object paramObject);
```

```
  V put(K paramK, V paramV);
  V remove(Object paramObject);

  void putAll(Map<? extends K, ? extends V> paramMap);
  void clear();

  Set<K> keySet();
  Collection<V> values();
  Set<Entry<K, V>> entrySet();

  boolean equals(Object paramObject);
  int hashCode();

  public static abstract interface Entry<K, V>
  {
    K getKey();
    V getValue();
    V setValue(V paramV);
    boolean equals(Object paramObject);
    int hashCode();
  }
}
```

What is difference between Map and SortedMap?

SortedMap interface extends the Map interface. In addition, an implementation of SortedMap interface maintains keys in a sorted order.

Methods are available in the interface to get a ranges of values based on their keys.

```
public interface SortedMap<K, V> extends Map<K, V> {
    Comparator<? super K> comparator();

    SortedMap<K, V> subMap(K fromKey, K toKey);

    SortedMap<K, V> headMap(K toKey);

    SortedMap<K, V> tailMap(K fromKey);

    K firstKey();

    K lastKey();
}
```

What is a HashMap?

HashMap implements Map interface – there by supporting key value pairs. Let's look at an example.

HashMap Example

```
Map<String, Cricketer> hashmap = new HashMap<String, Cricketer>();
hashmap.put("sachin",
        new Cricketer("Sachin", 14000));
hashmap.put("dravid",
        new Cricketer("Dravid", 12000));
hashmap.put("ponting", new Cricketer("Ponting",
```

```
        11500));
hashmap.put("bradman", new Cricketer("Bradman",
        9996));
```

What are the different methods in a Hash Map?

get method gets the value of the matching key.

```
System.out.println(hashmap.get("ponting"));//Ponting 11500

//if key is not found, returns null.
System.out.println(hashmap.get("lara"));//null
```

If existing key is reused, it would replace existing value with the new value passed in.

```
//In the example below, an entry with key "ponting" is already present.
//Runs are updated to 11800.
hashmap.put("ponting", new Cricketer("Ponting",
        11800));

//gets the recently updated value
System.out.println(hashmap.get("ponting"));//Ponting 11800
```

What is a TreeMap? How is different from a HashMap?

TreeMap is similar to HashMap except that it stores keys in sorted order. It implements NavigableMap interface and SortedMap interfaces along with the Map interface.

```
Map<String, Cricketer> treemap = new TreeMap<String, Cricketer>();
treemap.put("sachin",
        new Cricketer("Sachin", 14000));
System.out.println(treemap);
//{sachin=Sachin 14000}
```

We will now insert a Cricketer with key dravid. In sorted order,dravid comes before sachin. So, the value with key dravid is inserted at the start of the Map.

```
treemap.put("dravid",
        new Cricketer("Dravid", 12000));
System.out.println(treemap);
//{dravid=Dravid 12000, sachin=Sachin 14000}
```

We will now insert a Cricketer with key ponting. In sorted order, ponting fits in between dravid and sachin.

```
treemap.put("ponting", new Cricketer("Ponting",
        11500));
System.out.println(treemap);
//{dravid=Dravid 12000, ponting=Ponting 11500, sachin=Sachin 14000}

treemap.put("bradman", new Cricketer("Bradman",
        9996));
System.out.println(treemap);
```

```
//{bradman=Bradman 9996, dravid=Dravid 12000, ponting=Ponting 11500, sachin=Sachin
14000}
```

Can you give an example of implementation of NavigableMap Interface?

TreeMap is a good example of a NavigableMap interface implementation. Note that keys in TreeMap are sorted.

NavigableMap is a SortedMap extended with navigation methods reporting closest matches for given search targets.

```
interface NavigableMap<K, V> extends SortedMap<K, V> {
	Map.Entry<K, V> lowerEntry(K key);

	K lowerKey(K key);

	Map.Entry<K, V> floorEntry(K key);

	K floorKey(K key);

	Map.Entry<K, V> ceilingEntry(K key);

	K ceilingKey(K key);

	Map.Entry<K, V> higherEntry(K key);

	K higherKey(K key);

	Map.Entry<K, V> firstEntry();

	Map.Entry<K, V> lastEntry();

	Map.Entry<K, V> pollFirstEntry();

	Map.Entry<K, V> pollLastEntry();

	NavigableMap<K, V> descendingMap();

	NavigableSet<K> navigableKeySet();

	NavigableSet<K> descendingKeySet();
}
```

Example Program

```
TreeMap<Integer, Cricketer> numbersTreeMap = new TreeMap<Integer, Cricketer>();
numbersTreeMap.put(55, new Cricketer("Sachin",
        14000));
numbersTreeMap.put(25, new Cricketer("Dravid",
```

```
        12000));
numbersTreeMap.put(35, new Cricketer("Ponting",
        12000));
numbersTreeMap.put(5,
        new Cricketer("Bradman", 9996));
numbersTreeMap
        .put(45, new Cricketer("Lara", 10000));
```

lowerKey method finds the highest key lower than specified key. floorKey method finds the highest key lower than or equal to specified key. Corresponding methods for finding lowest key higher than specified key are higher and ceiling. A few examples using the Map created earlier below.

```
//Find the highest key which is lower than 25
System.out.println(numbersTreeMap.lowerKey(25));//5

//Find the highest key which is lower than or equal to 25
System.out.println(numbersTreeMap.floorKey(25));//25

//Find the lowest key higher than 25
System.out.println(numbersTreeMap.higherKey(25));//35

//Find the lowest key higher than or equal to 25
System.out.println(numbersTreeMap.ceilingKey(25));//25
```

What are the static methods present in the Collections class?

- static int binarySearch(List, key)
  - Can be used only on sorted list
- static int binarySearch(List, key, Comparator)
- static void reverse(List)
  - Reverse the order of elements in a List.
- static Comparator reverseOrder();
  - Return a Comparator that sorts the reverse of the collection current sort sequence.
- static void sort(List)
- static void sort(List, Comparator)

Advanced Collections

What is the difference between synchronized and concurrent collections in Java?

Synchronized collections are implemented using synchronized methods and synchronized blocks. Only one thread can executing any of the synchronized code at a given point in time. This places severe restrictions on the concurrency of threads – thereby affecting performance of the application. All the pre Java 5 synchronized collections (HashTable & Vector, for example) use this approach.

Post Java 5, collections using new approaches to synchronization are available in Java. These are called concurrent collections. More details below.

Explain about the new concurrent collections in Java?

Post Java 5, collections using new approaches to synchronization are available in Java. These are called concurrent collections. Examples of new approaches are :

- Copy on Write
- Compare and Swap
- Locks

These new approaches to concurrency provide better performance in specific context's. We would discuss each of these approaches in detail below.

Explain about CopyOnWrite concurrent collections approach?

Important points about Copy on Write approach

- All values in collection are stored in an internal immutable (not-changeable) array. A new array is created if there is any modification to the collection.
- Read operations are not synchronized. Only write operations are synchronized.

Copy on Write approach is used in scenarios where reads greatly out number write's on a collection. CopyOnWriteArrayList & CopyOnWriteArraySet are implementations of this approach. Copy on Write collections are typically used in Subject – Observer scenarios, where the observers very rarely change. Most frequent operations would be iterating around the observers and notifying them.

Example : CopyOnWriteArrayList : public boolean add(E e)

What is CompareAndSwap approach?

Compare and Swap is one of the new approaches (Java 5) introduced in java to handle synchronization. In traditional approach, a method which modifies a member variable used by multiple threads is completely synchronized – to prevent other threads accessing stale value.

In compare and swap approach, instead of synchronizing entire method, the value of the member variable before calculation is cached. After the calculation, the cached value is compared with the

current value of member variable. If the value is not modified, the calculated result is stored into the member variable. If another thread has modified the value, then the calculation can be performed again. Or skipped – as the need might be.

ConcurrentLinkedQueue uses this approach.

What is a Lock? How is it different from using synchronized approach?

CopyOnWriteArrayList : final ReentrantLock lock = this.lock;

When 10 methods are declared as synchronized, only one of them is executed by any of the threads at any point in time. This has severe performance impact.

Another new approach introduced in Java 5 is to use lock and unlock methods. Lock and unlock methods are used to divide methods into different blocks and help enhance concurrency. The 10 methods can be divided into different blocks, which can be synchronized based on different variables.

What is initial capacity of a Java Collection?

Extract from the reference : http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html. An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put).

The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations.

What is load factor?

Refer answer to Initial Capacity above.

When does a Java collection throw UnsupportedOperationException?

All Java Collections extend Collection interface. So, they have to implement all the methods in the Collection interface. However, certain Java collections are optimized to be used in specific conditions and do not support all the Collection operations (methods). When an unsupported operation is called on a Collection, the Collection Implementation would throw an UnsupportedOperationException.

Arrays.asList returns a fixed-size list backed by the specified array. When an attempt is made to add or remove from this collection an UnsupportedOperationException is thrown. Below code throws UnsupportedOperationException.

```
List<String> list=Arrays.asList(new String[]{"ac","bddefe"});

list.remove();//throws UnsupportedOperationException
```

What is difference between fail-safe and fail-fast iterators?

Fail Fast Iterators throw a ConcurrentModificationException if there is a modification to the underlying collection is modified. This was the default behavior of the synchronized collections of pre Java 5 age.

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class FailFast {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<String, String>();
		map.put("key1", "value1");
		map.put("key2", "value2");
		map.put("key3", "value3");

		Iterator<String> iterator = map.keySet().iterator();

		while (iterator.hasNext()) {
			System.out.println(map.get(iterator.next()));
			map.put("key4", "value4");
		}

	}

}
```

Fail Safe Iterators do not throw exceptions even when there are changes in the collection. This is the default behavior of the concurrent collections, introduced since Java 5.

Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure.

Fail Safe is efficient when traversal operations vastly outnumber mutations

```
package com.in28minutes.java.collections;

import java.util.Iterator;
import java.util.concurrent.ConcurrentHashMap;

public class FailSafe {

	public static void main(String[] args) {
		ConcurrentHashMap<String, String> map = new ConcurrentHashMap<String, String>();
		map.put("key1", "value1");
		map.put("key2", "value2");
		map.put("key3", "value3");
```

```
                Iterator<String> iterator = map.keySet().iterator();

                while (iterator.hasNext()) {
                        System.out.println(map.get(iterator.next()));
                        map.put("key4", "value4");
                }

        }
}
```

What are atomic operations in Java?

Atomic Access Java Tutorial states "In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete".

Let's assume we are writing a multi threaded program. Let's create an int variable i. Even a small operation, like i++ (increment), is not thread safe. i++ operation involves three steps.

1. Read the value which is currently stored in i
2. Add one to it (atomic operation).
3. Store it in i

In a multi-threaded environment, there can be unexpected results. For example, if thread1 is reading the value (step 1) and immediately after thread2 stores the value (step 3).

To prevent these, Java provides atomic operations. Atomic operations are performed as a single unit without interference from other threads ensuring data consistency.

A good example is AtomicInteger. To increment a value of AtomicInteger, we use the incrementAndGet() method. Java ensures this operation is Atomic.

What is BlockingQueue in Java?

BlockingQueue interface is introduced in Java specifically to address specific needs of some Producer Consumer scenarios. BlockedQueue allows the consumer to wait (for a specified time or infinitely) for an element to become available.

Generics

What are Generics?

Generics are used to create Generic Classes and Generic methods which can work with different Types(Classes).

Why do we need Generics? Can you give an example of how Generics make a program more flexible?

Consider the class below:

```
class MyList {
    private List<String> values;

    void add(String value) {
        values.add(value);
    }

    void remove(String value) {
        values.remove(value);
    }
}
```

MyList can be used to store a list of Strings only.

```
        MyList myList = new MyList();
        myList.add("Value 1");
        myList.add("Value 2");
```

To store integers, we need to create a new class. This is problem that Generics solve. Instead of hard-coding String class as the only type the class can work with, we make the class type a parameter to the class.

Example with Generics

Let's replace String with T and create a new class. Now the MyListGeneric class can be used to create a list of Integers or a list of Strings

```
class MyListGeneric<T> {
    private List<T> values;

    void add(T value) {
        values.add(value);
    }

    void remove(T value) {
        values.remove(value);
    }

    T get(int index) {
        return values.get(index);
    }
```

```
}

MyListGeneric<String> myListString = new MyListGeneric<String>();
myListString.add("Value 1");
myListString.add("Value 2");

MyListGeneric<Integer> myListInteger = new MyListGeneric<Integer>();
myListInteger.add(1);
myListInteger.add(2);
```

How do you declare a Generic Class?

Note the declaration of class:

```
class MyListGeneric<T>
```

Instead of T, We can use any valid identifier

What are the restrictions in using generic type that is declared in a class declaration?

If a generic is declared as part of class declaration, it can be used any where a type can be used in a class - method (return type or argument), member variable etc. For Example: See how T is used as a parameter and return type in the class MyListGeneric.

How can we restrict Generics to a subclass of particular class?

In MyListGeneric, Type T is defined as part of class declaration. Any Java Type can be used a type for this class. If we would want to restrict the types allowed for a Generic Type, we can use a Generic Restrictions. Consider the example class below: In declaration of the class, we specified a constraint "T extends Number". We can use the class MyListRestricted with any class extending (any sub class of) Number - Float, Integer, Double etc.

```
class MyListRestricted<T extends Number> {
    private List<T> values;

    void add(T value) {
        values.add(value);
    }

    void remove(T value) {
        values.remove(value);
    }

    T get(int index) {
        return values.get(index);
    }
}

MyListRestricted<Integer> restrictedListInteger = new MyListRestricted<Integer>();
```

```
restrictedListInteger.add(1);
restrictedListInteger.add(2);
```

String not valid substitute for constraint "T extends Number".

```
//MyListRestricted<String> restrictedStringList =
//                new MyListRestricted<String>();//COMPILER ERROR
```

How can we restrict Generics to a super class of particular class?

In MyListGeneric, Type T is defined as part of class declaration. Any Java Type can be used a type for this class. If we would want to restrict the types allowed for a Generic Type, we can use a Generic Restrictions. In declaration of the class, we specified a constraint "T super Number". We can use the class MyListRestricted with any class that is a super class of Number class.

Can you give an example of a Generic Method?

A generic type can be declared as part of method declaration as well. Then the generic type can be used anywhere in the method (return type, parameter type, local or block variable type).

Consider the method below:

```
    static <X extends Number> X doSomething(X number){
        X result = number;
        //do something with result
        return result;
    }
```

The method can now be called with any Class type extend Number.

```
Integer i = 5;
Integer k = doSomething(i);
```

Multi Threading

What is the need for Threads in Java?

Threads allow Java code to run in parallel. Let's look at an example to understand what we can do with Threads.

Need for Threads

We are creating a Cricket Statistics Application. Let's say the steps involved in the application are

- STEP I: Download and Store Bowling Statistics => 60 Minutes
- STEP II: Download and Store Batting Statistics => 60 Minutes
- STEP III: Download and Store Fielding Statistics => 15 Minutes
- STEP IV: Merge and Analyze => 25 Minutes

Steps I, II and III are independent and can be run in parallel to each other. Run individually this program takes 160 minutes. We would want to run this program in lesser time. Threads can be a solution to this problem. Threads allow us to run STEP I, II and III in parallel and run Step IV when all Steps I, II and III are completed.

Below example shows the way we would write code usually – without using Threads.

```
ThreadExamples example = new ThreadExamples();
example.downloadAndStoreBattingStatistics();
example.downloadAndStoreBowlingStatistics();
example.downloadAndStoreFieldingStatistics();

example.mergeAndAnalyze();
```

downloadAndStoreBowlingStatistics starts only after downloadAndStoreBattingStatistics completes execution. downloadAndStoreFieldingStatistics starts only after downloadAndStoreBowlingStatistics completes execution. What if I want to run them in parallel without waiting for the others to complete?

This is where Threads come into picture. Using Multi-Threading we can run each of the above steps in parallel and synchronize when needed. We will understand more about synchronization later.

How do you create a thread?

Creating a Thread class in Java can be done in two ways. Extending Thread class and implementing Runnable interface. Let's create the BattingStatisticsThread extending Thread class and BowlingStatisticsThread implementing Runnable interface.

How do you create a thread by extending Thread class?

Thread class can be created by extending Thread class and implementing the public void run() method.

Look at the example below: A dummy implementation for BattingStatistics is provided which counts from 1 to 1000.

```
class BattingStatisticsThread extends Thread {
    //run method without parameters
    public void run() {
        for (int i = 0; i < 1000; i++)
            System.out
            .println("Running Batting Statistics Thread "
                            + i);
    }
}
```

How do you create a thread by implementing Runnable interface?

Thread class can also be created by implementing Runnable interface and implementing the method declared in Runnable interface "public void run()". Example below shows the Batting Statistics Thread implemented by implementing Runnable interface.

```
class BowlingStatisticsThread implements Runnable {
    //run method without parameters
    public void run() {
        for (int i = 0; i < 1000; i++)
            System.out
            .println("Running Bowling Statistics Thread "
                            + i);
    }
}
```

How do you run a Thread in Java?

Running a Thread in Java is slightly different based on the approach used to create the thread.

Thread created Extending Thread class

When using inheritance, An object of the thread needs be created and start() method on the thread needs to be called. Remember that the method that needs to be called is not run() but it is start().

```
BattingStatisticsThread battingThread1 = new BattingStatisticsThread();
battingThread1.start();
```

Thread created implementing RunnableInterface.

Three steps involved.

- Create an object of the BowlingStatisticsThread(class implementing Runnable).
- Create a Thread object with the earlier object as constructor argument.
- Call the start method on the thread.

```
BowlingStatisticsThread battingInterfaceImpl = new BowlingStatisticsThread();
Thread battingThread2 = new Thread(
        battingInterfaceImpl);
battingThread2.start();
```

What are the different states of a Thread?

Different states that a thread can be in are defined the class State.

- NEW;
- RUNNABLE;
- RUNNING;
- BLOCKED/WAITING;
- TERMINATED/DEAD;

Let's consider the example that we discussed earlier.

Example Program

```
LINE 1: BattingStatisticsThread battingThread1 = new BattingStatisticsThread();
LINE 2: battingThread1.start();

LINE 3: BowlingStatisticsThread battingInterfaceImpl = new BowlingStatisticsThread();
LINE 4: Thread battingThread2 = new Thread(battingInterfaceImpl);
LINE 5:battingThread2.start();
```

Description

A thread is in NEW state when an object of the thread is created but the start method is not yet called. At the end of line 1, `battingThread1 is in NEW state.`

A thread is in RUNNABLE state when it is eligible to run, but not running yet. (A number of Threads can be in RUNNABLE state. Scheduler selects which Thread to move to RUNNING state). In the above example, sometimes the Batting Statistics thread is running and at other time, the Bowling Statistics Thread is running. When Batting Statistics thread is Running, the Bowling Statistics thread is ready to run. It's just that the scheduler picked Batting Statistics thread to run at that instance and vice-versa. When Batting Statistics thread is Running, the Bowling Statistics Thread is in Runnable state (Note that the Bowling Statistics Thread is not waiting for anything except for the Scheduler to pick it up and run it).

A thread is RUNNING state when it's the one that is currently , what else to say, Running.

A thread is in BLOCKED/WAITING/SLEEPING state when it is not eligible to be run by the Scheduler. Thread is alive but is waiting for something. An example can be a Synchronized block. If Thread1 enters synchronized block, it blocks all the other threads from entering synchronized code on the same instance or class. All other threads are said to be in Blocked state.

A thread is in DEAD/TERMINATED state when it has completed its execution. Once a thread enters dead state, it cannot be made active again.

What is priority of a thread? How do you change the priority of a thread?

Scheduler can be requested to allot more CPU to a thread by increasing the threads priority. Each thread in Java is assigned a default Priority 5. This priority can be increased or decreased (Range 1 to 10).

If two threads are waiting, the scheduler picks the thread with highest priority to be run. If all threads have equal priority, the scheduler then picks one of them randomly. Design programs so that they don't depend on priority.

Thread Priority Example

Consider the thread example declared below:

```
class ThreadExample extends Thread {
    public void run() {
        for (int i = 0; i < 1000; i++)
            System.out
                    .println( this.getName() + " Running "
                            + i);
    }
}
```

Priority of thread can be changed by invoking setPriority method on the thread.

```
ThreadExample thread1 = new ThreadExample();
thread1.setPriority(8);
```

Java also provides predefined constants Thread.MAX_PRIORITY(10), Thread.MIN_PRIORITY(1), Thread.NORM_PRIORITY(5) which can be used to assign priority to a thread.

What is ExecutorService?

The java.util.concurrent.ExecutorService interface is a new way of executing tasks asynchronously in the background. An ExecutorService is very similar to a thread pool.

Can you give an example for ExecutorService?

Below example shows how to create an Executor Service and use it to run a task implementing the Runnable interface.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
      public void run() {
            System.out.println("From ExecutorService");
      }
});

System.out.println("End of Main");

executorService.shutdown();
```

Explain different ways of creating Executor Services.

There are three ways of creating executor services. Below example shows the three different ways. executorService1 can execute one task at a time. executorService2 can execute 10 tasks at a time. executorService3 can execute tasks after certain delay or periodically.

```
// Creates an Executor that uses a single worker thread operating off an
// unbounded queue.
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
```

```
// Creates a thread pool that reuses a fixed number of threads
// operating off a shared unbounded queue. At any point, the parameter
// specifies the most threads that will be active processing tasks.

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

// Creates a thread pool that can schedule commands to run after a
// given delay, or to execute periodically.
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

How do you check whether an ExecutionService task executed successfully?

We can use a Future to check the return value. Below example shows how it can be done. Future get method would return null if the task finished successfully.

```
Future future = executorService1.submit(new Runnable() {
	public void run() {
		System.out.println("From executorService1");
	}
});

future.get(); // returns null if the task has finished correctly.
```

What is Callable? How do you execute a Callable from ExecutionService?

Runnable interface's run method has a return type void. So, it cannot return any result from executing a task. However, a Callable interface's call method has a return type. If you have multiple return values possible from a task, we can use the Callable interface. Example shows how to create a Callable interface and execute it using an executor service. The return value is printed to the output.

```
Future futureFromCallable = executorService1.submit(new Callable() {
	public String call() throws Exception {
		return "RESULT";
	}
});

System.out.println("futureFromCallable.get() = "
		+ futureFromCallable.get());
```

What is synchronization of threads?

Since Threads run in parallel, a new problem arises. What if thread1 modifies data which is being accessed by thread2? How do we ensure that different threads don't leave the system in an inconsistent state? This problem is usually called synchronization problem.

Let's first look at an example where this problem can occur. Consider the code in the setAndGetSum method.

```
int setandGetSum(int a1, int a2, int a3) {
```

```
    cell1 = a1;
    sleepForSomeTime();
    cell2 = a2;
    sleepForSomeTime();
    cell3 = a3;
    sleepForSomeTime();
    return cell1 + cell2 + cell3;
}
```

If following method is running in two different threads, funny things can happen. After setting the value to each cell, there is a call for the Thread to sleep for some time. After Thread 1 sets the value of cell1, it goes to Sleep. So, Thread2 starts executing. If Thread 2 is executing "**return** cell1 + cell2 + cell3;", it uses cell1 value set by Thread 1 and cell2 and cell3 values set by Thread 2. This results in the unexpected results that we see when the method is run in parallel. What is explained is one possible scenario. There are several such scenarios possible.

The way you can prevent multiple threads from executing the same method is by using the synchronized keyword on the method. If a method is marked synchronized, a different thread gets access to the method only when there is no other thread currently executing the method.

Let's mark the method as synchronized:

```
synchronized int setandGetSum(int a1, int a2, int a3) {
    cell1 = a1;
    sleepForSomeTime();
    cell2 = a2;
    sleepForSomeTime();
    cell3 = a3;
    sleepForSomeTime();
    return cell1 + cell2 + cell3;
}
```

Can you give an example of a synchronized block?

All code which goes into the block is synchronized on the current object.

```
    void synchronizedExample2() {
        synchronized (this){
        //All code goes here..
        }
    }
```

Can a static method be synchronized?

Yes. Consider the example below.

```
    synchronized static int getCount(){
        return count;
    }
```

Static methods and block are synchronized on the class. Instance methods and blocks are synchronized on the instance of the class i.e. an object of the class. Static synchronized methods and instance

synchronized methods don't affect each other. This is because they are synchronized on two different things.

```
static int getCount2(){
    synchronized (SynchronizedSyntaxExample.class) {
        return count;
    }
}
```

What is the use of join method in threads?

Join method is an instance method on the Thread class. Let's see a small example to understand what join method does.

Let's consider the thread's declared below: thread2, thread3, thread4

```
ThreadExample thread2 = new ThreadExample();
ThreadExample thread3 = new ThreadExample();
ThreadExample thread4 = new ThreadExample();
```

Let's say we would want to run thread2 and thread3 in parallel but thread4 can only run when thread3 is finished. This can be achieved using join method.

Join method example

Look at the example code below:

```
thread3.start();
thread2.start();
thread3.join();//wait for thread 3 to complete
System.out.println("Thread3 is completed.");
thread4.start();
```

thread3.join() method call force the execution of main method to stop until thread3 completes execution. After that, thread4.start() method is invoked, putting thread4 into a Runnable State.

Overloaded Join method

Join method also has an overloaded method accepting time in milliseconds as a parameter.

```
thread4.join(2000);
```

In above example, main method thread would wait for 2000 ms or the end of execution of thread4, whichever is minimum.

Describe a few other important methods in Threads?

Thread yield method

Yield is a static method in the Thread class. It is like a thread saying " I have enough time in the limelight. Can some other thread run next?".

A call to yield method changes the state of thread from RUNNING to RUNNABLE. However, the scheduler might pick up the same thread to run again, especially if it is the thread with highest priority.

Summary is yield method is a request from a thread to go to Runnable state. However, the scheduler can immediately put the thread back to RUNNING state.

Thread sleep method

sleep is a static method in Thread class. sleep method can throw a InterruptedException. sleep method causes the thread in execution to go to sleep for specified number of milliseconds.

What is a deadlock?

Let's consider a situation where thread1 is waiting for thread2 (thread1 needs an object whose synchronized code is being executed by thread1) and thread2 is waiting for thread1. This situation is called a Deadlock. In a Deadlock situation, both these threads would wait for one another for ever.

What are the important methods in java for inter-thread communication?

Important methods are wait, notify and notifyAll.

What is the use of wait method?

Below snippet shows how wait is used. wait method is defined in the Object class. This causes the thread to wait until it is notified.

```
synchronized(thread){
    thread.start();
    thread.wait();
}
```

What is the use of notify method?

Below snippet shows how notify is used. notify method is defined in the Object class. This causes the object to notify other waiting threads.

```
      synchronized (this) {0
        calculateSumUptoMillion();
        notify();
    }
```

What is the use of notifyAll method?

If more than one thread is waiting for an object, we can notify all the threads by using notifyAll method.

thread.notifyAll();

Can you write a synchronized program with wait and notify methods?

```
package com.rithus.threads;

class Calculator extends Thread {
    long sumUptoMillion;
    long sumUptoTenMillion;
```

```
    public void run() {
        synchronized (this) {
            calculateSumUptoMillion();
            notify();
        }
        calculateSumUptoTenMillion();
    }

    private void calculateSumUptoMillion() {
        for (int i = 0; i < 1000000; i++) {
            sumUptoMillion += i;
        }
        System.out.println("Million done");
    }

    private void calculateSumUptoTenMillion() {
        for (int i = 0; i < 10000000; i++) {
            sumUptoTenMillion += i;
        }
        System.out.println("Ten Million done");
    }
}

public class ThreadWaitAndNotify {
    public static void main(String[] args) throws InterruptedException {
        Calculator thread = new Calculator();
        synchronized(thread){
            thread.start();
            thread.wait();
        }
        System.out.println(thread.sumUptoMillion);
    }
}
```

Output

```
Million done
499999500000
Ten Million done
```

Functional Programming - Lamdba Expressions and Streams

What is functional programming?

Functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

Can you give an example of functional programming?

```
@Test
public void sumOfOddNumbers_Usual() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	int sum = 0;
	for (int number : numbers)
		if (number % 2 != 0)
			sum += number;
	assertEquals(11, sum);
}

@Test
public void sumOfOddNumbers_FunctionalProgrammingExample() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	int sum = numbers.stream().filter(Test123::isOdd).reduce(0,
			Integer::sum);
	assertEquals(11, sum);
}

static boolean isOdd(int number) {
	return number % 2 != 0;
}
```

What is a Stream?

A Stream is a source of objects. In the above example, we created a stream from List.

Streams have Intermediate Operations and Terminal Operations. In the example above, we used filter as intermediate operation and reduce as a terminal operation.

Explain about streams with an example?

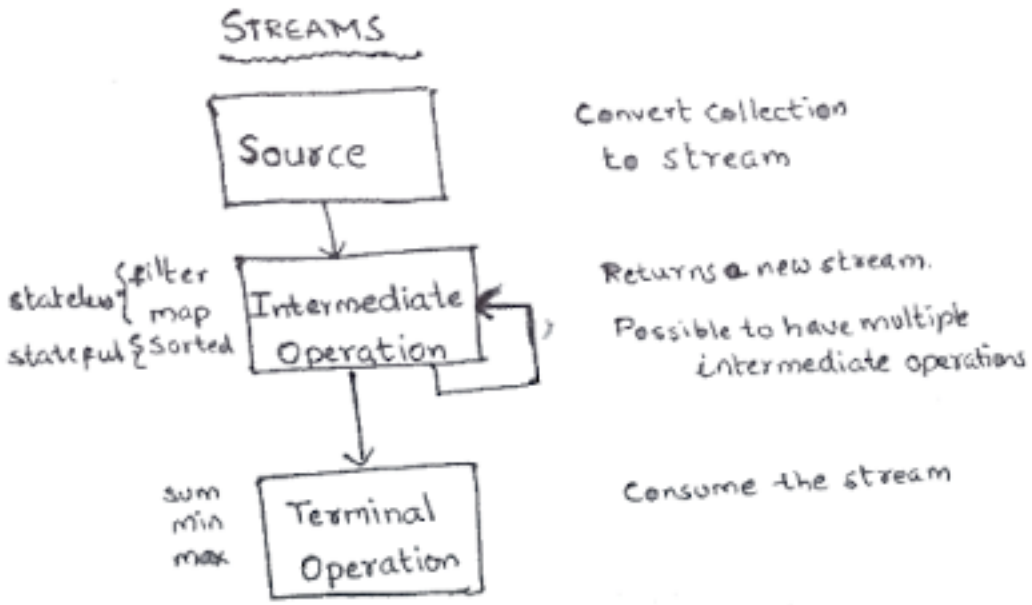
Streams are introduced in Java 8. In combination with Lambda expressions, they attempt to bring some of the important functional programming concepts to Java.

A stream is a sequence of elements supporting sequential and parallel aggregate operations. Consider the example code below. Following steps are done:

- Step I : Creating an array as a stream
- Step II : Use Lambda Expression to create a filter
- Step III : Use map function to invoke a String function

- Step IV : Use sorted function to sort the array
- Step V : Print the array using forEach

```
Arrays.stream(new String[] {
      "Ram", "Robert", "Rahim"
})
      .filter(s -> s.startsWith("Ro"))
      .map(String::toLowerCase)
      .sorted()
      .forEach(System.out::println);
```



In general any use of streams involves

- Source - Creation or use of existing stream : Step I above
- Intermediate Operations - Step II, III and IV above. Intermediate Operations return a new stream
- Terminal Operation – Step V. Consume the stream. Print it to output or produce a result (sum,min,max etc).

Intermediate Operations are of two kinds

- Stateful : Elements need to be compared against one another (sort, distinct etc)
- Stateless : No need for comparing with other elements (map, filter etc)

What are Intermediate Operations in Streams?

An Intermediate Operation on a Stream returns another Stream.

Examples : map, filter, distinct, sorted.

Distinct Example

```
@Test
public void streamExample_Distinct() {
	List<Integer> numbers = Arrays.asList(1, 1, 2, 6, 2, 3);
	numbers.stream().distinct().forEach(System.out::print);
	// 1263
}
```

Sorted Example

```
@Test
public void streamExample_Sorted() {
	List<Integer> numbers = Arrays.asList(1, 1, 2, 6, 2, 3);
	numbers.stream().sorted().forEach(System.out::print);
	// 112236
}
```

Filter Example

```
@Test
public void streamExample_Filter() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	numbers.stream().filter(Test123::isOdd).forEach(System.out::print);
	//137
}
```

What are Terminal Operations in Streams?

Terminal Operation either produce a result or create a side effect.

reduce is used to cumulate elements.

```
@Test
public void sumOfOddNumbers_FunctionalProgramming() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	int sum = numbers.stream().filter(Test123::isOdd).reduce(0,
			Integer::sum);
	assertEquals(11, sum);
}
```

forEach is used to create a side effect. Print to Output. Store to database.

```
@Test
public void streamExample_Filter() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	numbers.stream().filter(Test123::isOdd).forEach(System.out::print);
	//137
}
```

collect is used to group elements to a collection

```
@Test
public void streamExample_Collect() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	List<Integer> oddNumbers = numbers.stream().filter(Test123::isOdd)
			.collect(Collectors.toList());
	System.out.println(oddNumbers);
	// [1, 3, 7]
}
```

What are Method References?

Integer::sum, System.out::print in the above examples are method references. These two are simple static methods which are used instead of Lambda Expressions.

What are Lambda Expressions?

A lambda expression is an anonymous function. Simply put, it's a method without a declaration. There will be no access modifiers, no return value declaration, and no name. It's a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once, and the method definition is short.

Syntax : Parameters -> Executed code

Can you give an example of Lambda Expression?

In the example below, number -> System.out.print(number) is a lambda expression.

```
@Test
public void lambdaExpression_simpleExample() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	numbers.stream().filter(Test123::isOdd).forEach(
			number -> System.out.print(number));
	// 137
}
```

Can you explain the relationship between Lambda Expression and Functional Interfaces?

Look at the earlier example : Function we passed in is number -> System.out.print(number). Input to this function is number. The function consumes it and prints it to the output.

forEach function has an interface - void java.util.stream.Stream.forEach(Consumer<T> action)

The JavaDoc for java.util.function.Consumer<T> reads - @FunctionalInterface : Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.

When ever we create a Lambda Expression, we are defining a function which implements a pre-defined/custom defined Functional Interface.

What is a Predicate?

```
@FunctionalInterface
public interface Predicate<T> {
	boolean test(T t);
}

@Test
public void lambdaExpression_predicate() {
	List<Integer> numbers = Arrays.asList(1, 3, 4, 6, 2, 7);
	numbers.stream().filter((number) -> (number % 2 != 0)).forEach(
			number -> System.out.print(number));
	// 137
}
```

(number) -> (number % 2 != 0) is a Predicate. Takes an argument and returns true of false.

Signature of filter function : Stream<T> java.util.stream.Stream.filter(Predicate<? super T> predicate). filter returns a stream consisting of the elements of this stream that match the given predicate.

What is the functional interface - Function?

```
@FunctionalInterface
public interface Function<T, R> {
	R apply(T t);
}
```

What is a Consumer?

```
@FunctionalInterface
public interface Consumer<T> {
	void accept(T t);
}
```

Can you give examples of functional interfaces with multiple arguments?

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
R apply(T t, U u);
}
```

New Features

What are the new features in Java 5?

- Generics
- Enhanced for Loop
- Autoboxing/Unboxing
- Varargs
- Static Import
- Concurrent Collections
    - Copy on Write
    - Compare and Swap
- Locks

What are the new features in Java 6?

Java 6 has very few important changes in terms of api's. There are a few performance improvements but none significant enough to deserve a mention.

What are the new features in Java 7?

- Diamond Operator.
    - Example : Map<String , List <Trade>> trades = new TreeMap <> ();
- Using String in switch statements
- Automatic resource management
    - try(resources_to_be_cleant){ // your code }
- Numeric literals with underscores
- Improved exception handling
    - Multiple catches in same block
    - catch(ExceptionOne | ExceptionTwo | ExceptionThree e)

What are the new features in Java 8?

Java 8 brought in a number of important new features.

- Lamda Expressions. Example : Runnable java8Runner = () -> { sop("I am running"); };
- Nashorn : javascript engine that enables us to run javascript to run on a jvm
- String.join() function
- Streams