JAVA 8 FEATURES AND IMPORTANT JAVA DEVELOPER INTERVIEW QUESTIONS

Experience: 2 to 5 years

ABSTRACT

This PDF contains all the important Java 8 features and very important interview questions for the Java Developer profile. If you are planning to give any interview in the year 2021 with primary tech stack as Java 8, Spring & Hibernate, this document can be very useful.

Ankit Tandon

Java 8 Features

- ➔ Lambda Expression is single line implementation of abstract method present in functional interfaces.
- ➔ function interfaces are interfaces with SAM
- ➔ FI apart from SAM, can also have object class methods (generally used for updating documentation purpose)
- ➔ Data Type of Lambda Expression is FI.
- ➔ It is optional to use @FunctionalInterface over FI.
- ➔ Lambda Expression can be passed as variable in a function
- ➔ Lamda Expression is an object without an Identity.
- ➔ FI examples: Runnable, Comparator, FileFilter

```
@FunctionalInterface
public interface Runnable {
   public abstract void run();
}
```

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
        //other default & object class method which are not counted
}
```

```
@FunctionalInterface
public interface FileFilter {
    // Tests whether or not the specified abstract pathname should be included in a pathname list.
   boolean accept(File pathname);
```

```
}
```

➔ Lambda Expression Example:

Java.lang

```
Runnable r = () -> {
                    for(int i=0;i<6;i++){
                        System.out.println("Lamda Runnable " + i)
                    }
                };
```

Java.util

```
                    ***first time in java variable doesn't need any type
Comparator<String> c = (s1, s2) -> Integer.compare(s1.length(), s2.length());
```

Java.io

```
FileFilter f = (File pathname) -> { String fname =  file.getName();
                                    return fname.endsWith(".java");
                                   //to filter out all java file from a folder
                                   }
```

- ➔ New Package Introduced- java.util.function
  - 43 Functional Interfaces in total
- ➔ Devided in mainly 4 categories – Supplier, Consumer, Predicate & Function
  - Supplier<T> -> T get();
  - Concumer<T> -. void accept(T t)
  - BiConsumer<T, U> -> void accept(T t, U u)
  - Predicate<T> -> Boolean test(T t)
  - BiPredicate<T,U> Boolean test(T t, U u)
  - Function<T,R> -> R apply(T t),

- BiFunction<T,U,R> -> R apply(T t, U u)
- UnaryOperator<T> extends Function<T,T>
- BinaryOperator<T> extends BiFunction<T,T,T>

Method References (different syntax for lambda expression)

1)

Consumer<String> c= str -> System.out.println(str);

Can also be written as

Consumer<String> c= str -> System.out::println;

2)

Comparator <Integer> c = (i1,i2) -> Integer.compare(i1, i2);

Can also be written as

Comparator <Integer> c = Integer::compare;

➔ Generally we use lambda expressions with Collection package

```
List<Customer> customerList = new ArrayList<>();
customerList.forEach(customer-> Sysout(customer));
customerList.forEach(System.out::println);
```

- Now, From where forEach method comes into picture?
- To introduce foreach method, we have to add it to Iterable interface and then all the classes implementing Iterable Interface have to over write it and lot of refactoring is required.
- So, In java 8 this has been handled by adding Default method in Interface.
- i.e.

```
public interface Iterable<T> {
  Iterator<T> iterator();
```

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);

    for (T t : this) {
        action.accept(t);
    }
}
```

Default Method

- ➔ This is new Java 8 concept.
- ➔ It allows to change the old interfaces without breaking the existing implementation.
- ➔ Static methods are also introduced in java 8

Example of default method:

```
Predicate<String> p1 = s-> s.length()<20;
Predicate<String> p2 = s-> s.length()>10;
Predicate<String> p3 = p1.and(p2);
```

- ➔ Here, and() is a default method of Predicate Interface
- ➔ Another example is Predicate.isEqual(item to be checked);
- To chain to consumers, we can use c1.andThen(c2)

What is stream?

- **It is a java typed Interface**
- **Public Interface Stream<T> extends BaseStream<T, Stream<T>> { }**
- **It is not a collection (generally confused with)**

What does it actually do?

- **It gives way to efficiently process large amount of data and small amount of data as well.**
- **It can also process data in parallel.**
- **Efficiently here means-**
    - **Data can be processed in parallel automatically. The Developer doesn't need to write any technical code for the same.**
    - **Why do we need to process the data in parallel? -> It will leverage us to use the capacity of multicore CPU.**

- **All the process are done in a pipelined format. It will avoid unnecessary intermediary computation.**

- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- In more simple words, A Stream is an object on which one can define operations.
- Operation like – map, filter, reduce etc
- Point to remember- Stream doesn't hold any data.
- It is an object that doesn't change the data it processes. (taken as a rule, not mandatory, but if not followed in child classes, developer will have to face lot of problem)
- Stream Object is able to process the data in one pass.
- Stream Object is optimized from algorithm point of view.

How can we build a Stream()?

1. List<Person> list = new ArrayList<Person>();
   Stream<Person> stream = list.stream();
   Stream.forEach(p-> System.out.println(p));
   or
   Stream.forEach(System.out::println); // method reference
   ➔ * println() method is a consumer FI.
   ➔ Consumer Interface

```
@FunctionalInterface
public interface Consumer<T> {

void accept(T t);

  default Consumer<T> andThen(Consumer<? super T> after) {
     Objects.requireNonNull(after);
     return (T t) -> { accept(t); after.accept(t); };
   }
}
```

   ➔ Consumer Interface can be implemented using FI.
   ➔ Consumer<T> c = p-> System.out.println(p);

   ➔ Default andThen() method is used for consumer chaining

   ➔ Example –

➔ List<String> result = new ArrayList<>();
List<String> list = new ArrayList<>();
Consumer<String> c1 = result::add;
Consumer<String> c2 = System.out::println;

List.stream().forEach(c1.andThen(c2))

➔ A Second Operation: Filter

List<String> result = new ArrayList<>();
List<String> list = new ArrayList<>();
Stream<String> c1 = list.stream();
Stream<String> c2 = c1.filter(s->s.length()>3)

- Filter takes Predicate as a parameter.
- Here filter() method is able to create new instance of stream.
- c2 does not hold anything. It is only the declaration of the filter stream created.
- Predicate<String> p = s->s.length()>3;
- Predicate<String> p2 = p.isEqual("two");
- Predicate<String> p3 = p.isEqual("eight");

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t);


   default Predicate<T> and(Predicate<? super T> other) {// Used for
chaining
      Objects.requireNonNull(other);
      return (t) -> test(t) && other.test(t);
   }

   default Predicate<T> negate() {                    // Used for chaining
      return (t) -> !test(t);
   }
   default Predicate<T> or(Predicate<? super T> other) {// Used for chaining
      Objects.requireNonNull(other);
      return (t) -> test(t) || other.test(t);
```

```
    }
    static <T> Predicate<T> isEqual(Object targetRef) {
      return (null == targetRef)
            ? Objects::isNull
            : object -> targetRef.equals(object);
    }
}
```

- isEqual() is a static method used to create new predicate
- ***Here static method in Interface comes into picture, which is also a new feature of Java 8***
- new way to create stream-
  Stream<String> s= Stream.of("one","two","eight");

- A Call to filter method is Lazy.
- And all the methods of stream that return another stream are lazy.
- Lazy means- they(those streams) are not processing any data they are just declaration of Stream.
- We can also say-
  An operation on a Stream that return another stream is called an Intermediary Operation.
- An operation on a Stream that doesn't return another stream is called a Terminal Operation.

Intermediate Operations:

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.
   List number = Arrays.asList(2,3,4,5);
   List square = number.stream().map(x->x*x).collect(Collectors.toList());
   Here, number.stream() is called to get the stream object of the list and then number.stream().map() is called which is also an intermidiary operation, thus it returns a stream.

   It takes Function FI as an argument.
   (1 abstract, 2 defaullt and 1 static methood)

```
@FunctionalInterface
public interface Function<T, R> {
```

```
    R apply(T t);

     // Used for chaining
    default <V> Function<V, R> compose(Function<? super V, ? extends T>
before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

     // Used for chaining
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
{
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

     // Utility function – returns the same function
    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.
   List names = Arrays.asList("Reflection","Collection","Stream");
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

3. **sorted:** The sorted method is used to sort the stream.
   List names = Arrays.asList("Reflection","Collection","Stream");
   List result = names.stream().sorted().collect(Collectors.toList());

4. **peek:** It is exactly similar to forEach but it returns a stream. Peek takes consumer as a parameter.

5. **Flatmap:** It is an extension of map method functionality along with the flattening operation, i.e. flatmap works on stream of stream, and returns a single stream of data.
   ***** flattening means removing the second level of stream**

Terminal Operations:

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.
   List number = Arrays.asList(2,3,4,5,3);
   Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
   Other Collectors method- toList(), joining(",",),groupingBy(), mapping() – to turn list to map, counting(),Collectors.toCollection(TreeSet::new)

2. **forEach:** The forEach method is used to iterate through every element of the stream.
   List number = Arrays.asList(2,3,4,5);
   number.stream().map(x->x*x).forEach(y->System.out.println(y));

3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.
   The reduce method takes a BinaryOperator as a parameter.
   List number = Arrays.asList(2,3,4,5);
   int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Reduction

- It is a last step in stream API reduce call.
- It is basically of two types-
  - Agregation: sql operation like – min, max, avg, sum etc
    List<integer> ages = Arrays.asList(12,23,13,27);
    Stream<Integer> stream = ages.stream();
    Integer sum = stream.reduce(0, (arg1,arg2)-> arg1+arg2));

    Here,
    1st argument : It is the identity element of the reduction operation. It is a non mandatory parameter.
    2nd argument is a reduction operation of type BinaryOperator which is a child Interface of BiFunction FI.

Identity Element plays very important role during corner cases-

i.e.
1) If Stream has no element.
2) If Stream has only one element.

In first case, Identity element is returned.
In Second case, the only element of the list is returned. It is only possible if we choose the correct identity element.

For sum,

```
List<Integer> array = Arrays.asList(-2, 0, 4, 6, 8);

    // Finding sum of all elements
    int sum = array.stream().reduce(0,
          (element1, element2) -> element1 + element2);

    // Displaying sum of all elements
    System.out.println("The sum of all elements is " + sum);
```

0 is the Identity element.

=>Optional

- *Now For max,*
- *0 here doesn't work as identity here.*

```
List<String> words = Arrays.asList("GFG", "Geeks", "for",
                                   "GeeksQuiz", "GeeksforGeeks");

    // The lambda expression passed to
    // reduce() method takes two Strings
    // and returns the longer String.
    // The result of the reduce() method is
    // an Optional because the list on which
    // reduce() is called may be empty.
          Optional<String> longestString = words.stream()
                .reduce((word1, word2)
```

```
                -> word1.length() > word2.length()
                        ? word1 : word2);

    // Displaying the longest String
    longestString.ifPresent(System.out::println);
```

now this longeststring stream will return null if the words list is empty, which may cause null pointer exception.
So, to handle this java has brought a new concept of Optional class.

Optional means <<there might be no result>>

```
List<Integer> ages = …;
Stream<Integer> stream = ages.stream();
Optional<Integer> max = stream.max(Comparator.naturalOrder());
```

Methods of Optional Class:

1. max.get()
2. max.isPresent()
3. max.ifPresent()
4. max.orElse(0)// for assigning default value
5. max.orElseThrow(MyException::new)
   etc etc…

Different Types of Reduction Operation

1. max(), min(), count()
2. allMatch(),noneMatch(),anymatch() – Boolean reduction
3. findFirst(),findAny() – Reduction that return **Optional**

Point to Remember

- All the operation present in an stream work in a single go for each element of data. This makes stream more efficient.

- Stream once opened and utilized, cannot be reused. Stream has to be retrieved once again to perform another terminal operation.

Java8 Date and Time API

- Till Java 7 we just have two classes to work with date, i.e. java.util.Date class and java.util.Calendar class.

- Date class of Java 7 and earlier was a mutable class, i.e. the value of date can be modified by outside of the class where date is initialized.

The Date API in Java 8 – new API, so New Package – java.time.*

It is based on the completely new concept –

(A) Instant:

1. Instant: An Instant is a point on the timeline. The precision of this timeline is nanoseconds.
2. Instant 0 is 1st January, 1970 at Midnight GMT.
3. Instant.MIN is 1 billion year ago.
4. Instant.MAX is 31st December, of Year 1000000000
5. Instant.now() is the current Instant.
6. *An Instant object is Immutable*

```
Instant i1= Instant.now()
Instant i2= Instant.now()
Duration difference = Duration.between(i1,i2);
Long millis = difference.tiMillis();
```

(B) Duration:

1. Methods: toNanos(), toMillis(),toSeconds(), toMinutes(), toDays() etc
2. Opertation Methods- minusNanos(),plusNanos(), multiplyBy(), devidedBy(), negated(), isZero(), isNegative(),… etc

(C) LocalDate:

```
LocalDate now = LocalDate.now();
LocalDate ld = LocalDate.of(1564, Month.APRIL, 23);
```

(D) Period:

1. A Period is the amount of time elapsed between two periods.
2. Methods- same kind of method as in duration
3. Period p = d.until(now);
   Sysout(p.getYears());

(E) DateAdjuster:

- Useful to add or subtract an amount of time to an Instant or a LocalDate
- LocalDate now = LocalDate.now();
- LocalDate nextSunday = now.with(TemporalAdjusters.next(DayOfWeak.SUNDAY));
- // this will return next Sunday date
- TemporalAdjusters have 14 static methods to adjust Instant or LocalDate.

(F) LocalTime:

- A LocalTime is time of the day
- Ex: 10:40
- LocalTime now = LocalTime.now();
- LocalTime time = LocalTime.of(10,40);

(G) Zoned Time:

- There are time zones all over the earth.
- Java uses the IANA database.

- All Zones can be retrieved using-
  - Set<String> zoneIds = ZoneId.getAvailableZoneIds()
  - String ukTZ = ZoneId.of("Europe/London");
- Class for handling zone time – ZoneDateTime

How to Format a Date?

- New Date Time API have new formatter – DateTimeFormatter class
- DateTimeFormatter proposs a set of predefined formatter, available as constants.
- Like :
  - DateTimeFormatter.ISO_DATE_TIME.format(zonedDateTimeObject);
  - DateTimeFormatter.RFC_1123_DATE_TIME.format(zonedDateTimeObject);

Interoperate between Legacy Date API and new DateTime API :

(A) Instant & Date:

```
Date d = Date.from(instant);
Instant instant = d.toInstant();
```

(B) TimeStamp & Instant :

```
TimeStamp t = TimeStamp.from(instant);
Instant I = time.toInstant();
```

(C) LocalDate & Date:

```
Date d = Date.from(localDate);
LocalDate ld = d.toLocalDate()
```

(D) LocalTime & Time :

```
Time t = Time.from(localTime);
LocalTime localTime = t.toLocalTime();
```

Miscellaneous Upgrade:

1) ***Strings in Java 8 – new feature***

```
String s = “Hello World”;
IntStream stream = s.char(); // this method is introduced in java 8 to retrieve a stream.

Stream.mapToObject(letter-> (char)letter)
        .map(Character::toUpperCase)
        .forEach(System.out::println)
>> HELLO WORLD
```

2) ***StringJoiner – new feature:***

- Concatination of String is not that simple !
  - String s1 = “Hello”;
  - String s2 = “World”;
  - System.out.println(s1+” - ”+s2);

- But concatenation of String using + symbol is not considered efficient because of multiple intermediary operation.

- It was suggested to use String Buffer and StringBuilder class for this purpose but the process of joining the string has been made more simpler in Java 8 – StringJoiner

- A StringJoiner is built with a separator.

  - StringJoiner sj = new StringJoiner(“, ”);
  - Sj.add(“AAA”).add(“BBB”),add(”CCC”);
  - String s = sj.toString();
  - Sysout(s);

- StringJoiner has multiple constructors where developer can pass prefix postfix, etc

- Developer can use join() method of String class to achieve StringJoiner functionality.

3) ***Stream Operation Java File IO – new feature using stream***
4) ***Collection API and HashMap change***

New Method on Collection API –

i) Stream() & parallelStream()
ii) Sliterator() – it is used for parallel processing of collection.
iii) forEach() method in List class
iv) removeIf(it takes a Predicate here)
v) replaceAll(it takes a unary operator function here) -List
vi) list.sort(comparator object here) – List class

5) **New Static method in Comparator Interface**- *Which resolves the complexity of Comparing compare method implementation.*

   - Comparator<Person> compareLastName = Comparator.comparing(Person::getLastName);

   - Comparator chaining can be done using thenComparing() method – default method
   - Comparator<Person> compareLastNameThenFirstName = Comparator.comparing(Person::getLastName).thenComparing (Persion::getFirstName);
   - .naturalOrder(), reversed()- to reverse the chained comparator, nullFirst(),nullLast()

6) **New Method in Number Types:**
   - Sum(), max(), min()
   - Long.hasCode(elem);// saves the cost of autoboxing and unboxing

7) **New Method on Map:**
   - forEach(key,value); it takes a biconsumer as Parameter.
   - Map.get() was available but it could be confusing when a key has null as a value or the key is not present, as in both the cases null is returned, so to handle it new method introduced is – map.getOrDefault(key, person)
   - Map.putIfAbsent();
   - Map.replace(key,person);
   - Map.replaceAll(BiFunction);
   - Map.remove();
   - Compute(),computeIfPresent(),compute.IfAbsent()

- Merge(key,value, bifunction to merge the values if the key is present in both the map involved);

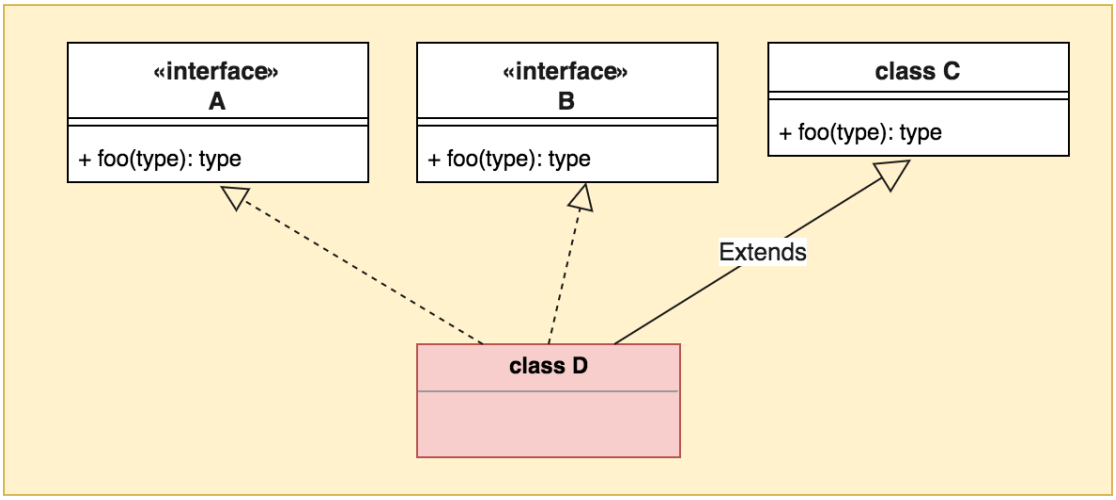
8) **Annotation can be repeated using @Repeatable Annotation even the position of placing annotation is also modified**

Diamond Problem of Inheritance in Java 8

Java 8 brought a major change where interfaces can provide default implementation for its methods. Java designers kept in mind the diamond problem of inheritance while making this big change. There are clearly defined conflict resolution rules while inheriting default methods from interfaces using Java 8.

Rule 1

Any method inherited from a class or a superclass is given higher priority over any default method inherited from an interface.

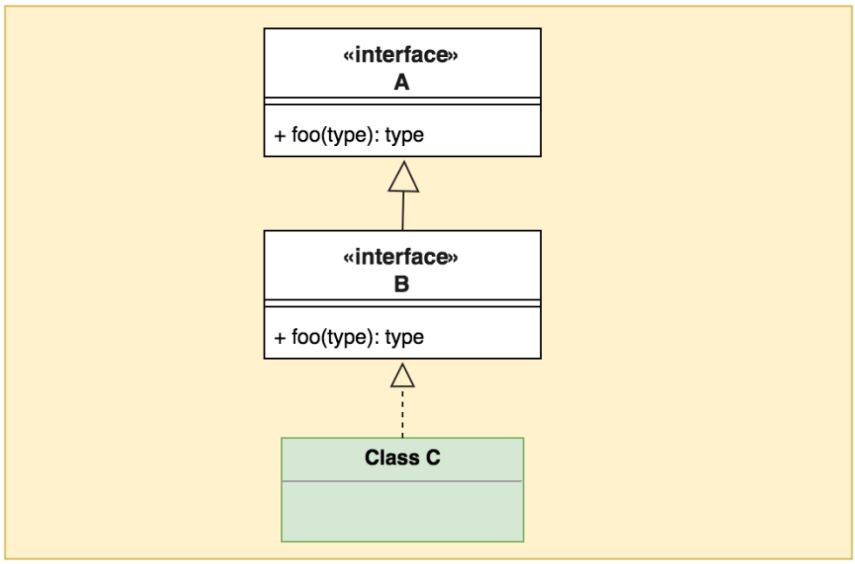


class has higher precedence than interface default methods.

In the diagram above, foo() method of class D will inherit from class C.

Rule 2

Derived interfaces or sub-interfaces take higher precedence than the interfaces higher-up in the inheritance hierarchy.

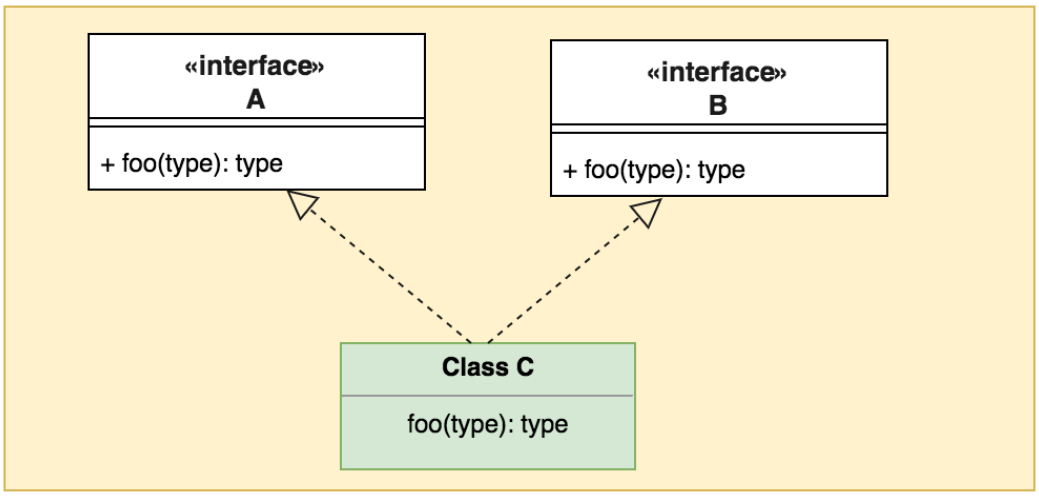


sub-interface has higher priority

In the above class diagram, foo() of class C will inherit from default method of interface B.

Rule 3

In case Rule 1 and Rule 2 are not able to resolve the conflict then the implementing class has to specifically override and provide a method with the same method definition.



class C must override foo method

In above class diagram, since interface A & B are at same level, to resolve conflict, class C must provide its own implementation by overriding method foo().

Rule 3

```
class C {
   void foo() {
      B.super.foo();
   }
}
```

foo() method can refer to A or B's default implementation using A.super.foo() or B.super.foo()

Question: what will be output of the below program?

Problem Statement

```
interface A {
   default void foo() { System.out.println("hello from A"); }
}
interface B extends A {
   default void foo() { System.out.println("hello from B"); }
}
interface C extends A {}
class D implements B, C {}


C c = new D();
c.foo();
```

Output

```
hello from B
```

The static type of c is unimportant here; what really counts is that it is an instance of class D, whose most specific version of foo() is inherited from class B.

How to Create an Immutable Class in Java

Here, we'll define the typical steps for creating an immutable class in Java and shed light on some common mistakes made while creating immutable classes.

An object is immutable if its state cannot change after construction. Immutable objects don't expose any way for other objects to modify their state; the object's fields are initialized only once inside the constructor and never change again.

In this article, we'll define the typical steps for creating an immutable class in Java and also shed light on the common mistakes which are made by developers while creating immutable classes.

1. Usage of Immutable Classes

Nowadays, the *"must-have"* specification for every software application is to be distributed and multi-threaded—multi-threaded applications always cause headaches for developers since developers are required to protect the state of their objects from concurrent modifications of several threads at the same time, for this purpose, developers normally use the *Synchronized* blocks whenever they modify the state of an object.

With immutable classes, states are never modified; every modification of a state results in a new instance, hence each thread would use a different instance and developers wouldn't worry about concurrent modifications.

2. Some Popular Immutable Classes

String is the most popular immutable class in Java. Once initialized its value cannot be modified. Operations like ***trim(), substring(), replace()*** always return a new instance and don't affect the current instance, that's why we usually call ***trim()*** as the following:

```
1
String alex = "Alex";
2
alex = alex.trim();
```

Another example from JDK is the wrapper classes like: ***Integer, Float, Boolean*** … these classes don't modify their state, however they create a new instance each time you try to modify them.

```
1
```

```
Integer a =3;
```

2

```
a += 3;
```

After calling ***a += 3,*** a new instance is created holding the value: 6 and the first instance is lost.

3. How Do We Create an Immutable Class

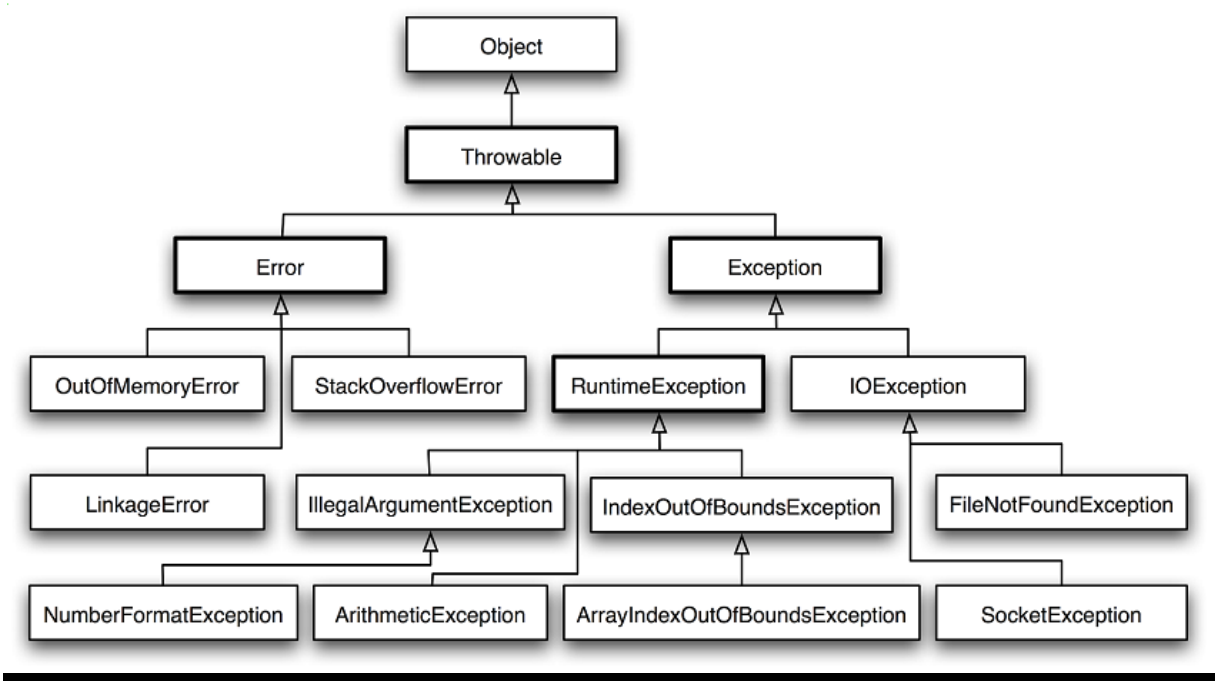
In order to create an immutable class, you should follow the below steps:

1. Make your class ***final,*** so that no other classes can extend it.
2. Make all your fields ***final,*** so that they're initialized only once inside the constructor and never modified afterward.
3. Don't expose setter methods.
4. When exposing methods which modify the state of the class, you must always return a new instance of the class.
5. If the class holds a mutable object:
    - Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.
    - Make sure to always return a clone copy of the field and never return the real object instance.

Reflection Concept?

- ***Overriding class declares a NPE? Yes it is possible as NPE is Runtime Exception***

Checked & Unchecked Exception



Precedence and order of loading spring bean.

Spring - Controlling Beans Loading Order by using @DependsOn

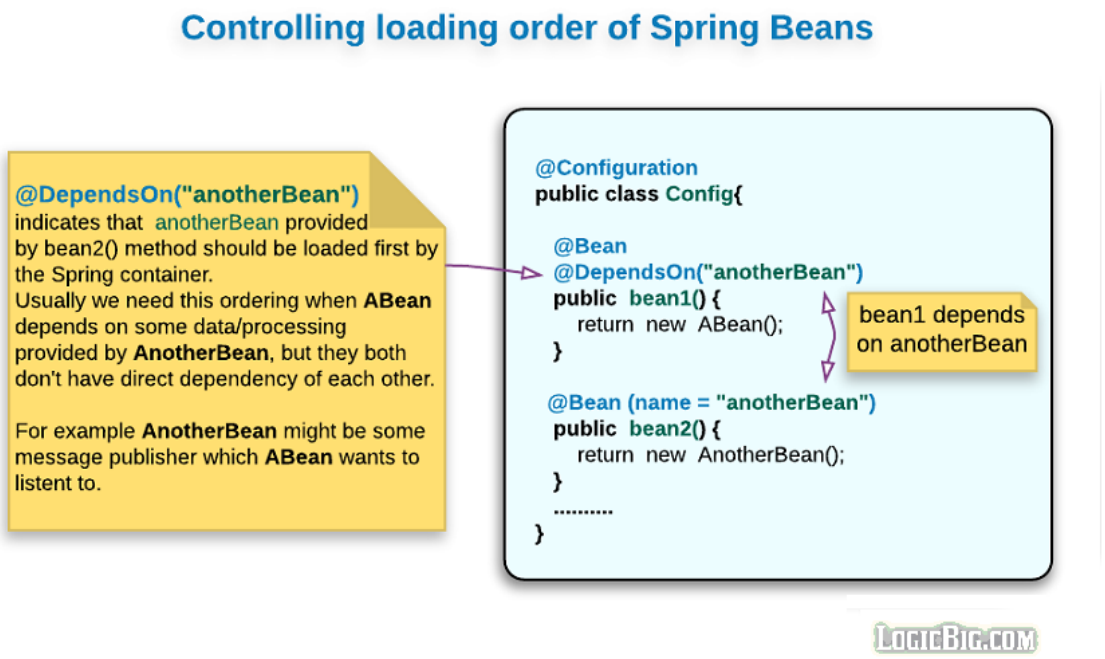
The order in which Spring container loads beans cannot be predicted. There's no specific ordering logi
framework. But Spring guarantees if a bean A has dependency of B (e.g. bean A has an instance variab
initialized first. But what if bean A doesn't have direct dependency of B and we still want B to initialize

When we want to control beans initializing order

There might be scenarios where A is depending on B indirectly. For example assume A is some kind of
those events. This is a typical scenario of observer pattern. We don't want B to miss any events and w
before A.

@DependsOn annotation

Spring provides `@DependsOn` indicating which bean should be initialized before this bean:



Example

Following example shows a very basic observer pattern.

EventManager, a facility to register listeners and publishing events

```
package com.logicbig.example.bean;

import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class EventManager {
 private final List<Consumer<String>> listeners = new ArrayList<>();

 @PostConstruct
 public void initialize() {
    System.out.println("initializing: "+this.getClass().getSimpleName());
 }

 public void publish(final String message) {
```

```
    listeners.forEach(l -> l.accept(message));
  }

  public void addListener(Consumer<String> eventConsumer) {
    listeners.add(eventConsumer);
  }
}
```

EventPublisher

```
package com.logicbig.example.bean;

import org.springframework.beans.factory.annotation.Autowired;
import javax.annotation.PostConstruct;

public class EventPublisher {
  @Autowired
  private EventManager eventManager;

  @PostConstruct
  public void initialize() {
    System.out.println("initializing: "+this.getClass().getSimpleName());
    eventManager.publish("event published from EventPublisherBean");
  }
}
```

EventListener

```
package com.logicbig.example.bean;

import org.springframework.beans.factory.annotation.Autowired;
import javax.annotation.PostConstruct;

public class EventListener {
  @Autowired
  private EventManager eventManager;
```

```
 @PostConstruct
 private void initialize() {
    System.out.println("initializing: "+this.getClass().getSimpleName());
    eventManager.addListener(s ->
          System.out.println("event received in EventListenerBean : " + s));

 }
}
```

Defining beans and running main class

```
package com.logicbig.example;

import com.logicbig.example.bean.EventListener;
import com.logicbig.example.bean.EventManager;
import com.logicbig.example.bean.EventPublisher;
import org.springframework.context.annotation.*;

@Configuration
@ComponentScan("com.logicbig.example")
public class AppConfig {

 @Bean
 @DependsOn("eventListenerBean")
 public EventPublisher eventPublisherBean() {
    return new EventPublisher();
 }

 @Bean
 public EventListener eventListenerBean() {
    return new EventListener();
 }

 @Bean
 public EventManager eventManagerBean() {
    return new EventManager();
```

```
 }

 public static void main(String... strings) {
    AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConf
    context.close();

 }
}
```

Output

```
initializing: EventManager
initializing: EventListener
initializing: EventPublisher
event received in EventListenerBean : event published from EventPublisherBean
```

If we don't use @DependsOn, there's no guarantee that EventListener will initialize first:

```
@Configuration
@ComponentScan("com.logicbig.example")
public class AppConfig {

  @Bean
 // @DependsOn("eventListenerBean")
  public EventPublisher eventPublisherBean() {
     return new EventPublisher();
  }

  @Bean
  public EventListener eventListenerBean() {
     return new EventListener();
  }

  @Bean
  public EventManager eventManagerBean() {
     return new EventManager();
```

```
    }

    public static void main(String... strings) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppCon
        context.close();

    }
}
```

Output

```
initializing: EventManager

initializing: EventPublisher

initializing: EventListener
```

@Order

It is used for Advice execution precedence.

The highest precedence advice runs first. The lower the number, the higher the precedence. For example, given two pieces of 'before' advice, the one with highest precedence will run first.

Another way of using it is for ordering Autowired collections

```
@Component
@Order(2)
class Toyota extends Car {
    public String getName() {
        return "Toyota";
    }
}

@Component
@Order(1)
class Mazda extends Car {
    public String getName() {
        return "Mazda";
```

```
    }
}

@Component
public class Cars {
    @Autowired
    List<Car> cars;

    public void printNames(String [] args) {

        for(Car car : cars) {
            System.out.println(car.getName())
        }
    }
}
```

You can find executable code here: https://github.com/patrikbego/spring-order-demo.git

Hope this clarifies it a little bit further.

Output:-

Mazda Toyota

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

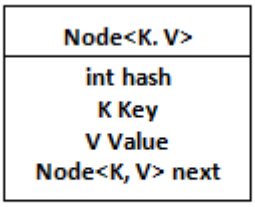


Figure: Representation of a Node

Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

- **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
- **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
- **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

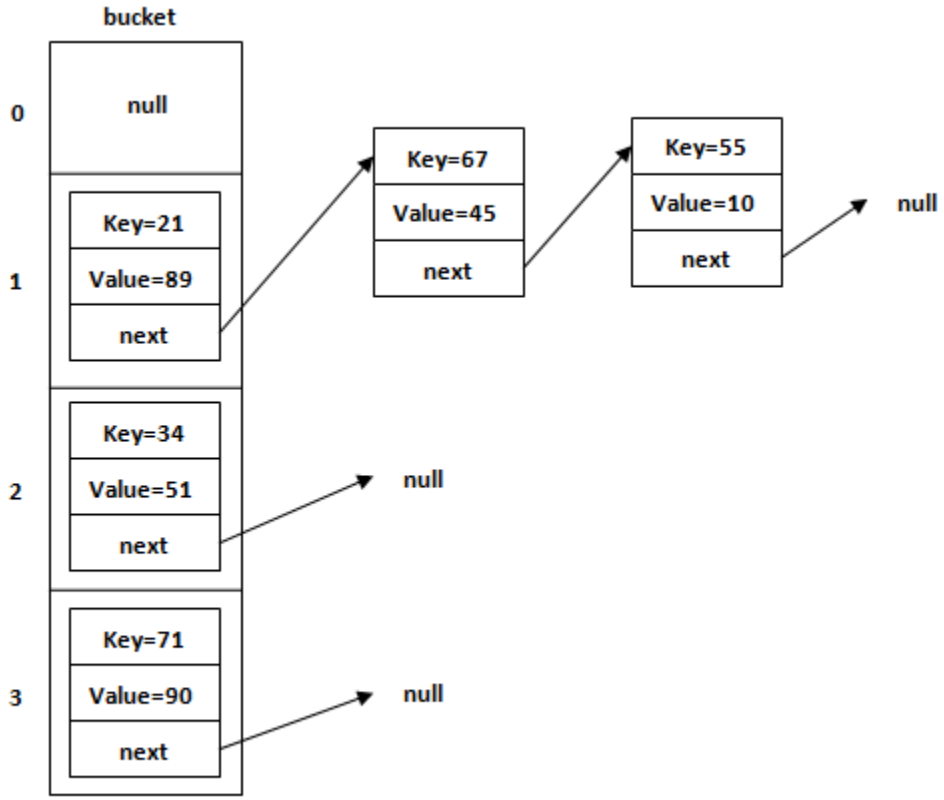


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

```
1. HashMap<String, Integer> map = new HashMap<>();
2. map.put("Aman", 19);
3. map.put("Sunny", 29);
4. map.put("Ritesh", 39);
```

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

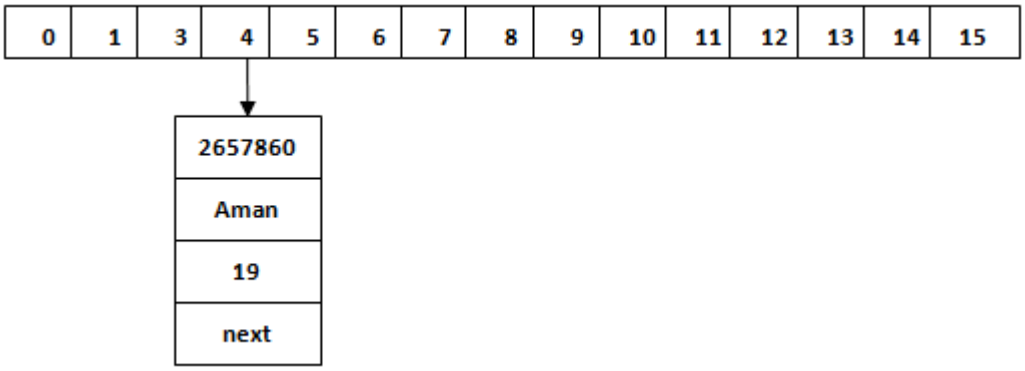
Index minimizes the size of the array. The Formula for calculating the index is:

1. Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

1. Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the Key and value will store in HashMap.



Hash Collision

This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.

General contract associated with hashCode() method

- The **hashCode()** method should return the same integer value for the same object for each calling of this method unless the value stored in the object is modified.

- If two objects are equal(according to **equals()** method) then the **hashCode()** method should return the same integer value for both the objects.
- But, it is not necessary that the **hashCode()** method will return the distinct result for the objects that are not equal (according to **equals()** method).

Difference between HashMap and ConcurrentHashMap

HashMap is the Class which is under Traditional Collection and ConcurrentHashMap is a Class which is under Concurrent Collections, apart from this there are various differences between them which are:

- HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
- HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
- While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying **ConcurrentModificationException**.Whereas In ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.
- In HashMap, null values are allowed for key and values, whereas in ConcurrentHashMap null value is not allowed for key and value, otherwise we will get Run-time exception saying **NullPointerException.**

Synchronized HashMap and ConcurrentHashMap

➔ Sync Hashmap has a bad performance where as ConcurrentHashMap is better in performance as Sync hashmap places lock on a complete object but in case of concurrenthashmap segment level lock is implemented i.e. the lock can be implemented at node level

Load Factor

The load factor (default load factor .75 of HashMap) 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.

- **In java 8 map implantation has changed internally to enhance the performance, now, java 8 uses TreeNode & Binary Search Tree for implementation to enhance the performance of HashMap.**

SOLID: The First 5 Principles of Object Oriented Design

SOLID stands for:

- **S** - Single-responsiblity Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Single-Responsibility Principle

Single-responsibility Principle (SRP) states:

A class should have one and only one reason to change, meaning that a class should have only one job.

Open-Closed Principle

Open-closed Principle (S.R.P.) states:

Objects or entities should be open for extension but closed for modification.

This means that a class should be extendable without modifying the class itself.

Liskov Substitution Principle

Liskov Substitution Principle states:

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

This means that every subclass or derived class should be substitutable for their base or parent class.

- **Interface Segregation Principle**

Interface segregation principle states:

A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.

- **Dependency Inversion Principle**

Dependency inversion principle states:

Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

This principle allows for decoupling.

String Constant Pool in Java

String is a sequence of characters. One of the most important characteristics of a string in Java is that they are immutable. In other words, once created, the internal state of a string remains the same throughout the execution of the program. This immutability is achieved through the use of a special string constant pool in the heap.

A string constant pool is a separate place in the heap memory where the values of all the strings which are defined in the program are stored. When we declare a string, an object of type String is created in the stack, while an instance with the value of the string is created in the heap. On standard assignment of a value to a string variable, the variable is allocated stack, while the value is stored in the heap in the string constant pool.

Var vs Let in JavaScript

The main difference between **let** and **var** is that scope of a variable defined with **let** is limited to the block in which it is declared while variable declared with **var** has the global scope. So we can say that **var** is rather a keyword which defines a variable globally regardless of block scope.

How can we maintain the immutability of a class with reflection in Java?

➔ Final class

Actually reflection is anti-OOP. It can be used to violate OOPs concept. Obviously it can break immutability too. Lets take an example.

Create an Immutable class.

```
1.  public final class ImmutableBean {
2.
3.      private final String name;
4.
5.      public ImmutableBean(String name) {
6.          this.name = name;
7.      }
8.
9.      public String getName() {
10.         return name;
11.     }
12.}
```

So once an object of ImmutableBean created, it can't be modified, but Reflection can do so.

```
1.  import java.lang.reflect.Field;
2.  import java.lang.reflect.ReflectPermission;
3.  import java.security.Permission;
4.
5.  public class ReflectionTest {
6.
7.      public static void main(String[] args) throws Exception {
8.
9.          // setSecuritymanager();
10.         ImmutableBean bean = new ImmutableBean("hemant");
11.         System.out.printf("old value: %s%n", bean.getName());
12.
13.         Field nameField = bean.getClass().getDeclaredField("name");
14.         nameField.setAccessible(true);
15.         nameField.set(bean, "hemant patel");
16.
17.         System.out.printf("new value: %s%n", bean.getName());
18.     }
19.}
```

ImmutableBean#name is a private final field, still its value has been modified. Output of the program is

```
1. old value: hemant
2. new value: hemant patel
```

To prevent it, we can take help of SecurityManager, define this method in ReflectionTest and uncomment its calling. This method prevents access check on private fields / methods via reflection.

```
1. private static void setSecuritymanager() {
2.
3.     System.setSecurityManager(new SecurityManager() {
4.         @Override
5.         public void checkPermission(Permission perm) {
6.             if (perm.getClass() == ReflectPermission.class &&
    "suppressAccessChecks".equals(perm.getName())) {
7.                 throw new SecurityException("can't supress AccessChecks");
8.             }
9.         }
10.    });
11.}
```

Now line nameField.setAccessible(true); throws exception.

How to print the linkedlist in reverse order without actually reversing the string?

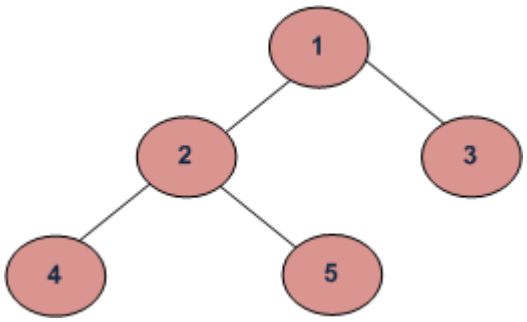
```
/* Function to print reverse of linked list */
void printReverse(Node head)
{
   if (head == null) return;

   // print list of head node
   printReverse(head.next);

   // After everything else is printed
   System.out.print(head.data+" ");
}
```

Write a Program to Find the Maximum Depth or Height of a Tree

Given a binary tree, find height of it. Height of empty tree is 0 and height of below tree is 3.



Algorithm:
maxDepth()

1. If tree is empty then return 0

2. Else

(a) Get the max depth of left subtree recursively i.e.,

call maxDepth(tree->left-subtree)

(a) Get the max depth of right subtree recursively i.e.,

call maxDepth(tree->right-subtree)

(c) Get the max of max depths of left and right

subtrees and add 1 to it for the current node.

max_depth = max(max dept of left subtree,

max depth of right subtree)

+ 1

(d) Return max_depth

See the below diagram for more clarity about execution of the recursive function maxDepth() for above example tree.

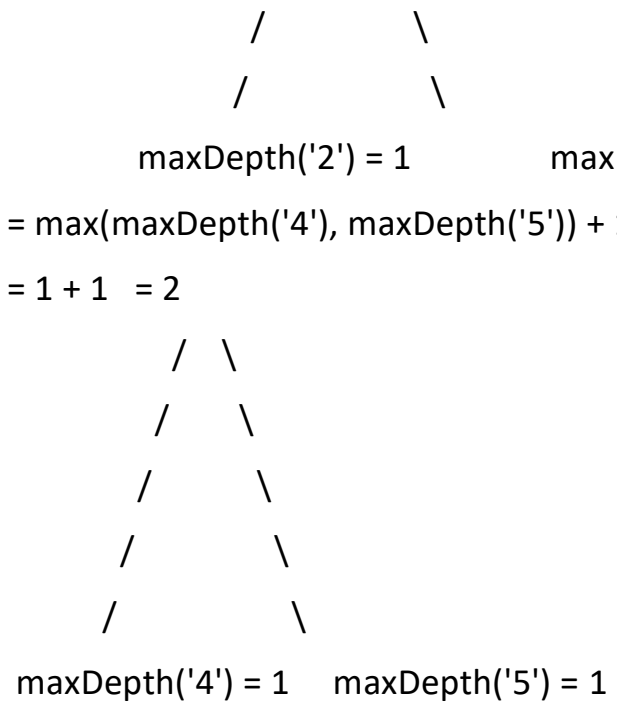
maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1

= 2 + 1

/ \

/ \

/ \




```
// Java program to find height of tree


// A binary tree node
class Node
{
    int data;
    Node left, right;


    Node(int item)
    {
```

```
        data = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    /* Compute the "maxDepth" of a tree -- the number of
       nodes along the longest path from the root node
       down to the farthest leaf node.*/
    int maxDepth(Node node)
    {
        if (node == null)
            return 0;
        else
        {
```

```
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}
/* Driver program to test above functions */
public static void main(String[] args)
{
    BinaryTree tree = new BinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
```

```
        tree.root.left.left = new Node(4);

        tree.root.left.right = new Node(5);

        System.out.println("Height of tree is : " +

                           tree.maxDepth(tree.root));

    }

}

// This code has been contributed by Mayank Jaiswal(mayank_24)
```

Output:
Height of tree is 3

Basic Thread Example

```
public class Demo extends Thread {

  public void run() {

    for (int i = 0; i < 5; i++) {

      System.out.println("The Thread name is " +
Thread.currentThread().getName());

    }

  }

  public static void main(String[] args) {

    Demo t1 = new Demo();

    t1.setName("Main Thread");

    t1.start();
```

```
    Thread t2 = currentThread();
    t2.setName("Current Thread");
    for (int i = 0; i < 5; i++) {
      System.out.println("The Thread name is " +
t1.currentThread().getName());
    }
  }
}
```

Runnable Interface

```
public class RunnableDemo {

    public static void main(String[] args)
    {
        System.out.println("Main thread is- "
                + Thread.currentThread().getName());
        Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());
        t1.start();
    }

    private class RunnableImpl implements Runnable {
```

```
        public void run()
        {
           System.out.println(Thread.currentThread().getName()
                        + ", executing run() method!");
        }
    }
}
```

Output:

Main thread is- main

Thread-0, executing run() method!

Select EMP with max SAL from each DEPT

Table Employee

name Salary Department_id
Hr/IT/Finance

```
select emp.did, emp.name,  emp.sal
from employee emp
inner join
(
select emp.deptno, max(emp.sal) sal
from employee
group by emp.deptno
) ss on emp.did = ss.did and emp.sal = ss.sal
```

More than one conditions can be added using on keyword

#Stream API Practice

```
List<Employee> list = new ArrayList<Employee>();
		list.add(new Employee(1, "A", 10000,"d1"));
		list.add(new Employee(2, "B", 20000,"d1"));
		list.add(new Employee(3, "C", 10000,"d3"));
		list.add(new Employee(4, "D", 20000,"d2"));
		list.add(new Employee(5, "E", 15000,"d4"));
		list.add(new Employee(6, "F", 20000,"d1"));
		list.add(new Employee(7, "G", 10000,"d3"));
		list.add(new Employee(8, "H", 20000,"d2"));
		list.add(new Employee(9, "I", 15000,"d4"));

		Map m = list.stream().collect(Collectors.toMap(Employee::getId,
Employee::getName));
		System.out.println(m);

		m= list.stream().collect(Collectors.groupingBy(Employee::getSal,
Collectors.counting()));
		System.out.println(m);
		m= list.stream().collect(Collectors.groupingBy(Employee::getSal,
Collectors.toList()));
		System.out.println(m);

		m= list.stream().collect(Collectors.groupingBy(Employee::getSal,
Collectors.mapping(Employee::getName,Collectors.toList())));
		System.out.println(m);



		m= list.stream().collect(Collectors.groupingBy(Employee::getSal,
Collectors.mapping(Employee::getName,Collectors.toList())));
		System.out.println(m);

		// map of (sal,map<dept,List of name>)

		m= list.stream().collect(Collectors.groupingBy(Employee::getSal,
Collectors.groupingBy(Employee::getDepartment,Collectors.mapping(Employe
e::getName, Collectors.toList()))));
		System.out.println(m);
```

```
		//BiFUnctionCoonsumer
		BiConsumer<Integer,	Integer> biConsumer = new
BiConsumer<Integer, Integer>() {
			@Override
			public void accept(Integer t, Integer u) {
				// TODO Auto-generated method stub
				int sum = t+u;
				System.out.println(sum);
			}
		};
		 m = list.stream().collect(Collectors.toMap(Employee::getId,
Employee::getSal));
		 m.forEach(biConsumer);
		 Iterator<Employee> i = list.iterator();
		 while(i.hasNext()){
			 System.out.println(i.next());
			 i.remove();
		 }
		 System.out.println(list);
```

Question: Print

```
1
2	5
3	7	11
4	9	14	19
```

```
int count =2;
for (int i = 0;i<4;i++){
	int elem =i+1;
	for (int j = 0;j<=i;j++){
		Sysout.print(elemj);
		elem+=count;
	}
	Sysout.println();
}
```

mention what are the HTTP methods supported by REST?

HTTP methods supported by REST are:

- **GET:** It requests a resource at the request URL. It should not contain a request body as it will be discarded. Maybe it can be cached locally or on the server.
- **POST:** It submits information to the service for processing; it should typically return the modified or new resource
- **PUT:** At the request URL it update the resource
- **DELETE:** At the request URL it removes the resource
- **OPTIONS:** It indicates which techniques are supported
- **HEAD:** About the request URL it returns meta information

Mention what is JAX-WS and JAX-RS?

- Both JAX-WS and JAX-RS are libraries (APIs) for doing communication in various ways in Java. JAX-WS is a library that can be used to do SOAP communication in JAVA, and JAX-RS lets you do the REST communication in JAVA.

Mention what is the difference between SOAP and REST?

SOAP	REST
• SOAP is a protocol through which two computer communicates by sharing XML document • SOAP permits only XML • SOAP based reads cannot be cached • SOAP is like custom desktop application, closely connected to the server • SOAP is slower than REST • It runs on HTTP but envelopes the message	• Rest is a service architecture and design for network-based software architectures • REST supports many different data formats • REST reads can be cached • A REST client is more like a browser; it knows how to standardized methods and an application has to fit inside it • REST is faster than SOAP • It uses the HTTP headers to hold meta information

Http-Status Code

1	**1xx: Informational** It means the request has been received and the process is continuing.
2	**2xx: Success** It means the action was successfully received, understood, and accepted.
3	**3xx: Redirection** It means further action must be taken in order to complete the request.
4	**4xx: Client Error** It means the request contains incorrect syntax or cannot be fulfilled.
5	**5xx: Server Error** It means the server failed to fulfill an apparently valid request.

Message	Description
200 OK	The request is OK.
201 Created	The request is complete, and a new resource is created .
202 Accepted	The request is accepted for processing, but the processing is not complete.
301 Moved Permanently	The requested page has moved to a new url .
302 Found	The requested page has moved temporarily to a new url .

400 Bad Request	The server did not understand the request.
401 Unauthorized	The requested page needs a username and a password.
403 Forbidden	Access is forbidden to the requested page.
404 Not Found	The server can not find the requested page.
405 Method Not Allowed	The method specified in the request is not allowed.
422 Unprocessable Entity	The HyperText Transfer Protocol (HTTP) **422 Unprocessable Entity** response status code indicates that the server understands the content type of the request entity, and the syntax of the request entity is correct, but it was unable to process the contained instructions. It is specific to malformed instruction. **Important**: The client should not repeat this request without modification.

Some of the Git Commands

- Git Clone : git clone <https://name-of-the-repository-link>
  Used for downloading the project from Github and maing a local clone of it in your machine.
- Git branch : git branch <branch-name>
  This command will create a branch **locally**. To push the new branch into the remote repository, you need to use the following command:
  git push -u <remote> <branch-name>
  Viewing branches:
  git branch or git branch --list
  Deleting a branch:
  git branch -d <branch-name>

- **Git checkout**

To work in a branch, first you need to switch to it. We use **git checkout** mostly for switching from one branch to another. We can also use it for checking out files and commits.
git checkout <name-of-your-branch>
There are some steps you need to follow for successfully switching between branches:

- The changes in your current branch must be committed or stashed before you switch
- The branch you want to check out should exist in your local

There is also a shortcut command that allows you to create and switch to a branch at the same time:

git checkout -b <name-of-your-branch>

This command creates a new branch in your local (-b stands for branch) and checks the branch out to new right after it has been created.

- Git status

The Git status command gives us all the necessary information about the current branch.

git status
We can gather information like:

- Whether the current branch is up to date
- Whether there is anything to commit, push or pull
- Whether there are files staged, unstaged or untracked
- Whether there are files created, modified or deleted
- **Git add**

When we create, modify or delete a file, these changes will happen in our local and won't be included in the next commit (unless we change the configurations).

We need to use the git add command to include the changes of a file(s) into our next commit.

To add a single file:
git add <file>
To add everything at once:
git add -A

6. Git commit

git commit -m "commit message"
Important: Git commit saves your changes only locally.

7. Git push

After committing your changes, the next thing you want to do is send your changes to the remote server. Git push uploads your commits to the remote repository.

git push <remote> <branch-name>
However, if your branch is newly created, then you also need to upload the branch with the following command:

git push --set-upstream <remote> <name-of-your-branch>
or

git push -u origin <branch_name>
Important: Git push only uploads changes that are committed.

8. Git pull

The **git pull** command is used to get updates from the remote repo. This command is a combination of **git fetch** and **git merge** which means that, when we use git pull, it gets the updates from remote repository (git fetch) and immediately applies the latest changes in your local (git merge).
git pull <remote>
This operation may cause conflicts that you need to solve manually.

9. Git revert

Sometimes we need to undo the changes that we've made. There are various ways to undo our changes locally or remotely (depends on what we need), but we must carefully use these commands to avoid unwanted deletions.

A safer way that we can undo our commits is by using **git revert**. To see our commit history, first we need to use **git log --oneline:**

```
Cem-MacBook-Pro:my-new-app cem$ git log --oneline
3321844 (HEAD -> master) test
e64e7bb Initial commit from Create React App
```

commit history of my master branch

Then we just need to specify the hash code next to our commit that we would like to undo:

git revert 3321844

After this, you will see a screen like below - just press **shift + q** to exit:

```
Revert "test"

This reverts commit 332184490ef2b5db289d85ed3f1a13ff2d5f94b9.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: Cem <cem@Cem-MacBook-Pro.local>
#
# On branch master
# Changes to be committed:
#       modified:   src/App.js
#       deleted:    src/components/myFirstComponent.js
#
~
~
~
~
~
~
~
~
~
~
"~/Desktop/my-new-app/.git/COMMIT_EDITMSG" 14L, 384C
```

The Git revert command will undo the given commit, but will create a new commit without deleting the older one:

```
Cem-MacBook-Pro:my-new-app cem$ git log --oneline
cd7fe6f (HEAD -> master) Revert "test"
3321844 test
e64e7bb Initial commit from Create React App
```

new "revert" commit

The advantage of using **git revert** is that it doesn't touch the commit history. This means that you can still see all of the commits in your history, even the reverted ones.

Another safety measure here is that everything happens in our local system unless we push them to the remote repo. That's why git revert is safer to use and is the preferred way to undo our commits.

10. Git merge

When you've completed development in your branch and everything works fine, the final step is merging the branch with the parent branch (dev or master). This is done with the git merge command.
Git merge basically integrates your feature branch with all of its commits back to the dev (or master) branch. It's important to remember that you first need to be on the specific branch that you want to merge with your feature branch.

For example, when you want to merge your feature branch into the dev branch:

First you should switch to the dev branch:
git checkout dev
Before merging, you should update your local dev branch:
git fetch
Finally, you can merge your feature branch into dev:
git merge <branch-name>
Hint: Make sure your dev branch has the latest version before you merge your branches, otherwise you may face conflicts or other unwanted problems.

different locking mechanisms in java

Difference between Object level lock and Class level lock in Java

---

In multithreading environment, two or more threads can access the shared resources simultaneously which can lead the inconsistent behavior of the system. Java uses concept of locks to restrict concurrent access of shared resources or objects. Locks can be applied at two levels –

- Object Level Locks – It can be used when you want non-static method or non-static block of the code should be accessed by only one thread.
- Class Level locks – It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime. It should always be used to make static data thread safe.

Sr. No.	Key	Object Level Lock	Class Level Lock
1	Basic	It can be used when you want non-static method or non-static block of the code should be accessed by only one thread	It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime
2	Static/Non Static	It should always be used to make non-static data thread safe.	It should always be used to make static data thread safe.
3	Number of Locks	Every object the class may have their own lock	Multiple objects of class may exist but there is always one class's class object lock available

Example of Class Level Lock

```
public class ClassLevelLockExample {

  public void classLevelLockMethod() {

    synchronized (ClassLevelLockExample.class) {

      //DO your stuff here

    }

  }

}
```

Example of Object Level Lock

```
public class ObjectLevelLockExample {

  public void objectLevelLockMethod() {

    synchronized (this) {

      //DO your stuff here

    }

  }

}
```

observable vs promises
CORS????

What is the N+1 query problem – left join fetch or via using Map or entity level(worst and not preffered)

routing between modules in angular

typescript vs javascript
typescript to javascript compilation is called as??
Jparepository
Crudrepository
And their difference
Synchronised hashmap
Concurrent

https://www.google.com/search?q=%2C%3D%3D+vs+%3D%3D%3D&rlz=1C1GCEU_enIN907IN908&oq=%2C%3D%3D+vs+%3D%3D%3D&aqs=chrome..69i57j0l2j0i20i263j0l6.1328j0j7&sourceid=chrome&ie=UTF-8

find the middle element of a linkedlist
What is BST?
How to check the given tree is BST??
Can a post call with request body can be converted to get call??
Path varriable vs request param
2 789 10 20 40
10 20 40 2 7 8 9

Suppose we have rotational array. How can we search for an element in the same in less than O(n).

```
                    12
          7                    18
    5          null      15          22
Null       6
    5.5          6.5
```

Is it a BST?

Properties of BST?

Transactional & transaction Level?

Transaction Propagation Level

REQUIRED
SUPPORTS
MANDATORY
REQUIRES_NEW
NOT_SUPPORTED

Connection Pooling

We would need connection pooling if application is being used by a multiple users. Multiple connections would be needed. Creating a connection every time is a very expensive operation so we should use connection pooling.

In a web application , application server can manage the connection pool but for standalone we have to rely on some third party solutions.

We can integrate Apache's connection pooling solutions (DBCP) or can use the C3P0 pooling framework with Hibernate. In fact Hibernate comes with C3P0.

Transaction Isolation Levels in DBMS

Prerequisite – Concurrency control in DBMS, ACID Properties in DBMS

As we know that, in order to maintain consistency in a database, it follows ACID properties. Among these four properties (Atomicity, Consistency, Isolation and Durability) Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.
Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena –

- **Dirty Read –** A Dirty read is the situation when a transaction reads a data that has not yet been committed. For example, Let's say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
- **Non Repeatable read –** Non Repeatable read occurs when a transaction reads same row twice, and get a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, Now if transaction T1 rereads the same data, it will retrieve a different value.
- **Phantom Read –** Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

Based on these phenomena, The SQL standard defines four isolation levels :

1. **Read Uncommitted –** Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.
2. **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

3. **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **Serializable –** This is the Highest isolation level.
   A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

The Table is given below clearly depicts the relationship between isolation levels, read phenomena and locks :

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Spring Filters and Filter Chaining

Spring Security's web infrastructure is based entirely on standard servlet filters. It doesn't use servlets or any other servlet-based frameworks (such as Spring MVC) internally, so it has no strong links to any particular web technology. It deals in HttpServletRequests and HttpServletResponses and doesn't care whether the requests come from a browser, a web service client, an HttpInvoker or an AJAX application.

Spring Filters are configured in web.xml

@Service vs @Component

Just like *@Repository, @Service* is another specialization of *@Component*:

```
1    @Component
2    public @interface Service {
3    }
```

Just like *@Repository, @Service* is also a type of *@Component*. That means Spring will also automatically detect such beans.
The ***@Service* annotation represents that our bean holds some business logic.** Till date, it doesn't provide any specific behavior over *@Component*.

@Repository vs @Component

@Repository annotation is a specialization over *@Component* annotation:

```
1    @Component
2    public @interface Repository {
3    }
```

Since *@Repository* is a type of *@Component*, Spring also auto-scans and registers them.

***@Repository* is a stereotype for the persistence layer. Its job is to catch all persistence related exceptions and rethrow them as a Spring *DataAccessException*.**

Handling of input and output format of REST API --- XML & JSON

Fail fast vs fail safe iterator.

Iterators in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw *ConcurrentModificationException* if there is **structural modification** of the collection. Structural modification means adding, removing any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.

Fail-Safe iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that's why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.

Optimistc Locking

Optimistic locking is when you check if the record was updated by someone else before you commit the transaction. Pessimistic **locking** is when you take an exclusive **lock** so that no one else can start modifying the record. Hibernate uses optimistic locking approach.

Hibernate Update vs Merge

Hibernate handles persisting any changes to objects in the session when the session is flushed. update can fail if an instance of the object is already in the session. Merge should be used in that case. It merges the changes of the detached object with an object in the session, if it exists.

Update: if you are sure that the session does not contains an already persistent instance with the same identifier,then use update to save the data in hibernate

Merge: if you want to save your modifications at any time with out knowing about the state of an session, then use merge() in hibernate.

Session.flush()

Flushing the session forces Hibernate to synchronize the in-memory state of the Session with the database (i.e. to write changes to the database). By default, Hibernate will flush changes automatically for you:

- before some query executions
- when a transaction is committed

Allowing to explicitly flush the Session gives finer control that may be required in some circumstances (to get an ID assigned, to control the size of the Session,...).

By calling flush() we force hibernate to execute the SQL commands on Database. But do understand that changesx are not "committed" yet. So after doing flush and before doing commit, if you access DB directly (say from SQL prompt) and check the modified rows, you will NOT see the changes.

This is same as opening 2 SQL command sessions. And changes done in 1 session are not visible to others until committed.

when we call session.flush() our statements are execute in database but not committed.

Suppose we don't call flush() method on session object and if we call commit method it will internally do the work of executing statements on the database and then committing.

commit=flush+commit (in case of functionality)

when we call method flush() on Session object, then it doesn't get commit but hits the database and executes the query and gets rollback too.

We should avoid save outside transaction boundary, otherwise mapped entities will not be saved causing data inconsistency. It’s very normal to forget flushing the session because it doesn’t throw any exception or warnings. By default, Hibernate will flush changes automatically for you: before some query

executions when a transaction is committed Allowing to explicitly flush the Session gives finer control that may be required in some circumstances

Thread Safe Singleton Class

```
public class SingletonClass {
	private static SingletonClass sc = null;

	private SingletonClass(){

		//sc= new SingletonClass();
	}

	public static  SingletonClass getSingletonClassObject(){
		if(sc ==null){
		synchronized(this){
			if(sc==null)
				sc= new SingletonClass();
			}
			return sc;
		} else {
			return sc;
		}
	}


}
```

Island of Isolation in Java

- Difficulty Level : Medium
- Last Updated : 07 Sep, 2016

In java, object destruction is taken care by the Garbage Collector module and the objects which do not have any references to them are eligible for garbage collection. Garbage Collector is capable to identify this type of objects.

Island of Isolation:

- Object 1 references Object 2 and Object 2 references Object 1. Neither Object 1 nor Object 2 is referenced by any other object. That's an island of isolation.
- Basically, an island of isolation is a group of objects that reference each other but they are not referenced by any active object in the

application. Strictly speaking, even a single unreferenced object is an island of isolation too.

Example:

```
public class Test

{

   Test i;

   public static void main(String[] args)

   {

      Test t1 = new Test();

      Test t2 = new Test();



      // Object of t1 gets a copy of t2

      t1.i = t2;



      // Object of t2 gets a copy of t1

      t2.i = t1;



      // Till now no object eligible

      // for garbage collection
```

```
        t1 = null;


        //now two objects are eligible for
        // garbage collection
        t2 = null;


        // calling garbage collector
        System.gc();


    }


    @Override
    protected void finalize() throws Throwable
    {
        System.out.println("Finalize method called");
    }
}
```

Some more Java Developer and basic javascript Questions/topics:

- observable vs promises
- CORS?

- **What is the N+1 query problem – left join fetch or via using Map or entity level(worst and not preffered)**

- routing between modules in angular

- typescript vs javascript
- typescript to javascript compilation is called as?
- Jparepository
- Crudrepository
- And their difference
- Synchronised hashmap
- Concurrent

- https://www.google.com/search?q=%2C%3D%3D+vs+%3D%3D%3D&rlz=1C1GCEU_enIN907IN908&oq=%2C%3D%3D+vs+%3D%3D%3D&aqs=chrome..69i57j0l2j0i20i263j0l6.1328j0j7&sourceid=chrome&ie=UTF-8

- find the middle element of a linkedlist
- What is BST?
- How to check the given tree is BST??
- Can a post call with request body be converted to get call?
- Path varriable vs request param
- Input -> 2 7 8 9 10 20 40
  Output-> 10 20 40 2 7 8 9

- Suppose we have rotational array. How can we search for an element in the same in less than O(n).
- 12
- 7 18
- 5 null 15 22
- Null 6
- 5.5 6.5

- Is it a BST?

- Properties of BST?
- Transactional & transaction Level?
- Controladvice?

To Be Continued…