PROGRAMMING INTERVIEW

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Figure: data structure operations

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Figure: array sorting algorithms

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Figure: graph operations

Type	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	-	O(1)	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))
Fibonacci Heap	-	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	O(1)

Figure: heap operations

It is a mistake when taken in the context that O(n) and O(n log n) functions have better complexity than O(1) and O(log n) functions. When looking typical cases of complexity in big O notation:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$$

Notice that this doesn't necessarily mean that they will always be better performance-wise - we could have an O(1) function that takes a long time to execute even though its complexity is unaffected by element count. Such a function

would look better in big O notation than an O(log n) function, but could actually perform worse in practice.

Generally speaking: a function with lower complexity (in big O notation) will outperform a function with greater complexity (in big O notation) *when n is sufficiently high*.

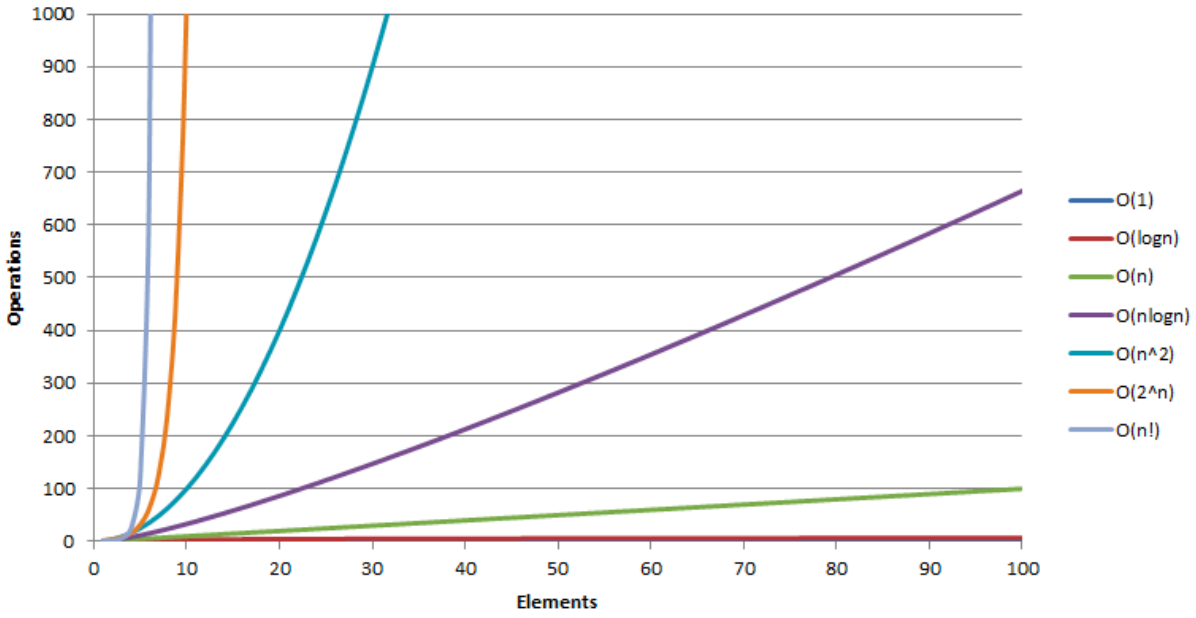


Figure: big-o complexity chart

When would you make a variable volatile?

How do you decide whether to use a HashMap or a TreeMap?

How do you decide whether to use a CopyOnWriteArrayList or a Collections.synchronizedList(ArrayList)?

For write (add) operation, CopyOnWriteArrayList uses ReentrantLock and creates a backup copy of the data and the underlying volatile array reference is only updated via setArray(Any read operation on the list during before setArray will return the old data before add).Moreover, CopyOnWriteArrayList provides snapshot fail-safe iterator and doesn't throw ConcurrentModificationException on write/ add.

But when I checked add method of CopyOnWriteArrayList.class, we are acquiring lock on complete collection object. Then how come CopyOnWriteArrayList is better than synchronizedList. The only difference I see in add method of CopyOnWriteArrayList is we are creating copy of that array each time add method get called.

1. No, the lock is not on the entire Collection object. As stated above it is a ReentrantLock and it is different from the intrinsic object lock.
2. The add method will always create a copy of the existing array and do the modification on the copy and then finally update the volatile reference of the array to point to this new array. And that's why we have the name "CopyOnWriteArrayList" - makes copy when you write into it.. This also avoids the ConcurrentModificationException

As per my understanding concurrent collection classes preferred over synchronized collection because concurrent collection classes don't take lock on complete collection object. Instead it takes lock on small segment of collection object.

This is true for some collections but not all. A map returned by Collections.synchronizedMap locks the entire map around every operation, whereas ConcurrentHashMap locks only

one hash bucket for some operations, or it might use a non-blocking algorithm for others.

For other collections, the algorithms in use, and thus the tradeoffs, are different. This is particularly true of lists returned by Collections.synchronizedList compared to CopyOnWriteArrayList. As you noted, both synchronizedList and CopyOnWriteArrayList take a lock on the entire array during write operations. So why are the different?

The difference emerges if you look at other operations, such as iterating over every element of the collection. The documentation for Collections.synchronizedList says,

It is imperative that the user manually synchronize on the returned list when iterating over it:

```
    List list = Collections.synchronizedList(new ArrayList());
    ...
    synchronized (list) {
        Iterator i = list.iterator(); // Must be in synchronized block
        while (i.hasNext())
            foo(i.next());
    }
```

Failure to follow this advice may result in non-deterministic behavior.

In other words, iterating over a synchronizedList is **not** thread-safe unless you do locking manually. Note that when using this technique, all operations by other threads on this list, including iterations, gets, sets, adds, and removals, are blocked. Only one thread at a time can do anything with this collection.

By contrast, the doc for CopyOnWriteArrayList says,

The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException. The

iterator will not reflect additions, removals, or changes to the list since the iterator was created.
Operations by other threads on this list can proceed concurrently, but the iteration isn't affected by changes made by any other threads. So, even though write operations lock the entire list, `CopyOnWriteArrayList` still can provide higher throughput than an ordinary `synchronizedList`. (Provided that there is a high proportion of reads and traversals to writes.)

How the ReentrantLock is different from the intrinsic object lock?

Java programming language supports multithreading or concurrency very well. Therefore, when we say multithreading or concurrent programming, it is necessary to have efficient synchronization or locking mechanisms, which saves concurrent program from unexpected concurrency issues like deadlocks. Reentrancy is one of those locking mechanism which help concurrent program to work as expected without any locking issue. Now, you want to know, what exactly this reentrancy do in Java? Let me explain you reentrancy or reentrant locking in Java, but before moving on it, I would like to tell you about, how Java threads take locks on objects and make operation synchronize or atomic. Therefore, before moving on reentrancy, I would like to explain intrinsic or monitor lock.

Intrinsic Or Monitor Lock Java provide built-in locking mechanism to make any operation atomic by synchronization in concurrent environment. In Java programming language

‘synchronized' keyword is used for taking lock or monitor on object. There are other locking utility also available in Java but we will concentrate on basic locking mechanism (synchronized) only. synchronized keyword can be use with method signature or with particular code block to make that synchronize. These built-in locks known as intrinsic or monitor lock.

Intrinsic Lock Examples There are few intrinsic or monitor lock examples mentioned below to make code synchronized.

1. Creating object level lock with method signature.

```
public synchronized void objectLevelLock() {
 // Code to make synchronized or atomic
}
```

2. Creating object level lock with synchronized block inside method.

```
public void objectLevelLock () {
 synchronized(lockObject) {
  // Code to make synchronized or atomic
 }
}
```

3. Creating class level lock on static members of class with method signature

```
public static synchronized void classLevelLock () {
 // Static members and Code to make synchronized or atomic
}
```

4. Creating class level lock on static members of class with synchronized block inside method.

```
public void classLevelLock () {
 synchronized(ClassName.class) {
  // Static members and Code to make synchronized or atomic
 }
}
```

Reentrant Lock Reentrancy is a locking mechanism provide by Java, which prevent Java locking from critical concurrency issue like deadlock . Suppose, there are two threads A and B. Thread B is trying to acquire lock or monitor on an object, which is already acquired by thread A. Thread B will be blocked until thread A release lock or monitor. But if thread A will try to acquire lock on other synchronized method or block on same object, it will be succeeded to acquire lock on same object. This facility is known as reentrancy. In other words, a thread who takes lock or monitor on object can be reenter any number of synchronized methods or blocks of same object on which it has already acquired lock. This is because, object locking is performed on *per thread basis*, not on *per invocation basis*.

What Could Be The Problem Without Reentrant Lock? In this section I would like to explain, If reentrancy wouldn't supported by Java, how it could affect thread execution and create concurrency issue. Please carefully go through the code given below.

```
public class Reentrancy {
```

```
 public synchronized void inner() {
  // Code to make synchronized or atomic
 }
 public synchronized void outer() {
  // Code to make synchronized or atomic
  inner();
 }
}
```

In this above code what will happen if reentrancy is not supported by Java. Suppose, a thread acquire lock on Reentrancy class object by invoking synchronized method outer(). But if you would notice outer() method invoking inner()method, which is also synchronized method. In such case, if Java wouldn't support reentrancy, outer() method couldn't succeeded to acquire lock on inner() method, because, it would be considered already locked and eventually result would be deadlock. But just because reentrancy is supported in Java, the thread who acquire lock on object can enter any synchronized method or block on same object to acquire lock.

How Reentrant Lock Works In Java? Reentrancy is implemented in Java by associating a counter with each thread. Initially counter initialized with value zero and considered as unlocked. When thread acquires lock on an object, counter get incremented by one. Again, if thread acquires lock on another synchronized method or block, counter again get incremented by one; counter will become two and so on. Same reverse process is followed, when thread release lock by leaving synchronized method or block. When thread releases lock from synchronized method or block; counter get decremented by one and so on. Once again, when counter

reached to zero; object gets unlocked. Now other threads are free to acquire lock on that object. This is the approach by which Java manage reentrancy.

Conclusion That's all how Java intrinsic and reentrant lock works. After reading this article, you are aware about, what are intrinsic and reentrant locks and how these locks works.

I am wondering is there big difference in using `ReentrantLock` and `synchronized (object)`.
The main differences are:

- With `synchronized` the locking / unlocking is tied to the block structure of your source code. A `synchronized` lock will be released when you exit the block, no matter how you do this. For instance, it will be released if the block terminates due to an unexpected exception.
  With explicit locking this is not the case, so you could acquire a `ReentrantLock` (or any other `Lock`) in one method and release it in another one. But the flip side is that you *must* remember to explicitly release the `Lock` at the appropriate time / place. If you don't you'll end up with a stuck lock, and (maybe) deadlocks. In short, `ReentrantLock` is more complicated and potentially more error prone.
- The primitive locking that you get with `synchronized` works
  with `Object.wait()` and `Object.notify()`. `Lock`s don't.
- A `ReentrantLock` can be created to be "fair", which means that threads that are waiting to acquire a given lock

will acquire the lock in fifo order. Primitive locks are not fair.
- The `ReentrantLock` API has methods that can be used to test whether the lock is in use, find out the length of the lock queue, try to acquire the lock without blocking, and various other things. None of this functionality is available for primitive locks.

Why it is called a reentrant lock? to permit recursive call from the same thread?

A reentrant lock allows a thread that is holding a lock to acquire it again. One of the ways that this could happen is through recursion, but there are others too.

For the record, `synchronized` locks are also reentrant, so you don't need to worry about recursion, or other scenarios where a thread might acquire a lock it already holds.

I hope this article will help you to understand about Java locking mechanism.

When would you use mergesort over quicksort?

When might recursion cause stack overflows?

Is Math.abs(Random.nextInt()) always positive?

What are the pros/cons of async vs. blocking I/O?

What unit tests would you write for Arrays.binarySearch?

What are anonymous classes? When, why, and how would you use them? Provide an example.

Anonymous classes are in-line expressions; often single-use classes for convenience that help make your code more concise. The following example instantiates a new ActionListener to handle events associated with a button:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        /* do something in response to button action event */
    }
});
```

This makes sense since the class isn't used elsewhere and doesn't need a name. However, if you pass an anonymous class to a registration method, for instance, you may want to keep track of its reference, in order to be able to unregister it later. Let's extend the above example to demonstrate this:

```
ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // do something in response
        // to button action event
    };
}

button.addActionListener(listener);

/* some time later... */
button.removeActionListener(listener);
```

What are abstract classes? When, why, and how would you use them? Provide an example.

Abstract classes are useful for defining abstract template methods that concrete subclasses must implement. All concrete subclasses are therefore guaranteed to honor the API specified by the abstract methods in the abstract class they inherit from. This is somewhat similar to the way in

which a Java interface specifies an API for all classes that implement it.

The common use case is where there is a category of objects that have a common behavior (e.g., all shapes have an area), but the details of calculating or performing those functions varies from one object to another. For example:

```
public abstract class Shape {

    public abstract double area();
}

public class Circle extends Shape {

    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return Math.PI * Math.pow(this.radius,2);
    }
}

public class Rectangle extends Shape {
    private double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() {
        return this.width * this.height;
    }
}
```

A couple of things worth noting:

- Abstract classes may not be instantiated directly; only their concrete subclasses are instantiable.
- A class may be declared abstract even if it has no abstract methods. This will preclude that class from being instantiated. This can be useful, for example, if a base class in a class hierarchy has no abstract methods but is not itself meant to be instantiated.

Compare and contrast checked and unchecked exceptions. Provide examples.

Unchecked exceptions are exceptions that are *not* considered to be recoverable. Java doesn't force you to catch or handle these because they indicate abnormal, unexpected problems with your code such as `NullPointerException`, `ArithmeticException` and `IndexOutOfBoundsException`. That is, these are problems you need to fix or prevent. Unchecked exceptions all derive from `RuntimeException`.

Checked exceptions are exceptions that *are* considered to be recoverable. Checked exceptions must explicitly be specified as part of a method's API; that is, a method that may throw one or more checked exceptions must list those potential exceptions as part of its method declaration (the Java compiler will actually enforce this).
When calling a method that throws exceptions, the caller must either handle (i.e., catch) those exceptions or must throw them itself. For example, if a method throws a checked exception, the caller might decide to ignore the error and continue (swallow it), display a dialog to the user, or rethrow the exception to let a method higher up the call chain handle it (in which case it must also declare that it throws the checked exception). For example:

```
public void readFile(File file) throws IOException, MyReadFileException {

    try {
        FileInputStream fis = new FileInputStream(file);
    }

    catch(FileNotFoundException e) {
        // We catch the FileNotFoundException and instead throw an IOException
        // so we don't include FileNotFoundException in our "throws" clause ab
        throw new IOException();
    }

    if (somethingBadHappened) {
        // We explicitly throw our own MyReadFileException here,
        // so we do include it in our "throws" clause above.
        throw new MyReadFileException();
    }
}
```

Checked exceptions clearly communicate and enforcing handling of error conditions. However, it can also be a pain for developers to continually need to include `try/catch` blocks to handle all known exceptions from the methods that they call. Although numerous checked exceptions are certainly permissible in Java, things can get a bit unwieldly. For example:

```
public void sillyMethod() throws DataFormatException,
                                 InterruptedException,
                                 IOException,
                                 SQLException,
                                 TimeoutException,
                                 ParseException {

    // some code
}
```

Accordingly, there has been raging debate for years on whether to use checked or unchecked exceptions when writing libaries, for example. As is true with many such debates, the truth is that there really is no one-size-fits-all, across-the-board correct answer. Checked and unchecked exceptions each have their own advantages and disadvantages, so the decision about which to use largely depends on the situation and context.

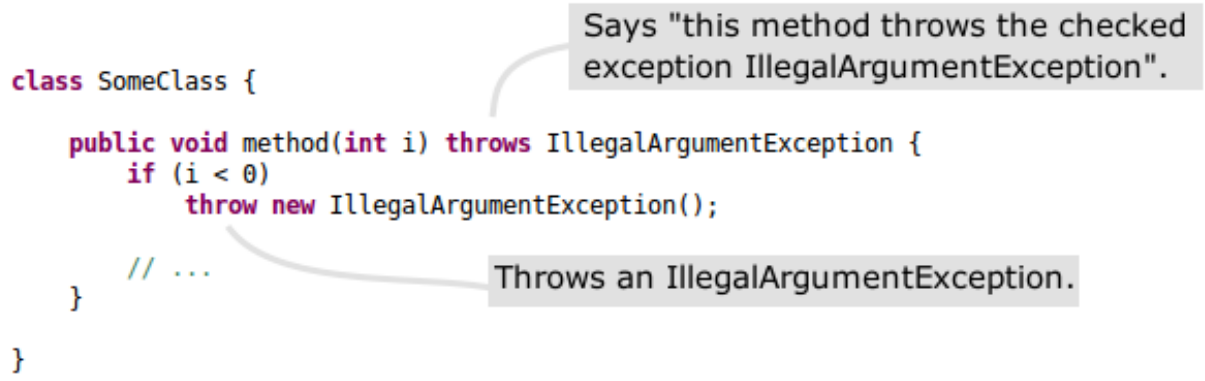
Exception handling: throw, throws and Throwable

- `throws`: Used when writing methods, to declare that the method in question throws the specified (checked) exception.
  As opposed to checked exceptions, runtime exceptions (NullPointerExceptions etc) may be thrown without having the method declare `throws NullPointerException`.

- `throw`: Instruction to actually throw the exception. (Or more specifically, the *Throwable*).

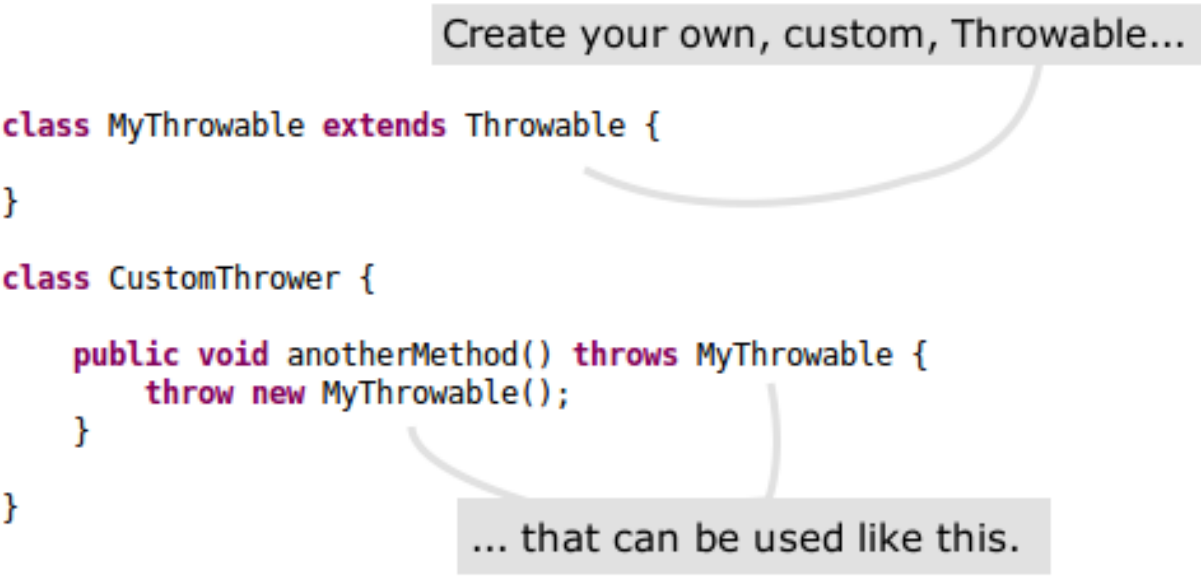
The `throw` keyword is followed by a reference to a `Throwable` (usually an exception).

Example:



- `Throwable`: A class which you must extend in order to create your own, custom, throwable.

Example:



StringBuilder/StringBuffer vs. "+" Operator

Using `String` concatenation is translated into `StringBuilder` operations by the compiler.
To see how the compiler is doing I'll take a sample class, compile it and decompile it with jad to see what's the generated bytecode.

Original class:

```
public void method1() {
    System.out.println("The answer is: " + 42);
}

public void method2(int value) {
    System.out.println("The answer is: " + value);
}

public void method3(int value) {
    String a = "The answer is: " + value;
    System.out.println(a + " what is the question ?");
}
```

The decompiled class:

```
public void method1()
{
    System.out.println("The answer is: 42");
}

public void method2(int value)
{
    System.out.println((new StringBuilder("The answer is: 
")).append(value).toString());
}

public void method3(int value)
{
    String a = (new StringBuilder("The answer is: 
")).append(value).toString();
```

```
    System.out.println((new
StringBuilder(String.valueOf(a))).append("   what   is   the
question ?").toString());
}
```

- On `method1` the compiler performed the operation at compile time.
- On `method2` the `String` concatenation is equivalent to manually use `StringBuilder`.
- On `method3` the `String` concatenation is definitely bad as the compiler is creating a second `StringBuilder` rather than reusing the previous one.

So my simple rule is that concatenations are good unless you need to concatenate the result again: for instance in loops or when you need to store an intermediate result.

Is it possible to have 2 unequal objects with the same hashCode?

Yes, that is certainly possible. Even if 2 objects are unequal, the hashCode can be the same. It is not required that if two objects are unequal according to the equals method, then calling the hashCode method on each of the two objects must produce distinct integer results. So there can exist two unequal objects with the same hashCode.

Describe Generics and provide examples of generic methods and classes in Java.

Java generics enable programmers to specify, with a single method or class declaration, functionality that can be applied to multiple different data types. Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time. Here, for example, is a **generic method** that uses `<E>` as the placeholder for a generic type:

```java
public <E> void printArray( E[] inputArray ) {
                    // Display array elements
    for ( E element : inputArray ) {

        System.out.printf( "%s ", element );
    }

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

The above method could then be invoked with various types of arrays and would handle them all appropriately; e.g.:

```java
// invoke generic printArray method with a Double array
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
printArray(doubleArray);

// invoke generic printArray method with a Character array
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
printArray(charArray);
```

There may be times, though, when you want to restrict the kinds of types that are allowed to be passed to a generic type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is accomplished in generic using a **bounded type parameter**, which list the type parameter's name followed by **extends** keyword. For example:

```java
// determines the largest of three Comparable objects
public static <T extends Comparable<T>> T maximum(T x, T y, T z) {

    T max = x; // assume x is initially the largest

    if ( y.compareTo( max ) > 0 ) {
        max = y; // y is the largest so far
    }

    if ( z.compareTo( max ) > 0 ) {
        max = z; // z is the largest now
    }

    return max; // returns the largest object
}
```

As with generic methods, the type parameter section of a **generic class** can have one or more type parameters separated by commas. For example:

```java
public class Cell<T> {

    private T val;

    public void set(T val) {
        this.val = val;
    }

    public T get() {
        return val;
    }

    public static void main(String[] args) {
        Cell<Integer> integerCell = new Box<Integer>();
        Cell<String> stringCell = new Box<String>();
        integerCell.add(new Integer(10));
        stringCell.add(new String("Hello World"));
        System.out.printf("Integer Value :%d\n\n", integerCell.get());
        System.out.printf("String Value :%s\n", stringCell.get());
    }
}
```

What is multiple inheritances? What are some potential problems with it and why has Java traditionally not supported it? How has this changed with the release of Java 8?

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class

can inherit characteristics and features from more than one parent object or parent class. It is distinct from single inheritance, where an object or class may only inherit from one particular object or class.
Until Java 8, Java only supported single inheritance. We'll discuss Java 8's quasi-support for multiple inheritance shortly, but first let's understand what problems can result from multiple inheritance and why it has been so heavily avoided in Java.

The main argument against multiple inheritance is the complexity, and potential ambiguity, that it can introduce. This is most typically exemplified by the commonly cited "diamond problem", whereby classes B and C inherit from class A, and class D inherits from both classes B and C. Ambiguity arises if there is a method in A that both B and C have overridden. If D does not override it, then which version of the method does it inherit; that of B, or that of C?

Let's consider a simple example. A university has people who are affiliated with it. Some are students, some are faculty members, some are administrators, and so on. So a simple inheritance scheme might have different types of people in different roles, all of whom inherit from one common "Person" class. The Person class could define an abstract `getRole()` method which would then be overridden by its subclasses to return the correct role type, i.e.:

But now what happens if we want to model a the role of a Teaching Assistant (TA)? Typically, a TA is *both* a grad student *and* a faculty member. This yields the classic diamond problem of multiple inheritance and the resulting ambiguity regarding the TA's `getRole()` method:

(Incidentally, note the diamond shape of the above inheritance diagram, which is why this is referred to as the "diamond problem".)

Which `getRole()` implementation should the TA inherit? Is that of the Faculty Member or that of the Grad Student? The simple answer might be to have the TA class override the `getRole()` method and return newly-defined role called "TA". But that answer is also imperfect as it would hide the fact that a TA is, in fact, both a faculty member and a grad student. There are multiple design approaches and patterns for addressing this type of situation without multiple inheritances, which is why some languages (Java being one of them) have made the decision to simply steer clear of multiple inheritances.

Java 8, however, introduces a form of quasi-support for multiple inheritances by allowing default methods to be specified on interfaces (prior to Java 8, only method signatures, not method definitions, were allowed on interfaces). Since Java *does* allow a single class to implement multiple interfaces (whereas a single class can only extend a single parent class), the allowance in Java 8 for method definitions in an interface introduces the potential for the diamond problem in Java for the first time.

For example, if A, B, and C are interfaces, B and C can each provide a different implementation to an abstract method of A, causing the diamond problem for any class D that implements B and C. Either class D must reimplement the method (the body of which can simply forward the call to one of the super implementations), or the ambiguity will be rejected as a compile error.

How can you exit a thread reliably using an external condition variable?

Sometimes developers want to terminate a thread when an external condition becomes true. Consider the following example of a `bus` thread that continues to drive indefinitely until the `pleaseStop` variable becomes true.

```
boolean pleaseStop = false; // The bus pull cord.

public void pleaseStopTheBus() {
    pleaseStop = true;
}

public void startTheBus() {

    new Thread("bus") {

        public void run() {

            // Infinitely drive the bus.
            while (!pleaseStop) {

                // Take some time driving to the next stop.
            }

            pleaseStop = false; // Reset pull cord.
        }

    }.start();
}
```

Seems straightforward. However, Java doesn't guarantee variable synchronization implicitly between thread boundaries, so this thread is not guaranteed to exit reliably, causing a lot of head scratching with less experienced Java developers.

Getting the above code to work properly would require synchronizing the threads as follows:

```java
boolean pleaseStop = false; // The bus pull cord.

Object driver = new Object(); // We can synchronize on any Java object.

public void pleaseStopTheBus() {

    // Here in "thread 1", synchronize on the driver object
    synchronized (driver) {
        pleaseStop = true;
    }
}

public void startTheBus() {

    new Thread("bus") {

        public void run() {

            // Infinitely drive the bus.
            while (true) {

                // And here in "thread 2", also synchronize on the driver object
                synchronized (driver) {

                    if (pleaseStop) {

                        pleaseStop = false; // Reset pull cord.
                        return; // Bus stopped.
                    }
                }

                // Take some time driving to the next stop.
            }
        }
    }.start();
}
```

How can `null` be problematic and how can you avoid its pitfalls?

For one thing, `null` is often ambiguous. It might be used to indicate success or failure. Or it might be used to indicate absence of a value. Or it might actually be a valid value in some contexts.
And even if one knows the meaning of `null` in a particular context, it can still cause trouble if the hapless developer forgets to check for it before de-referencing it, thereby triggering a **`NullPointerException`**.

One of the most common and effective techniques for avoiding these issues is to **use meaningful, non-null defaults.** In other words, simply avoid using `null` to the extent that you can. Avoid setting variables to `null` and avoid returning `null` from methods whenever possible (e.g., return an empty list rather than `null`).

In addition, as of JDK 8, Java has introduced support for the `Optional<T>` class (or if you're using an earlier version of Java, you can use the `Optional<T>` class in the Guava libraries. `Optional<T>` represents and wraps absence and presence with a value. While Optional adds a bit more *ceremony* to your code, by forcing you to unwrap the `Optional` to obtain the non-`null` value, it avoids what might otherwise result in a `NullPointerException`.

What is "boxing" and what are some of its problems to beware of?

Java's primitive types are `long`, `int`, `short`, `float`, `double`, `char`, `byte` and `boolean`. Often it's desirable to store primitive values as objects in various data structures that only accept objects such as `ArrayList`, `HashMap`, etc. So Java introduced the concept of "boxing" which boxes up primitives into object class equivalents, e.g., `Integer` for `int`, `Float` for `float`, and `Boolean` for `boolean`. Of course, as objects, they incur the overhead of object allocation, memory bloat and method calls, but they do achieve their purpose at some expense. "Autoboxing" is the automatic conversion by the compiler of primitives to boxed objects and vice versa. This is simply a convenience, e.g.:

```
ArrayList<Integer> ints = new ArrayList<Integer>();

// Autoboxing.  Compiler automatically converts "35" into a boxed Integer.
ints.add(35);

// So the above is equivalent to:
ints.add(new Integer(35));
```

Despite their convenience, though, boxed objects are notorious for introducing gnarly bugs, especially for less experienced Java developers. For one thing, consider this:

```
System.out.println(new Integer(5) == new Integer(5));   // false
```

In the above line of code, we are comparing the *identity* of two Integer objects. Since each `new Integer(5)` creates a new object, one `new Integer(5)` will not equal another `new Integer(5)`. But even more troubling is the following seemingly inexplicable distinction:

```
System.out.println(Integer.valueOf(127) == Integer.valueOf(127));   // true
System.out.println(Integer.valueOf(128) == Integer.valueOf(128));   // false
```

Huh? How can one of those be `true` and the other be `false`? That doesn't seem to make any sense. Indeed, the answer is quite subtle. As explained in an easily overlooked note in the Javadoc for the `Integer` class, the `valueOf()` method method caches Integer objects for values in the range -128 to 127, inclusive, and may cache other values outside of this range as well. Therefore, the Integer object returned by one call to `Integer.valueOf(127)` *will* match the Integer object returned by another call to `Integer.valueOf(127)`, since it is cached. But outside the range -128 to 127, `Integer.valueOf()` calls, even for the same value, will

not necessarily return the same Integer object (since they are not necessarily cached).

It's also important to note that computation using boxed objects can take around 6 times longer than using primitives, as can be evidenced by way of the following benchmarking code:

```
void sum() {
    Long sum = 0L; // Swap "Long" for "long" and speed dramatically improves.
    for (long i = 0; i <= Integer.MAX_VALUE; i++) {
        sum += i;
    }
}
```

Executing the above code with `sum` declared as `Long` took 6547ms whereas the same code with `sum` declared as `long` (i.e., the primitive type) took only 1138ms.

What is type erasure?

The addition of Generics to the language has not been without its problems. A particularly thorny issue with Java Generics is that of type erasure. As an example, consider the following code snippet:

```
List<String> a = new ArrayList<String>();
List<Integer> b = new ArrayList<Integer>();

return a.getClass() == b.getClass();  // returns true??!!
```

This should presumably return `false` since `a` and `b` are different class types (i.e., `ArrayList<String>` vs. `ArrayList<Integer>`), yet it returns `true`. Why?

The culprit here is type erasure. Once the above code passes all Java compiler validation, the compiler *erases* the `String` and `Integer` types in the above example, to maintain backward compatibility with older JDKs. The above code is therefore converted to the following by the Java compiler:

```
List a = new ArrayList(); List b = new ArrayList();
return a.getClass() == b.getClass();  // returns true (understandably)
```

And thus, in the compiled code, `a` and `b` are both simply untyped `ArrayList` objects, and the fact that one was an `ArrayList<String>` and the other was an `ArrayList<Integer>` is lost. Although in practice type erasure-related issues rarely cause problems for developers, it is an important issue to be aware of and can in certain cases lead to really gnarly bugs.

Describe the Observer pattern and how to use it in Java. Provide an example.

The Observer pattern lets objects sign up to receive notifications from an observed object when it changes. Java has built-in support with the `Observable` class and `Observer` interface. Here's a simple example of an implementation of an Observable:

```
public class Exhibitionist {

    MyObservable myObservable = new MyObservable();

    public Exhibitionist() {


    }

    public java.util.Observable getObservable() {
        return myObservable;
    }

    private void trigger(String condition) {
        myObservable.invalidate();
        myObservable.notifyObservers(condition);
    }

    private class MyObservable extends java.util.Observable {

        private void invalidate() {
            setChanged();
        }
    }
}
```

And here's a corresponding Observer example:

```
public class Voyeur implements Observer {

    public Voyeur(Exhibitionist exhibitionist) {

        // Register ourselves as interested in the Exhibitionist.
        exhibitionist.getObservable().addObserver(this);
    }

    @Override
    public void update(Observable o, Object arg) {
        // Called when the observable notifies its observers.
        System.out.println(arg.toString());
    }
}
```

There are a couple of downsides of using this though:

1. The observed class must extend Observable and thus prevents it from extending a more desirable class (refer to our earlier discussion of multiple inheritance)

2. Observed and observer classes are tightly coupled causing potential for NullPointerException's if you are not careful.

To circumvent the first issue, an advanced developer can use a proxy (delegate) Observable object instead of extending it. To address the second issue, one can use a loosely coupled publish-subscribe pattern. For example, you might use Google's Guava Library EventBus system where objects connect to a middleman.

Describe strong, soft, and weak references in Java. When, why, and how would you use each?

In Java, when you allocate a new object and assign its reference (simply a pointer) to a variable, a **strong reference** is created by default; e.g.:

```
String name = new String(); // Strong reference
```

There are, however, two additional reference strengths in Java that you can specify explicitly: **SoftReference** and **WeakReference**. (There is actually one additional reference strength in Java which is known as a **PhantomReference**. This is so rarely used, though, that even highly experienced Java developers will most likely not be familiar with it, so we omit it from our discussion here.)

Why are soft and weak references needed and when are they useful?

Java's garbage collector (GC) is a background process that periodically runs to free "dead" objects (one's without strong references) from your application's memory heap. Although the GC sometimes gives the impression of being a magical black box, it really isn't that magical after all.

Sometimes you have to help it out to prevent memory from filling up.

More specifically, the GC won't free objects that are strongly reachable from a chain of strongly referenced objects. What that simply means is that, if the GC still thinks an object is needed, it leaves it alone, which is normally what you want (i.e., you don't want an object you need to suddenly disappear when the GC kicks in).

But, sometimes strong references are too strong which is where soft and weak references can come in handy. Specifically:

- **SoftReference** objects are cleared at the discretion of the garbage collector in response to memory demand. Soft references are most often used to implement memory-sensitive caches.

- **WeakReference** objects do not prevent their referents from being made finalizable, finalized, and then reclaimed. Weak references are most often used to implement canonicalized mappings.

Java continues to survive and thrive as a dominant and popular language. But as might be expected with any language that has gone through so many iterations, it has many nuances and subtleties that not all developers are familiar with. Moreover, the ever-increasing breadth of Java's capabilities requires a great deal of experience to fully appreciate. Those who have mastered the language can therefore have a significant positive impact on your team's productivity and your system's performance, scalability, and stability.

The questions and tips presented herein can be valuable aids in identifying true Java masters. We hope you find them to be a useful foundation for “separating the wheat from the chaff” in your quest for the elite few among Java developers. Yet it is important to remember that these are merely intended as tools to be incorporated into the larger context of your overall recruiting toolbox and strategy.

Does Java is a pass-by-reference or pass-by-value?

That's sort of a trick question. Everything is passed by value, but objects are passed by reference, and those references are passed by value. You might think that that's pedantic, but it's really not. For example:

```
public void passByReferenceCheck(SomeObject object){
    object = new SomeObject()
}

public void passByValueCheck(SomeObject object){
    object.someMember = SOME_VALUE;
}
```

If Java were pass-by-reference then the outer scope of "object" would reflect the new object creation in "passByReferenceCheck". If Java passed objects by value then the outer scope of "object" would not reflect the change in "passByValueCheck".

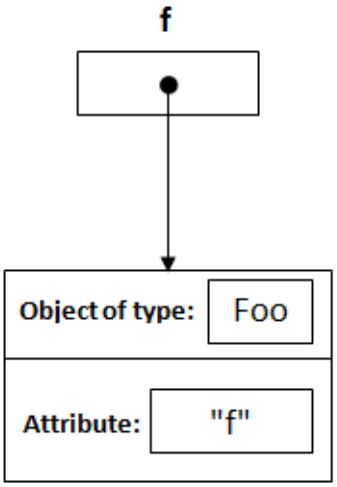
Java always passes arguments by value NOT by reference.

```
public class Main
{
     public static void main(String[] args)
     {
          Foo f = new Foo("f");
          changeReference(f); // It won't change the reference!
          modifyReference(f); // It will change the object that the reference variabl
     }
     public static void changeReference(Foo a)
     {
          Foo b = new Foo("b");
          a = b;
     }
     public static void modifyReference(Foo c)
     {
          c.setAttribute("c");
     }
}
```

I will explain this in steps:

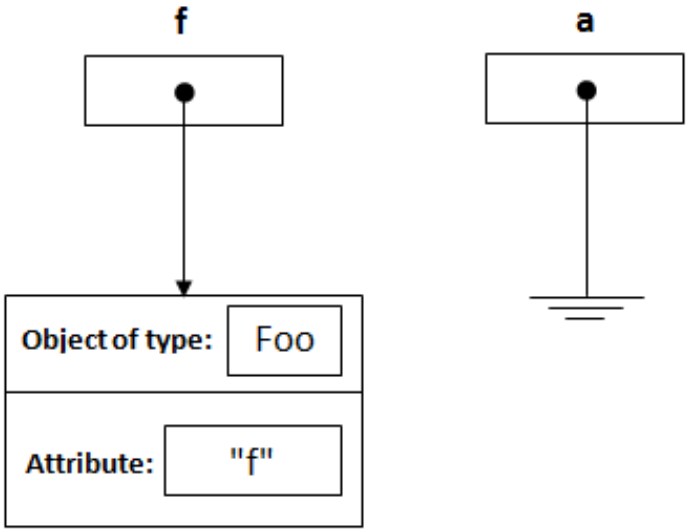
1- Declaring a reference named `f` of type `Foo` and assign it to a new object of type `Foo` with an attribute `"f"`.

```
Foo f = new Foo("f");
```



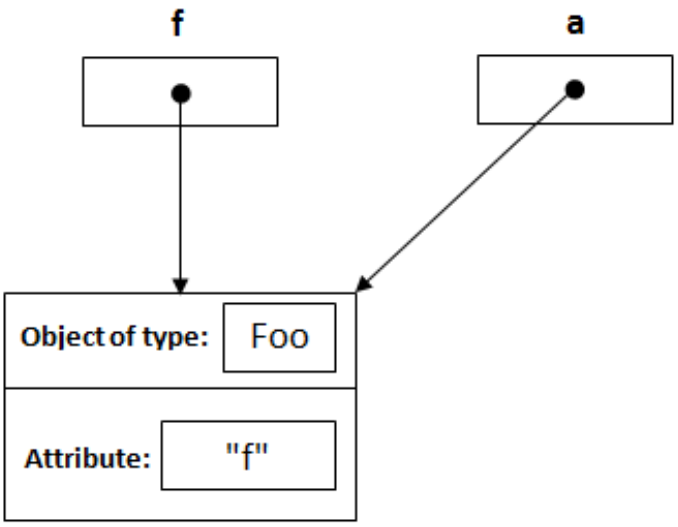
2- From the method side, a reference of type `Foo` with a name `a` is declared and it's initially assigned to `null`.

```
public static void changeReference(Foo a)
```



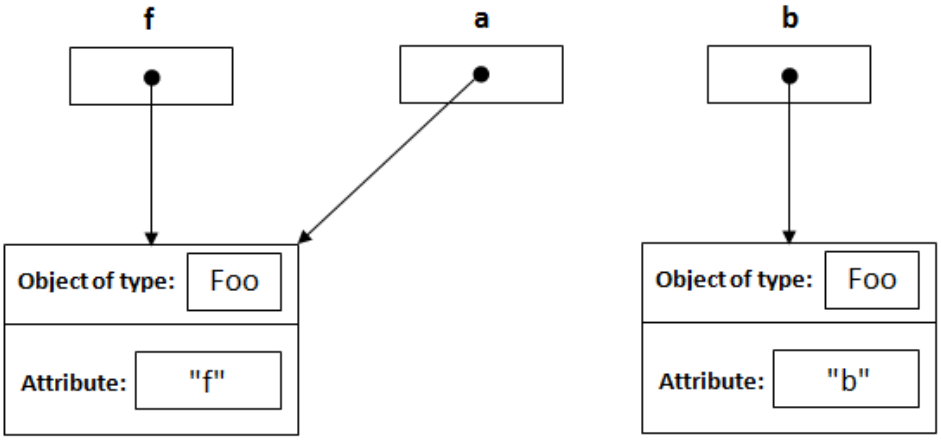
3- As you call the method changeReference, the reference a will be assigned to the object which is passed as an argument.

```
changeReference(f);
```

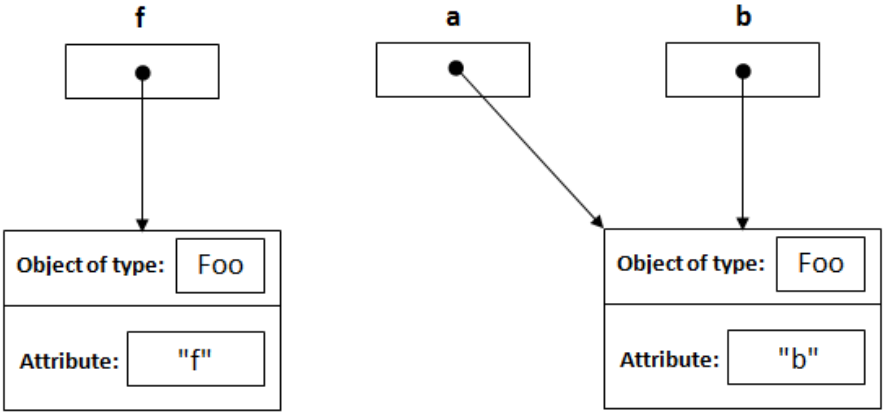


4- Declaring a reference named b of type Foo and assign it to a new object of type Foo with an attribute "b".

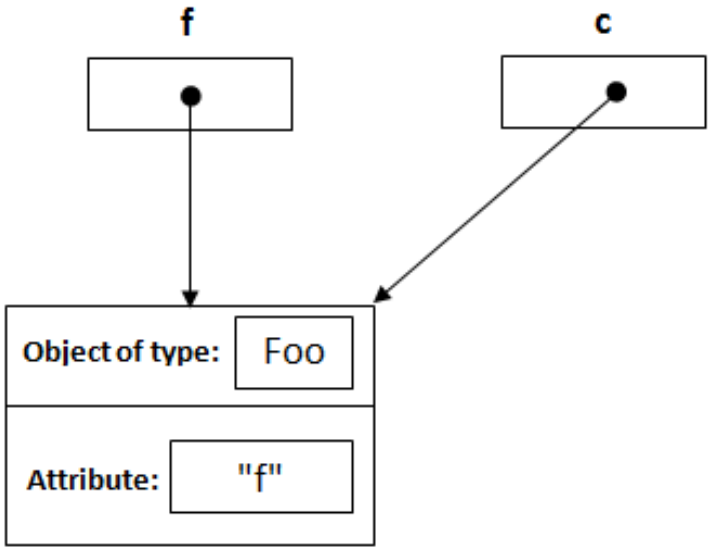
```
Foo b = new Foo("b");
```



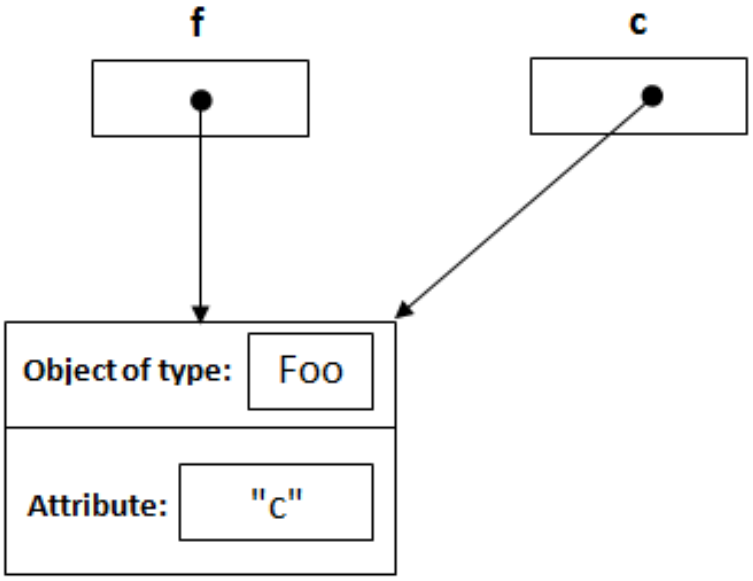
5- `a = b` is re-assigning the reference `a` NOT `f` to the object whose its attribute is `"b"`.



6- As you call `modifyReference(Foo c)` method, a reference `c` is created and assigned to the object with attribute `"f"`.



7- `c.setAttribute("c");` will change the attribute of the object that reference `c` points to it, and it's same object that reference `f` points to it.



Describe and compare fail-fast and fail-safe iterators.

The main distinction between fail-fast and fail-safe iterators is whether or not the collection can be modified while it is being iterated. Fail-safe iterators allow this; fail-fast iterators do not.

Fail-fast iterators operate directly on the collection itself. During iteration, fail-fast iterators fail as soon as they realize that the collection has been modified (i.e., upon realizing that a member has been added, modified, or removed) and will throw a ConcurrentModificationException. Some examples include ArrayList, HashSet, and HashMap (most JDK1.4 collections are implemented to be fail-fast).

Fail-safe iterates operate on a cloned copy of the collection and therefore do not throw an exception if the collection is modified during iteration. Examples would include iterators returned by ConcurrentHashMap or CopyOnWriteArrayList.

ArrayList, LinkedList, and Vector are all implementations of the List interface. Which of them is most efficient for adding and removing elements from the list? Explain your answer, including any other alternatives you may be aware of.

Of the three, `LinkedList` is generally going to give you the best performance. Here’s why:

`ArrayList` and `Vector` each use an array to store the elements of the list. As a result, when an element is inserted into (or removed from) the middle of the list, the

elements that follow must all be shifted accordingly. Vector is synchronized, so if a thread-safe implementation is *not* needed, it is recommended to use ArrayList rather than Vector.

LinkedList, on the other hand, is implemented using a doubly linked list. As a result, an inserting or removing an element only requires updating the links that immediately precede and follow the element being inserted or removed.

However, it is worth noting that if performance is that critical, it's better to just use an array and manage it yourself, or use one of the high performance 3rd party packages such as Trove or HPPC.

Why would it be more secure to store sensitive data (such as a password, social security number, etc.) in a character array rather than in a String?

In Java, Strings are immutable and are stored in the String pool. What this means is that, once a String is created, it stays in the pool in memory until being garbage collected. Therefore, even after you're done processing the string value (e.g., the password), it remains available in memory for an indeterminate period of time thereafter (again, until being garbage collected), which you have no real control over. Therefore, anyone having access to a memory dump can potentially extract the sensitive data and exploit it.

In contrast, if you use a mutable object like a character array, for example, to store the value, you can set it to blank once you are done with it with confidence that it will no longer be retained in memory.

Immutability of Strings in Java

Consider the following example.

```
String str = new String();

str  = "Hello";
System.out.println(str);  //Prints Hello

str = "Help!";
System.out.println(str);  //Prints Help!
```

Now, in Java, String objects are immutable. Then how come the object `str` can be assigned value "Help!". Isn't this contradicting the immutability of strings in Java? Can anybody please explain me the exact concept of immutability?
Edit:

Ok. I am now getting it, but just one follow-up question. What about the following code:

```
String str = "Mississippi";
System.out.println(str); // prints Mississippi

str = str.replace("i", "!");
System.out.println(str); // prints M!ss!ss!pp!
```

Does this mean that two objects are created again ("Mississippi" and "M!ss!ss!pp!") and the reference `str` points to a different object after `replace()` method?

`str` is not an object, it's a reference to an object. `"Hello"` and `"Help!"` are two distinct `String`objects. Thus, `str` *points to* a string. You can change what it *points to*, but not that which it *points at*.
Take this code, for example:

```
String s1 = "Hello";
String s2 = s1;
// s1 and s2 now point at the same string - "Hello"
```

Now, there is nothing[1] we could do to `s1` that would affect the value of `s2`. They refer to the same object - the string `"Hello"` - but that object is immutable and thus cannot be altered.

If we do something like this:

```
s1 = "Help!";
System.out.println(s2); // still prints "Hello"
```

Here we see the difference between mutating an object, and changing a reference. `s2` still points to the same object as we initially set `s1` to point to. Setting `s1` to `"Help!"` only changes the *reference*, while the `String` object it originally referred to remains unchanged.

If strings *were* mutable, we could do something like this:

```
String s1 = "Hello";
String s2 = s1;
s1.setCharAt(1, 'a'); // Fictional method that sets character at a given pos in string
System.out.println(s2); // Prints "Hallo"
```

Edit to respond to OP's edit:

If you look at the source code for String.replace(char,char) (also available in src.zip in your JDK installation directory -- a pro tip is to look there whenever you wonder how something really works) you can see that what it does is the following:

- If there is one or more occurrences of `oldChar` in the current string, make a copy of the current string where all occurrences of `oldChar` are replaced with `newChar`.
- If the `oldChar` is not present in the current string, return the current string.

So yes, `"Mississippi".replace('i', '!')` creates a new `String` object. Again, the following holds:

```
String s1 = "Mississippi";
String s2 = s1;
```

```
s1 = s1.replace('i', '!');
System.out.println(s1); // Prints "M!ss!ss!pp!"
System.out.println(s2); // Prints "Mississippi"
System.out.println(s1 == s2); // Prints "false" as s1 and s2 are two different objects
```

Your homework for now is to see what the above code does if you change `s1 = s1.replace('i', '!');` to `s1 = s1.replace('Q', '!');` :)

[1] Actually, it *is* possible to mutate strings (and other immutable objects). It requires reflection and is very, very dangerous and should never ever be used unless you're actually interested in destroying the program.

What is the ThreadLocal class? How and why would you use it?

A single `ThreadLocal` instance can store different values for each thread independently. Each thread that accesses the `get()` or `set()` method of a `ThreadLocal` instance is accessing its own, independently initialized copy of the variable. `ThreadLocal` instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or transaction ID). The example below, from the `ThreadLocal Javadoc`, generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes `ThreadId.get()` and remains unchanged on subsequent calls.

```java
public class ThreadId {
    // Next thread ID to be assigned
    private static final AtomicInteger nextId = new AtomicInteger(0);

    // Thread local variable containing each thread's ID
    private static final ThreadLocal<Integer> threadId =
        new ThreadLocal<Integer>() {
            @Override protected Integer initialValue() {
                return nextId.getAndIncrement();
        }
    };

    // Returns the current thread's unique ID, assigning it if necessary
    public static int get() {
        return threadId.get();
    }
}
```

Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).

What is the volatile keyword? How and why would you use it?

In Java, each thread has its own stack, including its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables into its own stack. The `volatile` keyword basically says to the JVM "Warning, this variable may be modified in another Thread".

In all versions of Java, the `volatile` keyword guarantees global ordering on reads and writes to a variable. This implies that every thread accessing a volatile field will read the variable's current value instead of (potentially) using a cached value.

In Java 5 or later, `volatile` reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex.
Using `volatile` may be faster than a lock, but it will not work in some situations. The range of situations in which volatile is effective was expanded in Java 5; in particular, double-checked locking now works correctly.

The volatile keyword is also useful for 64-bit types like long and double since they are written in two operations. Without the volatile keyword you risk stale or invalid values.

One common example for using `volatile` is for a flag to terminate a thread. If you've started a thread, and you want to be able to safely interrupt it from a different thread, you can have the thread periodically check a flag (i.e., to stop it, set the flag to `true`). By making the flag volatile, you can ensure that the thread that is checking its value will see that it has been set to `true` without even having to use a synchronized block. For example:

```
public class Foo extends Thread {
    private volatile boolean close = false;
    public void run() {
        while(!close) {
            // do work
        }
    }
    public void close() {
        close = true;
        // interrupt here if needed
    }
}
```

Compare the sleep() and wait() methods in Java, including when and why you would use one vs. the other

`sleep()` is a blocking operation that keeps a hold on the monitor / lock of the shared object for the specified number of milliseconds.

`wait()`, on the other hand, simply *pauses* the thread until *either* (a) the specified number of milliseconds have elapsed *or* (b) it receives a desired notification from another thread (whichever is first), *without* keeping a hold on the monitor/lock of the shared object.

`sleep()` is most commonly used for polling, or to check for certain results, at a regular interval. `wait()` is generally used in multithreaded applications, in conjunction with `notify()` / `notifyAll()`, to achieve synchronization and avoid race conditions.

Tail recursion is functionally equivalent to iteration. Since Java does not yet support tail call optimization, describe how to transform a simple tail recursive function into a loop and why one is typically preferred over the other.

Here is an example of a typical recursive function, computing the arithmetic series 1, 2, 3…N. Notice how the addition is performed after the function call. For each recursive step, we add another frame to the stack.

```java
public int sumFromOneToN(int n) {
  if (n < 1) {
    return 0;
  }

  return n + sumFromOneToN(n - 1);
}
```

Tail recursion occurs when the recursive call is in the tail position within its enclosing context - after the function calls itself, it performs no additional work. That is, once the base case is complete, the solution is apparent. For example:

```java
public int sumFromOneToN(int n, int a) {
  if (n < 1) {
    return a;
  }

  return sumFromOneToN(n - 1, a + n);
}
```

Here you can see that `a` plays the role of the accumulator - instead of computing the sum on the way down the stack, we compute it on the way up, effectively making the return trip unnecessary, since it stores no additional state and performs no further computation. Once we hit the base case, the work is done - below is that same function, “unrolled”.

```java
public int sumFromOneToN(int n) {
  int a = 0;

  while(n > 0) {
    a += n--;
  }

  return a;
}
```

Many functional languages natively support tail call optimization, however the JVM does not. In order to implement recursive functions in Java, we need to be aware of this limitation to avoid `StackOverflowError`s. In Java, iteration is almost universally preferred to recursion.

How can you catch an exception thrown by another thread in Java?

```java
// create our uncaught exception handler
Thread.UncaughtExceptionHandler handler = new Thread.UncaughtExceptionHandler() {
    public void uncaughtException(Thread th, Throwable ex) {
        System.out.println("Uncaught exception: " + ex);
    }
};

// create another thread
Thread otherThread = new Thread() {
    public void run() {
        System.out.println("Sleeping ...");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted.");
        }
        System.out.println("Throwing exception ...");
        throw new RuntimeException();
    }
};

// set our uncaught exception handler as the one to be used when the new thread
// throws an uncaught exception
otherThread.setUncaughtExceptionHandler(handler);

// start the other thread - our uncaught exception handler will be invoked when
// the other thread throws an uncaught exception
otherThread.start();
```

What is the Java Classloader? List and explain the purpose of the three types of class loaders.

The Java Classloader is the part of the Java runtime environment that loads classes on demand (lazy loading) into the JVM (Java Virtual Machine). Classes may be loaded from the local file system, a remote file system, or even the web.

When the JVM is started, three class loaders are used:

1. **Bootstrap Classloader:** Loads core java API file rt.jar from folder.
2. **Extension Classloader:** Loads jar files from folder.
3. **System/Application Classloader:** Loads jar files from path specified in the CLASSPATH environment variable.

When designing an abstract class, why should you avoid calling abstract methods inside its constructor?

This is a problem of initialization order. The subclass constructor will not have had a chance to run yet and there is no way to force it to run it before the parent class. Consider the following example class:

```
public abstract class Widget {

    private final int cachedWidth;
    private final int cachedHeight;

    public Widget() {
        this.cachedWidth = width();
        this.cachedHeight = height();
    }

    protected abstract int width();
    protected abstract int height();
}
```

This seems like a good start for an abstract Widget: it allows subclasses to fill in width and height, and caches their initial values. However, look when you spec out a typical subclass implementation like so:

```
public class SquareWidget extends Widget {

    private final int size;

    public SquareWidget(int size) {
        this.size = size;
    }

    @Override
    protected int width() {
        return size;
    }

    @Override
    protected int height() {
        return size;
    }
}
```

Now, we've introduced a subtle bug: Widget.cachedWidth and Widget.cachedHeight will always be zero for SquareWidget instances! This is because the this.size = size assignment occurs after the Widget constructor runs. Avoid calling abstract methods in your abstract classes' constructors, as it restricts how those abstract methods can be implemented.

What variance is imposed on generic type parameters? How much control does Java give you over this?

Java's generic type parameters are *invariant*. This means for any distinct types `A` and `B`, `G<A>` is not a subtype or supertype of `G<B>`. As a real world example, `List<String>` is not a supertype or subtype of `List<Object>`. So even though `String` extends (i.e. is a subtype of) `Object`, both of the following assignments will fail to compile:

```
List<String> strings = Arrays.<Object>asList("hi there");
List<Object> objects = Arrays.<String>asList("hi there");
```

Java does give you some control over this in the form of *use-site variance*. On individual methods, we can use `? extends Type` to create a *covariant* parameter. Here's an example:

```
public double sum(List<? extends Number> numbers) {
    double sum = 0;
    for (Number number : numbers) {
        sum += number.doubleValue();
    }
    return sum;
}


List<Long> longs = Arrays.asList(42L, 128L, -10L);
double sumOfLongs = sum(longs);
```

Even though `longs` are a `List<Long>` and not `List<Number>`, it can be passed to `sum`. Similarly, `? Super Type` lets a method parameter be *contravariant*. Consider a function with a callback parameter:

```
public void forEachNumber(Callback<? super Number> callback) {
    callback.call(50.0f);
    callback.call(123123);
    callback.call((short) 99);
}
```

`forEachNumber` allows `Callback<Object>` to be a subtype of `Callback <Number>`, which means any callback that handles a supertype of `Number` will do:

```
forEachNumber(new Callback<Object>() {
    @Override public void call(Object value) {
        System.out.println(value);
    }
});
```

Note, however, that attempting to provide a callback that handles only `Long` (a subtype of `Number`) will rightly fail:

```
forEachNumber(new Callback<Object>() {
    @Override public void call(Object value) {
        System.out.println(value);
    }
});
```

Liberal application of use-site variance can prevent many of the unsafe casts that often appear in Java code and is crucial when designing interfaces used by multiple developers.

If one needs a Set, how do you choose between HashSet vs. TreeSet?

At first glance, `HashSet` is superior in almost every way: O(1) `add`, `remove` and `contains`, vs. O(log(N)) for `TreeSet`. However, `TreeSet` is indispensable when you wish to maintain order over the inserted elements or query for a range of elements within the set.
Consider a `Set` of timestamped `Event` objects. They could be stored in a `HashSet`, with `equals` and `hashCode` based on that timestamp. This is efficient storage and permits looking up events by a specific timestamp, but how would you

get all events that happened on any given day? That would require a O(n) traversal of the `HashSet`, but it's only a O(log(n)) operation with `TreeSet` using the `tailSet` method:

```
public class Event implements Comparable<Event> {
    private final long timestamp;

    public Event(long timestamp) {
        this.timestamp = timestamp;
    }

    @Override public int compareTo(Event that) {
        return Long.compare(this.timestamp, that.timestamp);
    }
}

...

SortedSet<Event> events = new TreeSet<>();
events.addAll(...); // events come in

// all events that happened today
long midnightToday = ...;
events.tailSet(new Event(midnightToday));
```

If `Event` happens to be a class that we cannot extend or that doesn't implement `Comparable`, `TreeSet` allows us to pass in our own `Comparator`:

```
SortedSet<Event> events = new TreeSet<>(
        (left, right) -> Long.compare(left.timestamp, right.timestamp));
```

Generally speaking, `TreeSet` is a good choice when order matters and when reads are balanced against the increased cost of writes.

What are method references, and how are they useful?

Method references were introduced in Java 8 and allow constructors and methods (static or otherwise) to be used as lambdas. They allow one to discard the boilerplate of a lambda when the method reference matches an expected signature.

For example, suppose we have a service that must be stopped by a shutdown hook. Before Java 8, we would have code like this:

```
final SomeBusyService service = new SomeBusyService();
service.start();

onShutdown(new Runnable() {
    @Override
    public void run() {
        service.stop();
    }
});
```

With lambdas, this can be cut down considerably:

```
onShutdown(() -> service.stop());
```

However, `stop` matches the signature of `Runnable.run` (`void` return type, no parameters), and so we can introduce a method reference to the `stop` method of that specific `SomeBusyService` instance:

```
onShutdown(service::stop);
```

This is terse (as opposed to verbose code) and clearly communicates what is going on.

Method references don't need to be tied to a specific instance, either; one can also use a method reference to an arbitrary object, which is useful in `Stream` operations. For example, suppose we have a `Person` class and want just the lowercase names of a collection of people:

```
List<Person> people = ...

List<String> names = people.stream()
        .map(Person::getName)
        .map(String::toLowerCase)
        .collect(toList());
```

A complex lambda can also be pushed into a static or instance method and then used via a method reference instead. This makes the code more reusable and testable than if it were "trapped" in the lambda.

So we can see that method references are mainly used to improve code organization, clarity and terseness.

How are Java enums more powerful than integer constants? How can this capability be used?

Enums are essentially final classes with a fixed number of instances. They can implement interfaces but cannot extend another class.

This flexibility is useful in implementing the strategy pattern, for example, when the number of strategies is fixed. Consider an address book that records multiple methods of contact. We can represent these methods as an enum and attach fields, like the filename of the icon to display in the UI, and any corresponding behaviour, like how to initiate contact via that method:

```
public enum ContactMethod {
    PHONE("telephone.png") {
        @Override public void initiate(User user) {
            Telephone.dial(user.getPhoneNumber());
        }
    },
    EMAIL("envelope.png") {
        @Override public void initiate(User user) {
            EmailClient.sendTo(user.getEmailAddress());
        }
    },
    SKYPE("skype.png") {
        ...
    };

    ContactMethod(String icon) {
        this.icon = icon;
    }

    private final String icon;

    public abstract void initiate(User user);

    public String getIcon() {
        return icon;
    }
}
```

We can dispense with `switch` statements entirely by simply using instances of `ContactMethod`:

```
ContactMethod method = user.getPrimaryContactMethod();
displayIcon(method.getIcon());
method.initiate(user);
```

This is just the beginning of what can be done with enums. Generally, the safety and flexibility of enums means they should be used in place of integer constants, and switch statements can be eliminated with liberal use of abstract methods.

What does it mean for a collection to be "backed by" another? Give an example of when this property is useful.

If a collection backs another, it means that changes in one are reflected in the other and vice-versa. For example, suppose we wanted to create a `whitelist` function that removes invalid keys from a `Map`. This is made far easier with `Map.keySet`, which returns a set of keys that is backed by the original map. When we remove keys from the key set, they are also removed from the backing map:

```
public static <K, V> Map<K, V> whitelist(Map<K, V> map, K... allowedKeys) {
    Map<K, V> copy = new HashMap<>(map);
    copy.keySet().retainAll(asList(allowedKeys));
    return copy;
}
```

`retainAll` writes through to the backing map, and allows us to easily implement something that would otherwise require iterating over the entries in the input map, comparing them against `allowedKey`, etcetera.

Note, it is important to consult the documentation of the backing collection to see which modifications will successfully write through. In the example above, `map.keySet().add(value)` would fail, because we cannot add a key to the backing map without a value.

What is reflection? Give an example of functionality that can only be implemented using reflection.

Reflection allows programmatic access to information about a Java program's types. Commonly used information includes: methods and fields available on a class, interfaces implemented by a class, and the runtime-retained annotations on classes, fields and methods.

Examples given are likely to include:

i. Annotation-based serialization libraries often map class fields to JSON keys or XML elements (using annotations). These libraries need reflection to inspect those fields and their annotations and also to access the values during serialization.
ii. Model-View-Controller frameworks call controller methods based on routing rules. These frameworks must use reflection to find a method corresponding to an action name, check that its signature conforms to what the framework expects (e.g. takes a `Request` object, returns a `Response`), and finally, invoke the method.
iii. Dependency injection frameworks lean heavily on reflection. They use it to instantiate arbitrary beans for injection, check fields for annotations such as `@Inject` to discover if they require injection of a bean, and also to set those values.
iv. Object-relational mappers such as Hibernate use reflection to map database columns to fields or getter/setter pairs of a class, and can go as far as to infer table and column names by reading class and getter names, respectively.

A concrete code example could be something simple, like copying an object's fields into a map:

```
Person person = new Person("Doug", "Sparling", 31);

Map<String, Object> values = new HashMap<>();
for (Field field : person.getClass().getDeclaredFields()) {
    values.put(field.getName(), field.get(person));
}

// prints {firstName=Doug, lastName=Sparling, age=31}
System.out.println(values);
```

Such tricks can be useful for debugging, or for utility methods such as a `toString` method that works on any class.
Aside from implementing generic libraries, direct use of reflection is rare but it is still a handy tool to have. Knowledge of reflection is also useful for when these mechanisms fail.

However, it is often prudent to avoid reflection unless it is strictly necessary, as it can turn straightforward compiler errors into runtime errors.

What are static initializers and when would you use them?

A static initializer gives you the opportunity to run code during the initial loading of a class and it guarantees that this code will only run once and will finish running before your class can be accessed in any way.
They are useful for performing initialization of complex static objects or to register a type with a static registry, as JDBC drivers do.
Suppose you want to create a static, immutable Map containing some feature flags. Java doesn't have a good one-liner for initializing maps, so you can use static initializers instead:

```
public static final Map<String, Boolean> FEATURE_FLAGS;
static {
    Map<String, Boolean> flags = new HashMap<>();
    flags.put("frustrate-users", false);
    flags.put("reticulate-splines", true);
    flags.put(...);
    FEATURE_FLAGS = Collections.unmodifiableMap(flags);
}
```

Within the same class, you can repeat this pattern of declaring a static field and immediately initializing it, since multiple static initializers are allowed.

Nested classes can be static or non-static (also called an inner class). How do you decide which to use? Does it matter?

The key difference between is that inner classes have full access to the fields and methods of the enclosing class. This can be convenient for event handlers, but comes at a cost: every instance of an inner class retains and requires a reference to its enclosing class.

With this cost in mind, there are many situations where we should prefer static nested classes. When instances of the nested class will outlive instances of the enclosing class, the nested class should be static to prevent memory leaks. Consider this implementation of the factory pattern:

```
public interface WidgetParser {
    Widget parse(String str);
}

public class WidgetParserFactory {
    public WidgetParserFactory(ParseConfig config) {
        ...
    }

    public WidgetParser create() {
        new WidgetParserImpl(...);
    }

    private class WidgetParserImpl implements WidgetParser {
        ...

        @Override public Widget parse(String str) {
            ...
        }
    }
}
```

At a glance, this design looks good: the WidgetParserFactory hides the implementation details of the parser with the nested class WidgetParserImpl. However, WidgetParserImpl is not static, and so if WidgetParserFactory is discarded immediately after the WidgetParser is created, the factory will leak, along with all the references it holds.

WidgetParserImpl should be made static, and if it needs access to any of WidgetParserFactory's internals, they should be passed into WidgetParserImpl's constructor instead. This also makes it easier to extract WidgetParserImpl into a separate class should it outgrow its enclosing class.

Inner classes are also harder to construct via reflection due to their “hidden” reference to the enclosing class, and this reference can get sucked in during reflection-based serialization, which is probably not intended.

So we can see that the decision of whether to make a nested class static is important, and that one should aim to make nested classes static in cases where instances will “escape” the enclosing class or if reflection on those nested classes is involved.

A third party library is throwing NoClassDefFoundError or NoSuchMethodError, even though all your code compiles without error. What is happening?

A modern Java project can have dozens, or even hundreds, of dependencies in the form of JAR files that are all loaded onto the same classpath. Juggling these dependencies requires the use of build tools such as Maven, Gradle, SBT or Ivy.

Among other things, these tools help prevent multiple versions of the same library from being added to the project, which leads to non-deterministic behaviour based on which JARs are loaded first. When your project includes two libraries that rely on different versions of the same dependency, often the newest version of that dependency is selected. However, since Java does not employ static linking (as C++ or C do), we get NoClassDefFoundError or NoSuchMethodError at runtime when one library was not compatible with such an upgrade. Fixing this issue can involve some detective work. Websites such as findJAR can help track down the name of the

dependency/JAR whose API changed. Java build tools and IDEs can also produce dependency reports that tell you which libraries depend on that JAR; usually the issue is resolvable by identifying and upgrading the library that depends on the older JAR.

What is the difference between String s = "Test" and String s = new String("Test")? Which is better and why?

In general, String s = "Test" is more efficient to use than String s = new **String("Test")**

In the case of String s = "Test", a String with the value “Test” will be created in the String pool. If another String with the same value is then created (e.g., String s2 = "Test"), it will reference this same object in the String pool.

However, if you use String s = new String("Test"), in addition to creating a String with the value “Test” in the String pool, that String object will then be passed to the constructor of the String Object (i.e., new String("Test")) and will create another String object (not in the String pool) with that value. Each such call will therefore create an additional String object (e.g., String s2 = new String("Test") would create an addition String object, rather than just reusing the same String object from the String pool).

What’s the String Pool?

Java String Pool – **the special memory region where *Strings* are stored by the JVM.**

String Interning

Thanks to the immutability of *Strings* in Java, the JVM can optimize the amount of memory allocated for them by **storing only one copy of each literal *String* in the pool**. This process is called *interning*.

When we create a *String* variable and assign a value to it, the JVM searches the pool for a *String* of equal value.

If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory.

If not found, it'll be added to the pool (interned) and its reference will be returned.

Let's write a small test to verify this:

```
1  String constantString1 = "Baeldung";
2  String constantString2 = "Baeldung";
3
4  assertThat(constantString1)
5    .isSameAs(constantString2);
```

3. *Strings* Allocated using the Constructor

When we create a *String* via the *new* operator, the Java compiler will create a new object and store it in the heap space reserved for the JVM.

Every *String* created like this will point to a different memory region with its own address.

Let’s see how this is different from the previous case:

```
String constantString = "Baeldung";
String newString = new String("Baeldung");

assertThat(constantString).isNotSameAs(newString);
```

4. *String* Literal vs *String Object*

When we create a *String* object using the *new()* operator, it always creates a new object in heap memory. On the other hand, if we create an object using *String* literal syntax e.g. “Baeldung”, it may return an existing object from the String pool, if it already exists. Otherwise, it will create a new String object and put in the string pool for future re-use.

At a high level, both are the *String* objects, but the main difference comes from the point that *new()* operator always creates a new *String* object. Also, when we create a *String* using literal – it is interned.

This will be much more clear when we compare two *String* objects created using *String* literal and the *new*operator:

```
String first = "Baeldung";
String second = "Baeldung";
System.out.println(first == second); // True
```

In this example, the *String* objects will have the same reference.

Next, let’s create two different objects using *new* and check that they have different references:

```
String third = new String("Baeldung");
String fourth = new String("Baeldung");
System.out.println(third == fourth); // False
```

Similarly, when we compare a *String* literal with a *String* object created using *new()* operator using the == operator, it will return *false:*

```
1  String fifth = "Baeldung";
2  String sixth = new String("Baeldung");
3  System.out.println(fifth == sixth); // False
```

In general, **we should use the *String* literal notation when possible**. It is easier to read and it gives the compiler a chance to optimize our code.

5. Manual Interning

We can manually intern a *String* in the Java String Pool by calling the *intern()* method on the object we want to intern.

Manually interning the *String* will store its reference in the pool, and the JVM will return this reference when needed.

Let’s create a test case for this:

```
1  String constantString = "interned Baeldung";
2  String newString = new String("interned Baeldung");
3
4  assertThat(constantString).isNotSameAs(newString);
5
6  String internedString = newString.intern();
7
8  assertThat(constantString)
9    .isSameAs(internedString);
```

6. Garbage Collection

Before Java 7, the JVM **placed the Java String Pool in the *PermGen* space, which has a fixed size — it can’t be expanded at runtime and is not eligible for garbage collection**.

The risk of interning *Strings* in the *PermGen* (instead of the *Heap*) is that **we can get an *OutOfMemory* error**from the JVM if we intern too many *Strings*.

From Java 7 onwards, the Java String Pool is **stored in the *Heap* space, which is garbage collected** by the JVM. The advantage of this approach is the **reduced risk of *OutOfMemory* error** because unreferenced *Strings* will be removed from the pool, thereby releasing memory.

7. Performance and Optimizations

In Java 6, the only optimization we can perform is increasing the *PermGen* space during the program invocation with the *MaxPermSize* JVM option:

```
1  -XX:MaxPermSize=1G
```

In Java 7, we have more detailed options to examine and expand/reduce the pool size. Let's see the two options for viewing the pool size:

```
1  -XX:+PrintFlagsFinal
1  -XX:+PrintStringTableStatistics
```

The default pool size is 1009. If we want to increase the pool size, we can use the *StringTableSize* JVM option:

```
1  -XX:StringTableSize=4901
```

Note that increasing the pool size will consume more memory but has the advantage of reducing the time required to insert the *Strings* into the table.

8. A Note About Java 9

Until Java 8, *Strings* were internally represented as an array of characters – *char[]*, encoded in *UTF-16*, so that every character uses two bytes of memory.

With Java 9 a new representation is provided, called *Compact Strings*. This new format will choose the appropriate

encoding between *char[]* and *byte[]* depending on the stored content.

Since the new *String* representation will use the *UTF-16* encoding only when necessary, the amount of *heap*memory will be significantly lower, which in turn causes less *Garbage Collector* overhead on the *JVM*.

9. Conclusion

In this guide, we showed how the JVM and the Java compiler optimize memory allocations for *String*objects via the Java String Pool.

Which two method you need to implement for key Object in HashMap?

In order to use any object as Key in HashMap, it must implements equals and hashcode method in Java.

What is immutable object? Can you write immutable object?

Immutable classes are Java classes whose objects cannot be modified once created. Any modification in Immutable object results in new object. For example is String is immutable in Java. Mostly Immutable are also final in Java, in order to prevent sub class from overriding methods in Java, which can compromise Immutability. You can achieve same functionality by making member as non-final but private and not modifying them except in constructor.

What is the difference between creating String as new() and literal?

When we create string with new() Operator, it's created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.

String s = new String("Test"); does not put the object in String pool , we need to call String.intern() method which is used to put them into String pool explicitly. It's only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool.

Before Java 7, the JVM placed the Java String Pool in the *PermGen* space, which has a fixed size — it can't be expanded at runtime and is not eligible for garbage collection.

The risk of interning *Strings* in the *PermGen* (instead of the *Heap*) is that we can get an *OutOfMemory* errorfrom the JVM if we intern too many *Strings*.

From Java 7 onwards, the Java String Pool is stored in the *Heap* space, which is garbage collected by the JVM. The advantage of this approach is the reduced risk of *OutOfMemory* error because unreferenced *Strings* will be removed from the pool, thereby releasing memory.

What is difference between StringBuffer and StringBuilder in Java ?

StringBuffer methods are synchronized while StringBuilder is non-synchronized.

What is the difference between ArrayList and Vector?

Synchronization and Thread-Safe: Vector is synchronized while ArrayList is not synchronized. Synchronization and thread safe means at a time only one thread can access the code .In Vector class all the methods are synchronized. That's why the Vector object is already synchronized when it is created.

Performance: Vector is slow as it is threading safe. In comparison ArrayList is fast as it is non-synchronized. Thus in ArrayList two or more threads can access the code at the same time, while Vector is limited to one thread at a time.

Automatic Increase in Capacity": A Vector defaults to doubling size of its array. While when you insert an element into the ArrayList, it increases its Array size by 50%. By default ArrayList size is 10. It checks whether it reaches the last element then it will create the new array, copy the new data of last array to new array, then old array is garbage collected by the Java Virtual Machine (JVM).

Set Increment Size: ArrayList does not define the increment size. Vector defines the increment size.
You can find the following method in Vector Class
public synchronized void setSize(int i) { //some code }
There is no setSize() method or any other method in ArrayList which can manually set the increment size.

Enumerator: Other than Hashtable ,Vector is the only other class which uses both Enumeration and Iterator .While ArrayList can only use Iterator for traversing an ArrayList.

What is the difference between iterator and enumeration?

Iterator is the interface and found in the java.util package. It has three methods hasNext(), next() and remove(). Enumeration is also an interface and found in the java.util package. An enumeration is an object that generates elements one at a time. It is used for passing through a collection, usually of unknown size. The traversing of elements can only be done once per creation. It has following methods hasMoreElements() and nextElement() An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework

How do you handle error condition while writing stored procedure or accessing stored procedure from java?

My take is that stored procedure should return error code if some operation fails but if stored procedure itself fail than catching SQLException is only choice.

What is difference between Executor.submit() and Executer.execute() method?

There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with execute this exception will go to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with submit any thrown exception, checked exception or not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will re-throw this exception, wrapped in an ExecutionException.

What is the difference between factory and abstract factory pattern?

Abstract Factory provides one more level of abstraction. Consider different factories each extended from an Abstract Factory and responsible for creation of different hierarchies of objects based on the type of factory. E.g. AbstractFactory extended by AutomobileFactory, UserFactory, RoleFactory etc. Each individual factory would be responsible for creation of objects in that genre.

What is Singleton? Is it better to make whole method synchronized or only critical section synchronized?

Singleton in Java is a class with just one instance in whole Java application, for example java.lang.Runtime is a Singleton class. Creating Singleton was tricky prior Java 4 but once Java 5 introduced Enum it’s very easy.

Sometimes it’s important for some classes to have exactly one instance. There are many objects we only need one instance of them and if we, instantiate more than one, we’ll run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

You may require only one object of a class, for example, when you are a creating the context of an application, or a thread manageable pool, registry settings, a driver to connect to the input or output console etc. More than one object of that type clearly will cause inconsistency to your program.

The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to it. However, although the Singleton is the simplest in terms of its class diagram because there is only one single class, its implementation is a bit trickier.

In this lesson, we will try different ways to create only a single object of the class and will also see how one way is better than the other.

2. How to create a class using the Singleton Pattern
There could be many ways to create such type of class, but still, we will see how one way is better than the other.

Let’s start with a simple way.

What if, we provide a global variable that makes an object accessible? For example:

```
public class SingletonEager {

    public static SingletonEager sc = new SingletonEager();

}
```

As we know, there is only one copy of the static variables of a class, we can apply this. As far as, the client code is using this sc static variable its fine. But, if the client

uses a new operator there would be a new instance of this class.

To stop the class to get instantiated outside the class, let’s make the constructor of the class as private.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    public static SingletonEager sc = new SingletonEager();
    private SingletonEager(){}

}
```

Is this going to work? I think yes. By keeping the constructor private, no other class can instantiate this class. The only way to get the object of this class is using the sc static variable which ensures only one object is there.

But as we know, providing a direct access to a class member is not a good idea. We will provide a method through which the sc variable will get access, not directly.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonEager {
    private static SingletonEager sc = new SingletonEager();
    private SingletonEager(){}
    public static SingletonEager getInstance(){
```

```
            return sc;
        }
    }
```

So, this is our singleton class which makes sure that only one object of the class gets created and even if there are several requests, only the same instantiated object will be returned.

The one problem with this approach is that the object would get created as soon as the class gets loaded into the JVM. If the object is never requested, there would be an object useless inside the memory.

It’s always a good approach that an object should get created when it is required. So, we will create an object on the first call and then will return the same object on other successive calls.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazy {

    private static SingletonLazy sc = null;
    private SingletonLazy(){}
    public static SingletonLazy getInstance(){
        if(sc==null){
```

```
                sc = new SingletonLazy();
            }
            return sc;
        }
    }
```

In the getInstance() method, we check if the static variable sc is null, then instantiate the object and return it. So, on the first call when sc would be null the object gets created and on the next successive calls it will return the same object.

This surely looks good, doesn’t it? But this code will fail in a multi-threaded environment. Imagine two threads concurrently accessing the class, thread t1 gives the first call to the getInstance() method, it checks if the static variable sc is null and then gets interrupted due to some reason. Another thread t2 calls the getInstance() method successfully passes the if check and instantiates the object. Then, thread t1 gets awake and it also creates the object. At this time, there would be two objects of this class.

To avoid this, we will use the synchronized keyword to the getInstance() method. With this way, we force every thread to wait its turn before it can enter the method. So, no two threads will enter the method at the same time. The synchronized comes with a price, it will decrease the performance, but if the call to the getInstance() method isn’t causing a substantial overhead for your application, forget about it. The other workaround is to move to eager instantiation approach as shown in the previous example.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyMultithreaded {

    private static SingletonLazyMultithreaded sc = null;
    private SingletonLazyMultithreaded(){}
    public              static              synchronized
SingletonLazyMultithreaded getInstance(){
        if(sc==null){
            sc = new SingletonLazyMultithreaded();
        }
        return sc;
    }
}
```

But if you want to use synchronization, there is another technique known as “double-checked locking” to reduce the use of synchronization. With the double-checked locking, we first check to see if an instance is created, and if not, then we synchronize. This way, we only synchronize the first time.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletonLazyDoubleCheck {

    private volatile static SingletonLazyDoubleCheck sc
= null;
    private SingletonLazyDoubleCheck(){}
    public        static        SingletonLazyDoubleCheck
getInstance(){
        if(sc==null){
```

```
    synchronized(SingletonLazyDoubleCheck.class){
                    if(sc==null){
                        sc = new SingletonLazyDoubleCheck();
                    }
                }
            }
            return sc;
        }
    }
```

Apart from this, there are some other ways to break the singleton pattern.

If the class is Serializable.
If it’s Clonable.
It can be break by Reflection.
And also if, the class is loaded by multiple class loaders.

The following example shows how you can protect your class from getting instantiated more than once.

```
public class Singleton implements Serializable{

      private static final long serialVersionUID = -
1093810940935189395L;
      private static Singleton sc = new Singleton();

     private Singleton(){
             if(sc!=null){
                 throw                                  new
     IllegalStateException("Already created.");
             }
     }

     public static Singleton getInstance(){
          return sc;
     }

    public        static        SingletonLazyDoubleCheck
getInstance(){

        if(sc==null){


 synchronized(SingletonLazyDoubleCheck.class){

                 if(sc==null){
                     sc              =              new
 SingletonLazyDoubleCheck();

                }

            }
```

```
        }

        return sc;

    }



    private      Object      readResolve()      throws
ObjectStreamException{
        return sc;
    }

    private      Object      writeReplace()      throws
ObjectStreamException{
        return sc;
    }

    public         Object         clone()         throws
CloneNotSupportedException{
        throw new CloneNotSupportedException("Singleton,
cannot be clonned");
    }

    private  static  Class  getClass(String  classname)
throws ClassNotFoundException {
        ClassLoader             classLoader            =
Thread.currentThread().getContextClassLoader();
        if(classLoader == null)
            classLoader                                =
Singleton.class.getClassLoader();
        return (classLoader.loadClass(classname));
    }

}
```

Implement the readResolve() and writeReplace() methods in your singleton class and return the same object through them.
You should also implement the clone() method and throw an exception so that the singleton cannot be cloned.
An “if condition” inside the constructor can prevent the singleton from getting instantiated more than once using reflection.
To prevent the singleton getting instantiated from different class loaders, you can implement the getClass() method. The above getClass() method associates the classloader with the current thread; if that classloader is null, the method uses the same classloader that loaded the singleton class.

Although we can use all these techniques, there is one simple and easier way of creating a singleton class. As of JDK 1.5, you can create a singleton class using enums. The Enum constants are implicitly static and final and you cannot change their values once created.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletoneEnum {

    public enum SingleEnum{
        SINGLETON_ENUM;
    }
}


public enum SingletonEnum {

    INSTANCE;
    int value;
```

```
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
}


public class EnumDemo {

    public static void main(String[] args) {
        SingletonEnum singleton =
SingletonEnum.INSTANCE;
        System.out.println(singleton.getValue());
        singleton.setValue(2);
        System.out.println(singleton.getValue());
    }
}
```

You will get a compile time error when you attempt to explicitly instantiate an Enum object. As Enum gets loaded statically, it is thread-safe. The clone method in Enum is final which ensures that enum constants never get cloned. Enum is inherently serializable, the serialization mechanism ensures that duplicate instances are never created as a result of deserialization. Instantiation using reflection is also prohibited. These things ensure that no instance of an enum exists beyond the one defined by the enum constants.

3. When to use Singleton

There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

When do you override hashcode and equals() ?

Whenever necessary especially if you want to do equality check or want to use your object as key in HashMap.

What will be the problem if you don't override hashcode() method ?

You will not be able to recover your object from hash Map if that is used as key in HashMap.

15. Is it better to synchronize critical section of getInstance() method or whole getInstance() method ?

It would be better to synchronized the critical section because if we lock whole method than every time some one call this method will have to wait even though we are not creating any object)

What is the difference when String is gets created using literal or new() operator ?

When we create string with new() its created in heap and not added into string pool while String created using literal are created in String pool itself which exists in Perm area of heap.

Doesn’t overriding hashcode() method has any performance implication?

A poor hashcode function will result in frequent collision in HashMap, which eventually increase time for adding an object into Hash Map.

What's wrong using HashMap in multithreaded environment? When get() method go to infinite loop ?

It's during concurrent access and re-sizing.

What do you understand by thread-safety? Why is it required? Finally, how to achieve thread-safety in Java Applications?

Java Memory Model defines the legal interaction of threads with the memory in a real computer system. In a way, it describes what behaviors are legal in multi-threaded code. It determines when a Thread can reliably see writes to variables made by other threads. It defines semantics for volatile, final & synchronized that makes guarantee of visibility of memory operations across the Threads.

Memory Barrier are the base for our further discussions. There are two types of memory barrier instructions in JMM - read barriers and write barrier.

A read barrier invalidates the local memory (cache, registers, etc) and then reads the contents from the main memory, so that changes made by other threads becomes visible to the current Thread.
A write barrier flushes out the contents of the processor's local memory to the main memory, so that changes made by the current Thread becomes visible to the other threads.

What's the JMM semantics for synchronized?

When a thread acquires monitor of an object, by entering into a synchronized block of code, it performs a read barrier (invalidates the local memory and reads from the heap instead). Similarly exiting from a synchronized block as part of releasing the associated monitor, it performs a write barrier (flushes changes to the main memory) Thus modifications to a shared state using synchronized block by one Thread, is guaranteed to be visible to subsequent synchronized reads by other threads. This guarantee is provided by JMM in presence of synchronized code block.

What’s the JMM semantics for Volatile fields?

Read & write to volatile variables have it memory semantics as that of acquiring and releasing a monitor using synchronized code block. So the visibility of volatile field is guaranteed by the JMM. Moreover afterwards Java 1.5, volatile reads and writes are not reorderable with any other memory operations (volatile and non-volatile both). Thus when Thread A writes to a volatile variable V, and afterwards Thread B reads from variable V, any variable values that were visible to A at the time V was written are guaranteed now to be visible to B.

Let's try to understand the same using the following code,

```
Data data = null;
volatile boolean flag = false;

Thread A
========

data = new Data();
```

```
flag = true;  <-- writing to volatile will flush data as well as flag to main memory


Thread B
========

if(flag==true){  <--  as=""  barrier=""  data.=""  flag=""  font=""  for=""  from=""  perform=""  read=""  reading=""  volatile="" well="" will="">
use data;  <!--- data is guaranteed to visible even though it is not declared volatile because of the JMM semantics of volatile flag.
}
```

What will happen if you call return statement or System.exit on try or catch block? Will finally block execute?

This is a very *popular tricky Java question* and its tricky because many programmers think that finally block always executed. This question challenges that concept by putting return statement in try or catch block or calling System.exit from try or catch block. Finally block will execute even if you put return statement in try block or catch block but finally block won't run if you call System.exit form try or catch.

Can you override private or static method in Java?

Overriding is a good topic to ask trick questions in Java. Anyway, you can’t override private or static method in Java, if you create similar method with same return type and same method arguments that's called method hiding.

What will happen if we put a key object in a HashMap which is already there?

If you put the same key again than it will replace the old mapping/ value because HashMap doesn't allow duplicate keys.

If a method throws NullPointerException in super class, can we override it with a method which throws RuntimeException?

Its possible to throw super class of RuntimeException in overridden method but you cannot do it if it's checked Exception.

What is the issue with following implementation of compareTo() method in Java?

```
public int compareTo(Object o){

   Employee emp = (Employee) emp;
   return this.id - o.id;
}
```

How do you ensure that N thread can access N resources without deadlock?

Key point here is order, if you acquire resources in a particular order and release resources in reverse order you can prevent deadlock.

What is difference between CyclicBarrier and CountDownLatch in Java

Main difference between both of them is that you can reuse CyclicBarrier even if Barrier is broken but you cannot reuse CountDownLatch in Java.

Can you access non-static variable in static context?

No you can't access static variable in non-static context in Java.

Explain the insertion and extraction runtime complexity of Hash table and Binary tree.

For binary search tree, in the average case, access, search, insertion and deletion time complexity is O(log(n)) and in the worst case scenario, in all the mentioned cases, time complexity is O(n). The space complexity in the worst case scenario is O(n). For hash table, in the average case, search, insertion and deletion time complexity is O(1) and in the worst case scenario, in all the mentioned cases, time complexity is O(n). The space complexity in the worst case scenario is O(n).

What’s the adapter design pattern?

Adapter design pattern convert the interface of a class into another interface clients expect and lets classes work together that couldn't otherwise because of incompatible interfaces. It wraps an existing class with a new interface and provide impedance match an old component to a new system.

What’s the facade design pattern?

The Facade Pattern makes a complex interface easier to use, using a Facade class. The Facade Pattern provides a unified interface to a set of interface in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

What’s the composite design pattern?

The Composite Pattern allows you to compose objects into a tree structure to represent the part-whole hierarchy which means you can create a tree of objects that is made of different parts, but that can be treated as a whole one big thing. Composite lets clients to treat individual objects and compositions of objects uniformly that are the intent of the Composite Pattern.

What’s the bridge design pattern?

The Bridge Pattern’s intent is to decouple an abstraction from its implementation so that the two can vary independently. It puts the abstraction and implementation into two different class hierarchies so that both can be extend independently.

What’s the singleton design pattern?

The Singleton pattern is used when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point or when the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

What’s the observer design pattern?

The Observer Pattern is a kind of behavior pattern, which is concerned, with the assignment of responsibilities between objects. It should be used when an abstraction has two aspects, one dependent on the other, when a change to one object requires changing others, and you don’t know how many objects need to be changed or when an object should be able to notify other objects without making assumptions about who

these objects are. In other words, you don’t want these objects tightly coupled.

What’s the mediator design pattern?

The Mediator Pattern defines an object that encapsulates how a set of objects interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. Rather than interacting directly with each other, objects ask the Mediator to interact on their behalf, which results in reusability and loose coupling. You will learn how and when the Mediator design pattern should be used and how to structure your code in order to implement it.

What’s the proxy design pattern?

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. It comes up with many different variations. Some of the important variations are, Remote Proxy, Virtual Proxy, and Protection Proxy. In this lesson, we will know more about these variations and we will implement each of them in Java. But before we do that, let’s get to know more about the Proxy Pattern in general. You will learn how and when the Proxy design pattern should be used and how to structure your code in order to implement it.

What’s the chain of responsibility design pattern?

The Chain of Responsibility pattern is a behavior pattern in which a group of objects is chained together in a sequence and a responsibility (a request) is provided in-order to be handled by the group. If an object in the group can process

the particular request, it does so and returns the corresponding response. Otherwise, it forwards the request to the subsequent object in the group.

What’s the flyweight design pattern?

The Flyweight Pattern is designed to control object creation where objects in an application have great similarities and are of a similar kind, and provides you with a basic caching mechanism. It allows you to create one object per type (the type here differs by a property of that object), and if you ask for an object with the same property (already created), it will return you the same object instead of creating a new one.

What’s the builder design pattern?

The intent of the Builder Pattern is to separate the construction of a complex object from its representation, so that the same construction process can create different representations. This type of separation reduces the object size. The design turns out to be more modular with each implementation contained in a different builder object. Adding a new implementation (i.e., adding a new builder) becomes easier.

What’s the factory design pattern?

The Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method. The Factory Method selects an appropriate class from a class hierarchy based on the

application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type.

What’s the abstract factory design pattern?

The Abstract Factory (A.K.A. Kit) is a design pattern, which provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Abstract Factory pattern takes the concept of the Factory Method Pattern to the next level. An abstract factory is a class that provides an interface to produce a family of objects.

What’s the prototype design pattern?

The Prototype design pattern is used to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. The concept is to copy an existing object rather than creating a new instance from scratches, something that may include costly operations. The existing object acts as a prototype and contains the state of the object.

What’s the memento design pattern?

Sometimes it’s necessary to record the internal state of an object. This is required when implementing checkpoints and “undo” mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere, so that you can restore objects to their previous conditions. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this

state would violate encapsulation, which can compromise the application’s reliability and extensibility. The Memento pattern can be used to accomplish this without exposing the object’s internal structure.

What’s the template design pattern?

The Template Design Pattern is a behavior pattern and, as the name suggests, it provides a template or a structure of an algorithm, which is used by users. A user provides its own implementation without changing the algorithm’s structure. The Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses to redefine certain steps of an algorithm without changing the algorithm’s structure.

What’s the state design pattern?

The State Design Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed.

What’s the strategy design pattern?

The Strategy Design Pattern seems to be the simplest of all design patterns, yet it provides great flexibility to your code. This pattern is used almost everywhere, even in

conjunction with the other design patterns. The Strategy Design Pattern defines a family of algorithms, encapsulating each one, and making them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

What’s the command design pattern?

The Command Design Pattern is a behavioral design pattern and helps to decouple the invoker from the receiver of a request. The intent of the Command Design Pattern is to encapsulate a request as an object, thereby letting the developer to parameterize clients with different requests, queue or log requests, and support undoable operations.

What’s the interpreter design pattern?

The Interpreter Design Pattern is a heavy-duty pattern. It’s all about putting together your own programming language, or handling an existing one, by creating an interpreter for that language. Given a language, we can define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

What’s the decorator design pattern?

The intent of the Decorator Design Pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality. The pattern is used to extend the functionality of an object dynamically without having to change the original class source or using inheritance. This is accomplished by creating an object wrapper referred to as a Decorator around the actual object.

What’s the iterator design pattern?

The intent of the Iterator Design Pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents.

What’s the visitor design pattern?

The Visitor Design Pattern provides you with a way to add new operations on the objects without changing the classes of the elements, especially when the operations change quite often. The intent of the Visitor Design Pattern is to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

What’s the collection design pattern?

What’s the combined design pattern?

What’s the ducks design pattern?

What’s the template method design pattern?

What is the difference between State and Strategy patterns?

While the implementation is similar they solve different problems. The State
pattern deals with what state an object is in - it encapsulates state-dependent behavior. The Strategy pattern deals with how an object performs a certain task – it encapsulates an algorithm.

What is the difference between Strategy and Template Method patterns?

In Template Method, the algorithm is chosen at compile time via inheritance.
With Strategy pattern the algorithm is chosen at runtime via composition.

What is the difference between Proxy and Decorator patterns?

The difference is the intent of the patterns. While Proxy controls access to
the object Decorator is used to add responsibilities to the object.

What is the difference between Chain of Responsibility and Intercepting Filter patterns?

While the implementations look similar there are differences. The Chain of
Responsibility forms a chain of request processors and the processors are then executed one by one until the correct processor is found. In Intercepting Filter the chain is constructed from filters and the whole chain is always executed.

What is the difference between Visitor and Double Dispatch patterns?

The Visitor pattern is a means of adding a new operation to existing classes.
Double dispatch is a means of dispatching function calls with respect to two
polymorphic types, rather than a single polymorphic type, which is what
languages like C++ and Java _do not_ support directly.

What are the differences between Flyweight and Object Pool patterns?

They differ in the way they are used. Pooled objects can simultaneously be used by a single "client" only. For that, a pooled object must be checked out from the pool, then it can be used by a client, and then the client must return the object back to the pool. Multiple instances of identical objects may exist, up to the maximal capacity of the pool. In contrast, a Flyweight object is singleton, and it can be used simultaneously by multiple clients.

As for concurrent access, pooled objects can be mutable and they usually don't need to be thread safe, as typically, only one thread is going to use a
specific instance at the same time. Flyweight must either be immutable (the
best option), or implement thread safety.

As for performance and scalability, pools can become bottlenecks, if all the
pooled objects are in use and more clients need them, threads will become

blocked waiting for available object from the pool. This is not the case with
Flyweight.

What are the differences between FluentInterface and Builder patterns?

Fluent interfaces are sometimes confused with the Builder pattern, because they share method chaining and a fluent usage. However, fluent interfaces are not primarily used to create shared (mutable) objects, but to configure complex objects without having to respecify the target object on every property change.

What is the difference between Java.io.Serialization and Memento pattern?

Memento is typically used to implement rollback/save-point support. Example we might want to mark the state of an object at a point in time, do some work and then decide to rollback to the previous state.

On the other hand serialization may be used as a tool to save the state of an object into byte[] and preserving the contents in memory or disk. When someone invokes the memento to revert object's previous state then we can deserialize the information stored and recreate previous state.

So, Memento is a pattern and serialization is a tool that can be used to implement this pattern. Other ways to implement the pattern can be to clone the contents of the object and keep track of those clones.

What’s Object Oriented Programming (OOP)?

Java is a computer programming language that is concurrent, class-based and object-oriented. The advantages of object oriented software development are shown below:

a. Modular development of code, which leads to easy maintenance and modification.
b. Reusability of code
c. Improved reliability and flexibility of code.
d. Increased understanding of code.

Object-oriented programming contains many significant features, such as encapsulation, inheritance, polymorphism and abstraction. We analyze each feature separately in the following sections.

What’s encapsulation?

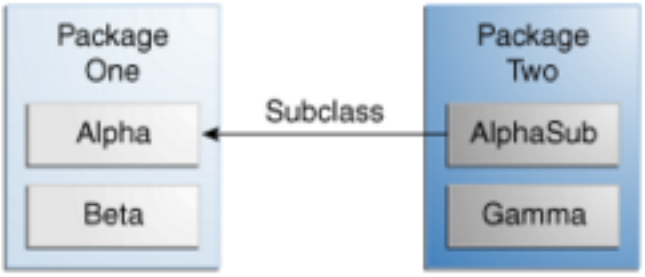
Encapsulation provides objects with the ability to hide their internal characteristics and behavior. Each object provides a number of methods, which can be accessed by other objects and change its internal data.

In Java, there are 3 access modifiers: public, private and protected. Each modifier imposes different access rights to other classes, either in the same or in external packages. Some of the advantages of using encapsulation are listed below:

- Hiding its attributes protects the internal state of every object.
- It increases usability and maintenance of code, because the behavior of an object can be independently changed or extended.
- It improves modularity by preventing objects to interact with each other, in an undesired way.

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N



Classes and Packages of the Example Used to Illustrate Access Levels

Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

What's the Polymorphism?

Polymorphism is the ability of programming languages to present the same interface for differing underlying data types. A polymorphic type is a type whose operations can also be applied to values of some other type. The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different

forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
                int startCadence,
                int startSpeed,
                int startGear,
                String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
      return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Polymorphism can be demonstrated with a minor modification to the Bicycle class. For example, a printDescription method could be added to the class that displays all the data currently stored in an instance.

```
public void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

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

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

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

What's the Inheritance?

Inheritance provides an object with the ability to acquire the fields and methods of another class, called base class. Inheritance provides re-usability of code and can be used to add additional features to an existing class, without modifying it.

What's the Abstraction?

Abstraction is the process of separating ideas from specific instances. Thus, develop classes in terms of their own functionality, instead of their implementation details. Java supports the creation and existence of abstract classes that expose interfaces, without including the actual implementation of all methods. The abstraction technique aims to separate the implementation details of a class from its behavior.

What's the difference between Abstraction and Encapsulation?

Abstraction and encapsulation are complementary concepts. On the one hand, abstraction focuses on the behavior of an

object. On the other hand, encapsulation focuses on the implementation of an object's behavior. Encapsulation is usually achieved by hiding information about the internal state of an object and thus, can be seen as a strategy used in order to provide abstraction.

What is JVM? Why is Java called the "Platform Independent Programming Language"?

A Java virtual machine (JVM) is a process virtual machine that can execute Java bytecode. Each Java source file is compiled into a bytecode file, which is executed by the JVM. Java was designed to allow application programs to be built that could be run on any platform, without having to be rewritten or recompiled by the programmer for each separate platform. A Java virtual machine makes this possible, because it is aware of the specific instruction lengths and other particularities of the underlying hardware platform.

What is the Difference between JDK and JRE?

The Java Runtime Environment (JRE) is basically the Java Virtual Machine (JVM) where your Java programs are being executed. It also includes browser plugins for applet execution. The Java Development Kit (JDK) is the full-featured Software Development Kit for Java, including the JRE, the compilers and tools (like JavaDoc, and Java Debugger), in order for a user to develop, compile and execute Java applications.

What does the "static" keyword mean? Can you override private or static method in Java?

The static keyword denotes that a member variable or method can be accessed, without requiring an instantiation of the class to which it belongs. A user cannot override static methods in Java, because method overriding is based upon

dynamic binding at runtime and static methods are statically bind at compile time. A static method is not associated with any instance of a class so the concept is not applicable.

Can you access non-static variable in static context?

A static variable in Java belongs to its class and its value remains the same for all its instances. A static variable is initialized when the class is loaded by the JVM. If your code tries to access a non-static variable, without any instance, the compiler will complain, because those variables are not created yet and they are not associated with any instance.

What are the Data Types supported by Java? What is Autoboxing and Unboxing?

The 8 primitive data types supported by the Java programming language are:

- **byte**
- **short**
- **int**
- **long**
- **float**
- **double**
- **boolean**
- **char**

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

What is Function Overriding and Overloading in Java?

Method overloading in Java occurs when two or more methods in the same class have the exact same name, but different parameters. On the other hand, method overriding is defined as the case when a child class redefines the same method as a parent class. Overridden methods must have the same name, argument list, and return type. The overriding method may not limit the access of the method it overrides.

What is a Constructor, Constructor Overloading in Java and Copy-Constructor?

A constructor gets invoked when a new object is created. Every class has a constructor. In case the programmer does not provide a constructor for a class, the Java compiler (Javac) creates a default constructor for that class. The constructor overloading is similar to method overloading in Java. Different constructors can be created for a single class. Each constructor must have it’s own unique parameter list. Finally, Java does support copy constructors like C++, but the difference lies in the fact that Java doesn’t create a default copy constructor if you don’t write your own.

Does Java support multiple inheritances?

No, Java does not support multiple inheritances. Each class is able to extend only on one class, but is able to implement more than one interface.

What is the difference between an Interface and an Abstract class?

Java provides and supports the creation both of abstract classes and interfaces. Both implementations share some common characteristics, but they differ in the following features:

i. All methods in an interface are implicitly abstract. On the other hand, an abstract class may contain both abstract and non-abstract methods.

ii. A class may implement a number of Interfaces, but can extend only one abstract class.

iii. In order for a class to implement an interface, it must implement all its declared methods. However, a class may not implement all declared methods of an abstract class. Though, in this case, the sub-class must also be declared as abstract.

iv. Abstract classes can implement interfaces without even providing the implementation of interface methods.

v. Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.

vi. Members of a Java interface are public by default. A member of an abstract class can either be private, protected or public.

vii. An interface is absolutely abstract and cannot be instantiated. An abstract class also cannot be instantiated, but can be invoked if it contains a main method.

What are pass by reference and pass by value?

When an object is passed by value, this means that a copy of the object is passed. Thus, even if changes are made to that object, it doesn’t affect the original value. When an object is passed by reference, this means that the actual object is not passed, rather a reference of the object is passed. Thus, any changes made by the external method, are also reflected in all places.

==========

JENKOV TUTS <http://tutorials.jenkov.com/java-concurrency/index.html >

JAVA CODE GEEKS MULTI-THREADING QUESTIONS <https://www.javacodegeeks.com/2014/11/multithreading-concurrency-interview-questions-answers.html>

What do we understand by the term concurrency?

What is the difference between processes and threads?

In Java, what is a process and a thread?

What is a scheduler?

How many threads does a Java program have at least?

How can a Java application access the current thread?

What properties does each Java thread have?

What is the purpose of thread groups?

What states can a thread have and what is the meaning of each state?

How do we set the priority of a thread?

How is a thread created in Java?

How do we stop a thread in Java?

Why should a thread not be stopped by calling its method stop()?

Is it possible to start a thread twice?

What is the output of the following code?

What is a daemon thread?

Is it possible to convert a normal user thread into a daemon thread after it has been started?

What do we understand by busy waiting?

How can we prevent busy waiting?

Can we use Thread.sleep() for real-time processing?

How can a thread be woken up that has been put to sleep before using Thread.sleep()?

How can a thread query if it has been interrupted?

How should an InterruptedException be handled?

After having started a child thread, how do we wait in the parent thread for the termination of the child thread?

What is the output of the following program?

What happens when an uncaught exception leaves the run() method?

What is a shutdown hook?

For what purposes is the keyword synchronized used?

What intrinsic lock does a synchronized method acquire?

Can a constructor be synchronized?

Can primitive values be used for intrinsic locks?

Are intrinsic locks reentrant?

What do we understand by an atomic operation?

Is the statement c++ atomic?

What operations are atomic in Java?

Is the following implementation thread-safe?

What do we understand by a deadlock?

What are the requirements for a deadlock situation?

Is it possible to prevent deadlocks at all?

Is it possible to implement a deadlock detection?

What is a livelock?

What do we understand by thread starvation?

Can a synchronized block cause thread starvation?

What do we understand by the term race condition?

What do we understand by fair locks?

Which two methods that each object inherits from java.lang.Object can be used to implement a simple producer/consumer scenario?

What is the difference between notify() and notifyAll()?

How it is determined which thread wakes up by calling notify()?

Is the following code that retrieves an integer value from some queue implementation correct?

Is it possible to check whether a thread holds a monitor lock on some given object?

What does the method Thread.yield() do?

What do you have to consider when passing object instances from one thread to another?

Which rules do you have to follow in order to implement an immutable class?

What is the purpose of the class java.lang.ThreadLocal?

What are possible use cases for java.lang.ThreadLocal?

Is it possible to improve the performance of an application by the usage of multi-threading? Name some examples.

What do we understand by the term scalability?

Is it possible to compute the theoretical maximum speed up for an application by using multiple processors?

What do we understand by lock contention?

Which techniques help to reduce lock contention?

Which technique to reduce lock contention can be applied to the following code?

Explain by an example the technique lock splitting.

What kind of technique for reducing lock contention is used by the SDK class ReadWriteLock?

What do we understand by lock striping?

What do we understand by a CAS operation?

Which Java classes use the CAS operation?

Provide an example why performance improvements for single-threaded applications can cause performance degradation for multi-threaded applications.

Is object pooling always a performance improvement for multi-threaded applications?

What is the relation between the two interfaces Executor and ExecutorService?

What happens when you submit() a new task to an ExecutorService instance whose queue is already full?

What is a ScheduledExecutorService?

Do you know an easy way to construct a thread pool with 5 threads that executes some tasks that return a value?

What is the difference between the two interfaces Runnable and Callable?

Which are use cases for the class java.util.concurrent.Future?

What is the difference between HashMap and Hashtable particularly with regard to thread-safety?

Is there a simple way to create a synchronized instance of an arbitrary implementation of Collection, List or Map?

What is a semaphore?

What is a CountDownLatch?

What is the difference between a CountDownLatch and a CyclicBarrier?

What kind of tasks can be solved by using the Fork/Join framework?

Is it possible to find the smallest number within an array of numbers using the Fork/Join-Framework?

What is the difference between the two classes RecursiveTask and RecursiveAction?

Is it possible to perform stream operations in Java 8 with a thread pool?

How can we access the thread pool that is used by parallel stream operations?

==========

What is the difference between processes and threads?

A process is an execution of a program, while a Thread is a single execution sequence within a process. A process can contain multiple threads. A Thread is sometimes called a lightweight process.

Explain different ways of creating a thread. Which one would you prefer and why?

There are 3 ways that can be used in order for a Thread to be created:

i. A class may extend the **Thread** class.
ii. A class may implement the **Runnable** interface.
iii. An application can use the Executor framework, in order to create a **thread pool**

The Runnable interface is preferred, as it does not require an object to inherit the Thread class. In case your application design requires multiple inheritances, only interfaces can help you. Also, the thread pool is very efficient and can be implemented and used very easily.

Explain the available thread states in a high-level.

During its execution, a thread can reside in one of the following states:

i. **NEW**: The thread becomes ready to run, but does not necessarily start running immediately.
ii. **RUNNABLE**: The Java Virtual Machine (JVM) is actively executing the thread's code.
iii. **BLOCKED**: The thread is in a blocked state while waiting for a monitor lock.
iv. **WAITING**: The thread waits for another thread to perform a particular action.
v. **TIMED_WAITING**: The thread waits for another thread to perform a particular action up to a specified waiting time.
vi. **TERMINATED**: The thread has finished its execution.

What is the difference between a synchronized method and a synchronized block?

In Java programming, each object has a lock. A thread can acquire the lock for an object by using the synchronized keyword. The synchronized keyword can be applied in a method level (coarse grained lock) or block level of code.

How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply?

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian that watches over a sequence of synchronized code and ensuring that only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. The thread is not allowed to execute the code until it obtains the lock.

What's a deadlock?

Deadlock is a condition that occurs when two processes are waiting for each other to complete, before proceeding. The result is that both processes wait endlessly.

How do you ensure that N threads can access N resources without deadlock?

A very simple way to avoid deadlock while using N threads is to impose an ordering on the locks and force each thread to follow that ordering. Thus, if all threads lock and unlock the murexes in the same order, no deadlocks can arise.

What are the basic interfaces of Java Collections Framework?

Java Collections Framework provides a well-designed set of interfaces and classes that support operations on collections of objects. The most basic interfaces that reside in the Java Collections Framework are:

- **Collection**, which represents a group of objects known as its elements.
- **Set**, which is a collection that cannot contain duplicate elements.
- **List**, which is an ordered collection and can contain duplicate elements.
- **Map**, which is an object that maps keys to values and cannot contain duplicate keys.

Why Collection doesn’t extend Cloneable and Serializable interfaces?

The Collection interface specifies groups of objects known as elements. Each concrete implementation of a Collection can choose its own way of how to maintain and order its elements. Some collections allow duplicate keys, while some other collections don’t. The semantics and the implications of either cloning or serialization come into play when dealing with actual implementations. Thus, the concrete

implementations of collections should decide how they could be cloned or serialized.

What is an Iterator?

The Iterator interface provides a number of methods that are able to iterate over any Collection. Each Java Collection contains the iterator method that returns an Iterator instance. Iterators are capable of removing elements from the underlying collection during the iteration.

How's the growth rate graph of Fibonacci series?

Exponential

What differences exist between Iterator and ListIterator?

The differences of these elements are listed below:

a. An Iterator can be used to traverse the Set and List collections, while the ListIterator can be used to iterate only over Lists.

b. The Iterator can traverse a collection only in forward direction, while the ListIterator can traverse a List in both directions.

c. The ListIterator implements the Iterator interface and contains extra functionality, such as adding an element, replacing an element, getting the index position for previous and next elements, etc.

What is difference between the Iterator's fail-fast and fail-safe?

The Iterator's fail-safe property works with the clone of the underlying collection and thus, it is not affected by any modification in the collection. All the collection classes in java.util package are fail-fast, while the collection classes in java.util.concurrent are fail-safe. Fail-fast iterators throw a ConcurrentModificationException, while fail-safe iterator never throws such an exception.

How HashMap works in Java?

A HashMap in Java stores key-value pairs. The HashMap requires a hash function and uses hashCode and equals methods, in order to put and retrieve elements to and from the collection respectively. When the put method is invoked, the HashMap calculates the hash value of the key and stores the pair in the appropriate index inside the collection. If the key exists, its value is updated with the new value. Some important characteristics of a HashMap are its capacity, its load factor and the threshold resizing.

What is the importance of hashCode() and equals() methods ?

HashMap uses the hashCode and equals methods to determine the index of the key-value pair and to detect duplicates. More specifically, the hashCode method is used in order to determine where the specified key will be stored. Since different keys may produce the same hash value, the equals method is used, in order to determine whether the specified key actually exists in the collection or not. Therefore, the implementation of both methods is crucial to the accuracy and efficiency of the HashMap.

What differences exist between HashMap and Hashtable?

Both the HashMap and Hashtable classes implement the Map interface and thus, have very similar characteristics. However, they differ in the following features:

a. A HashMap allows the existence of **null keys and values**, while a Hashtable doesn't allowed neither null keys, nor null values.

b. A Hashtable is synchronized, while a HashMap is not. Thus, HashMap is preferred in single-threaded environments, while a Hashtable is suitable for multi-threaded environments.

c. A HashMap provides its set of keys and a Java application can iterate over them. Thus, a HashMap is fail-fast. On the other hand, a Hashtable provides an Enumeration of its keys.

d. The Hashtable class is considered to be a legacy class.

What is difference between Array and ArrayList? When will you use Array over ArrayList?

The Array and ArrayList classes differ on the following features:

a. Arrays can contain primitive or objects, while an ArrayList can contain only objects.

b. Arrays have fixed size, while an ArrayList is dynamic.

c. An ArrayList provides more methods and features, such as addAll, removeAll, iterator, etc.

d. For a list of primitive data types, the collections use autoboxing (say, convert from int to Integer) to reduce the coding effort. However, this approach makes them slower when working on fixed size primitive data types.

What is difference between ArrayList and LinkedList?

Both the ArrayList and LinkedList classes implement the List interface, but they differ on the following features:

a. An ArrayList is an index based data structure backed by an Array. It provides random access to its elements with a performance equal to O(1). On the other hand, a LinkedList stores its data as list of elements and every element is linked to its previous and next element. In this case, the search operation for an element has execution time equal to O(n).

b. The Insertion, addition and removal operations of an element are faster in a LinkedList compared to an ArrayList, because there is no need of resizing an array or updating the index when an element is added in some arbitrary position inside the collection.

c. A LinkedList consumes more memory than an ArrayList, because every node in a LinkedList stores two references, one for its previous element and one for its next element. ArrayList is implemented on Array.

	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time

(roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Time Complexity of O(1)

a. isEmpty() add(x)
b. add(x, i) // need to be sure, it may be O(n)
c. set(x, i)
d. size() get(i)
e. remove(i)

Time Complexity of O(N)

a. indexof(x)
b. clear() remove(x) remove(i)

What is Comparable and Comparator interface? List their differences

Comparable interface contains only one method, called compareTo. This method compares two objects, in order to impose an order between them. Specifically, it returns a negative integer, zero, or a positive integer to indicate that the input object is less than, equal or greater than the existing object.

Comparator interface contains two methods, called compare and equals. The first method compares its two input arguments and imposes an order between them. It returns a negative integer, zero, or a positive integer to indicate that the first argument is less than, equal to, or greater than the second. The equal's method requires an object as a parameter and aims to decide whether the input object is equal to the comparator. The method returns true, only if

the specified object is also a comparator and it imposes the same ordering as the comparator.

What is Java Priority Queue?

A priority queue is an abstract data type, which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

The **PriorityQueue** is an unbounded queue, based on a priority heap and its elements are ordered in their natural order. At the time of its creation, we can provide a Comparator that is responsible for ordering the elements of the PriorityQueue. A PriorityQueue doesn't allow null values, those objects that doesn't provide natural ordering, or those objects that don't have any comparator associated with them. Finally, the Java PriorityQueue is not thread-safe and it requires **O(log(n))** time for its enqueing and dequeing operations.

What do you know about the big-O notation and can you give some examples with respect to different data structures?

The Big-O notation simply describes how well an algorithm scales or performs in the worst-case scenario as the number of elements in a data structure increases. The Big-O notation can also be used to describe other behavior such as memory consumption. Since the collection classes are actually data structures, we usually use the Big-O notation to choose the best implementation to use, based on time, memory and performance. Big-O notation can give a good indication about performance for large amounts of data.

What is the tradeoff between using an unordered array versus an ordered array?

The major advantage of an ordered array is that the search times have time complexity of **O(log n)**, compared to that of an unordered array, which is O(n). The disadvantage of an ordered array is that the insertion operation has a time complexity of O(n), because the elements with higher values must be moved to make room for the new element. Instead, the insertion operation for an unordered array takes constant time of O(1).

What are some of the best practices relating to the Java Collection framework?

i. Choosing the right type of the collection to use, based on the application's needs, is very crucial for its performance. For example if the size of the elements is fixed and know a priori, we shall use an Array, instead of an ArrayList.

ii. Some collection classes allow us to specify their initial capacity. Thus, if we have estimation on the number of elements that will be stored, we can use it to avoid rehashing or resizing.

iii. Always use Generics for type-safety, readability, and robustness. Also, by using Generics you avoid the **ClassCastException** during runtime.

iv. Use immutable classes provided by the Java Development Kit (JDK) as a key in a Map, in order to avoid the implementation of the **hashCode** and **equals** methods for our custom class.

v. Program in terms of interface not implementation.

vi. Return **zero-length collections or arrays** as opposed to returning a **null** in case the underlying collection is actually empty.

What's the difference between Enumeration and Iterator interfaces?

Enumeration is twice as fast as compared to an Iterator and uses very less memory. However, the Iterator is much safer compared to Enumeration, because other threads are not able to modify the collection object that is currently traversed by the iterator. Also, Iterators allow the caller to remove elements from the underlying collection, something that is not possible with Enumerations.

What is the difference between HashSet and TreeSet?

The HashSet is implemented using a hash table and thus, its elements are not ordered. The add, remove, and contains methods of a HashSet have constant time complexity **O(1)**. On the other hand, a TreeSet is implemented using a tree structure. The elements in a TreeSet are sorted, and thus, the add, remove, and contains methods have time complexity of **O(logn)**.

What is the purpose of garbage collection in Java, and when is it used?

The purpose of garbage collection is to identify and discard those objects that are no longer needed by the application, in order for the resources to be reclaimed and reused.

What does System.gc() and Runtime.gc() methods do?

These methods can be used as a hint to the JVM, in order to start a garbage collection. However, this it is up to the Java Virtual Machine (JVM) to start the garbage collection immediately or later in time.

When is the finalize() called? What is the purpose of finalization?

The finalize method is called by the garbage collector, just before releasing the object's memory. It is normally advised to release resources held by the object inside the finalize method.

If an object reference is set to null, will the Garbage Collector immediately free the memory held by that object?

No, the object will be available for garbage collection in the next cycle of the garbage collector.

What is structure of Java Heap? What is Perm Gen space in Heap?

The JVM has a heap that is the runtime data area from which memory for all class instances and arrays is allocated. It is created at the JVM start-up. Heap memory for objects is reclaimed by an automatic memory management system, which is known as a garbage collector. Heap memory consists of live and dead objects. Live objects are accessible by the application and will not be a subject of garbage collection. Dead objects are those, which will never be accessible by the application, but have not been collected by the garbage collector yet. Such objects occupy the heap memory space until the garbage collector eventually collects them.

What is the difference between Serial and Throughput Garbage collector?

The throughput garbage collector uses a parallel version of the young generation collector and is meant to use with applications that have medium to large data sets. On the other hand, the serial collector is usually adequate for most small applications (those requiring heaps of up to approximately 100MB on modern processors).

When does an Object become eligible for Garbage collection in Java?

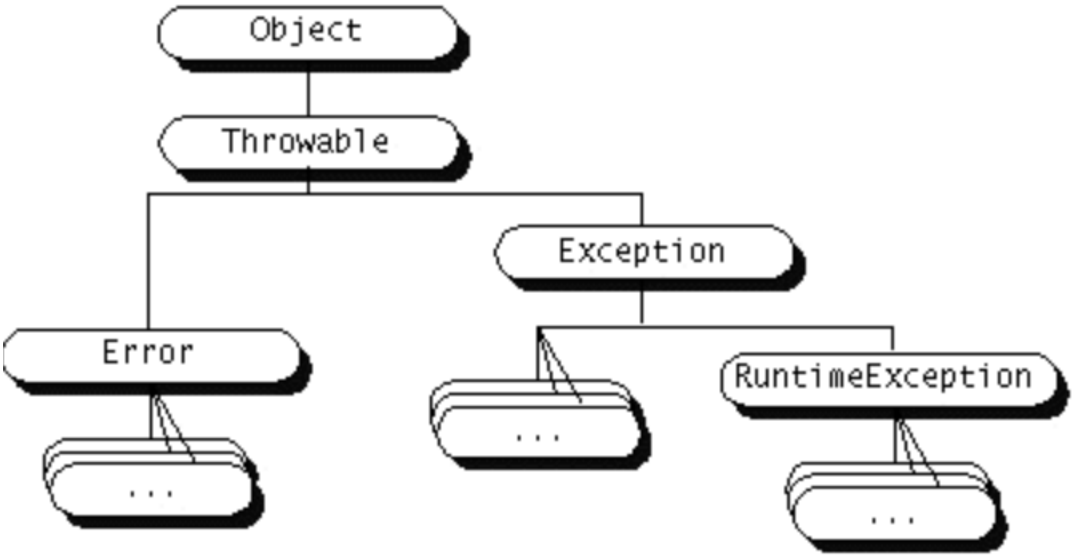
A Java object is subject to garbage collection when it becomes unreachable to the program in which it is currently used.

Does Garbage collection occur in permanent generation space in JVM?

Garbage Collection does occur in PermGen space and if PermGen space is full or cross a threshold, it can trigger a full garbage collection. If you look carefully at the output of the garbage collector, you will find that PermGen space is also garbage collected. This is the reason why correct sizing of PermGen space is important to avoid frequent full garbage collections.

What are the two types of Exceptions in Java? Which are the differences between them?

Java has two types of exceptions: checked exceptions and unchecked exceptions. Unchecked exceptions do not need to be declared in a method or a constructor’s throws clause, if they can be thrown by the execution of the method or the constructor, and propagate outside the method or constructor boundary. On the other hand, checked exceptions must be declared in a method or a constructor’s throws clause.



From the above hierarchy of Throwable class,checked and unchecked exceptions are classified as below

Unchecked: “Error” and its subclasses,”RunTimeException” and its subclasses

Checked: Every exception other than Unchecked exception

Checked Exception: These are the exceptions which may occur regularly in a program and compiler will check for those exceptions at “CompileTime” ,those exceptions are called Checked Exceptions

Example: FileNotFoundException, EndOfFileException etc.

So the compiler at compile time will check if a certain method is throwing any of the checked exceptions or not,if yes it will check whether the method is handling that exception either with “Try&Catch” or “throws”,if in case the

method is not providing the handling code then compiler will throw error saying “Unreported Exception”

For example

```
import java.io.*

Class Example {

    public static void main(String[] args) {

        PrintWriter   pw=new   PrintWriter("xyz.txt");
        //creating a new printwriter object to write
        in a file named "xyz.txt"

        pw.println("Hello World");

    }

}
```

The above snippet is supposed to print “Hello World” in file named “xyz.txt”

There may be a chance of file “xyz.txt” is not present in specified directory,so the compiler will check if any handling code is provided in case file is not present

In above snippet handling code is not provided either with “Try&Catch” or “throws” so the compiler will throw error

The same example with some modification:

```
import java.io.*

Class Example {

     public static void main(String[] args) throws
     FileNotFoundException {



          PrintWriter pw=new PrintWriter("xyz.txt");
          //creating a new printwriter object to write in a
          file named "xyz.txt"

          pw.println("Hello World");

     }

}
```

In this example handling code is provided with “throws” so compiler will not throw any error.

Unchecked Exceptions:

There are some exceptions which do not occur regularly in a program,and compiler will not check for those exceptions,these kind of exceptions are called Unchecked Exceptions

Example: ArithmeticException, NullPointerException etc

For example:

```
class Example{

    public static void main(String[] args){

        System.out.println(10/0);          //Arithmetic
        Exception

    }

}
```

The above program should throw “Arithmetic Exception” as division with “0” is not allowed

In this case the program compiles fine because compiler will not check for “Unchecked Exceptions” but the program will throw error at “Run Time” as division with “0” is illegal.

P.S: All the exceptions occurs at “Run Time” only.

writing this line because many people think that “Checked Exceptions” occurs at Compile time and “Unchecked Exceptions” occurs at Run time,which is not the case.

What is the difference between Exception and Error in java?

Exception and Error classes are both subclasses of the Throwable class. The Exception class is used for exceptional

conditions that a user's program should catch. The Error class defines exceptions that have not expected to be caught by the user program.

What is the difference between throw and throws?

The throw keyword is used to explicitly raise an exception within the program. On the contrary, the throws clause is used to indicate those exceptions that are not handled by a method. Each method must explicitly specify which exceptions do not handle, so the callers of that method can guard against possible exceptions. Finally, a comma separates multiple exceptions.

What is the importance of finally block in exception handling?

A finally block will always be executed, whether or not an exception is actually thrown. Even in the case where the catch statement is missing and an exception is thrown, the finally block will still be executed. Last thing to mention is that the finally block is used to release resources like I/O buffers, database connections, etc.

What will happen to the Exception object after exception handling?

The Exception object will be garbage collected in the next garbage collection.

How does finally block differ from finalize() method?

A finally block will be executed whether or not an exception is thrown and is used to release those resources held by the application. Finalize is a protected method of the Object class, which is called by the Java Virtual Machine (JVM) just before an object is garbage collected.

What is an Applet?

A java applet is program that can be included in a HTML page and be executed in a java enabled client browser. Applets are used for creating dynamic and interactive web applications.

Explain the life cycle of an Applet.

An applet may undergo the following states:

- **Init**: An applet is initialized each time is loaded.
- **Start**: Begin the execution of an applet.
- **Stop**: Stop the execution of an applet.
- **Destroy**: Perform a final cleanup, before unloading the applet.

What happens when an applet is loaded?

First of all, an instance of the applet’s controlling class is created. Then, the applet initializes itself and finally, it starts running.

What is the difference between an Applet and a Java Application?

Applets are executed within a java-enabled browser, but a Java application is a standalone Java program that can be executed outside of a browser. However, they both require the existence of a Java Virtual Machine (JVM). Furthermore, a Java application requires a main method with a specific signature, in order to start its execution. Java applets don’t need such a method to start their execution. Finally, Java applets typically use a restrictive security policy, while Java applications usually use more relaxed security policies.

What are the restrictions imposed on Java applets?

Mostly due to security reasons, the following restrictions are imposed on Java applets:

i. An applet can’t load libraries or define native methods.

ii. An applet cannot ordinarily read or write files on the execution host.

iii. An applet cannot read certain system properties.

iv. An applet cannot make network connections except to the host that it came from.

v. An applet cannot start any program on the host that’s executing it.

What are untrusted applets?

Untrusted applets are those Java applets that cannot access or execute local system files. By default, all downloaded applets are considered as untrusted.

What is the difference between applets loaded over the Internet and applets loaded via the file system?

Regarding the case where an applet is loaded over the Internet, the applet is loaded by the applet classloader and is subject to the restrictions enforced by the applet security manager. Regarding the case where an applet is loaded from the client’s local disk, the file system loader loads the applet. Applets loaded via the file system are allowed to read files, write files and to load libraries on the client. Also, applets loaded via the file system are allowed to execute processes and finally, applets loaded via

the file system are not passed through the byte code verifier.

What is the applet class loader, and what does it provide?

When an applet is loaded over the Internet, the applet is loaded by the applet classloader. The class loader enforces the Java name space hierarchy. Also, the class loader guarantees that a unique namespace exists for classes that come from the local file system, and that a unique namespace exists for each network source. When a browser loads an applet over the net, that applet's classes are placed in a private namespace associated with the applet's origin. Then, those classes loaded by the class loader are passed through the verifier.The verifier checks that the class file conforms to the Java language specification . Among other things, the verifier ensures that there are no stack overflows or underflows and that the parameters to all bytecode instructions are correct.

What is the applet security manager, and what does it provide?

The applet security manager is a mechanism to impose restrictions on Java applets. A browser may only have one security manager. The security manager is established at startup, and it cannot thereafter be replaced, overloaded, overridden, or extended.

What is the difference between a Choice and a List?

A Choice is displayed in a compact form that must be pulled down, in order for a user to be able to see the list of all available choices. Only one item may be selected from a Choice. A List may be displayed in such a way that several List items are visible. A List supports the selection of one or more List items.

What is a layout manager?

A layout manager is the used to organize the components in a container.

What is the difference between a Scrollbar and a JScrollPane?

A Scrollbar is a Component, but not a Container. A ScrollPane is a Container. A ScrollPane handles its own events and performs its own scrolling.

Which Swing methods are thread-safe?

There are only three thread-safe methods: repaint, revalidate, and invalidate.

Name three Component subclasses that support painting

The Canvas, Frame, Panel, and Applet classes support painting.

What is clipping?

Clipping is defined as the process of confining paint operations to a limited area or shape.

What is the difference between a MenuItem and a CheckboxMenuItem?

The CheckboxMenuItem class extends the MenuItem class and supports a menu item that may be either checked or unchecked.

How are the elements of a BorderLayout organized?

The elements of a BorderLayout are organized at the borders (North, South, East, and West) and the center of a container.

How are the elements of a GridBagLayout organized?

The elements of a GridBagLayout are organized according to a grid. The elements are of different sizes and may occupy more than one row or column of the grid. Thus, the rows and columns may have different sizes.

What is the difference between a Window and a Frame?

The Frame class extends the Window class and defines a main application window that can have a menu bar.

What is the relationship between clipping and repainting?

When a window is repainted by the AWT painting thread, it sets the clipping regions to the area of the window that requires repainting.

What is the relationship between an event-listener interface and an event-adapter class?

An event-listener interface defines the methods that must be implemented by an event handler for a particular event. An event adapter provides a default implementation of an event-listener interface.

How can a GUI component handle its own events?

A GUI component can handle its own events, by implementing the corresponding event-listener interface and adding itself as its own event listener.

What advantage do Java's layout managers provide over traditional windowing systems?

Java uses layout managers to lay out components in a consistent manner, across all windowing platforms. Since layout managers aren't tied to absolute sizing and positioning, they are able to accommodate platform-specific differences among windowing systems.

What is the design pattern that Java uses for all Swing components?

The design pattern used by Java for all Swing components is the Model View Controller (MVC) pattern.

What is JDBC?

JDBC is an abstraction layer that allows users to choose between databases. JDBC enables developers to write database applications in Java, without having to concern themselves with the underlying details of a particular database.

Explain the role of Driver in JDBC

The JDBC Driver provides vendor-specific implementations of the abstract classes provided by the JDBC API. Each driver must provide implementations for the following classes of the java.sql package:

a. Connection
b. Statement
c. PreparedStatement
d. CallableStatement
e. ResultSet
f. Driver

What is the purpose Class.forName method?

This method is used to method is used to load the driver that will establish a connection to the database.

What is the advantage of PreparedStatement over Statement?

PreparedStatements are precompiled and thus,their performance is much better. Also, PreparedStatement objects can be reused with different input values to their queries.

What is the use of CallableStatement ? Name the method, which is used to prepare a CallableStatement

A CallableStatement is used to execute stored procedures. Stored procedures are stored and offered by a database. Stored procedures may take input values from the user and may return a result. The usage of stored procedures is highly encouraged, because it offers security and modularity.The method that prepares a CallableStatement is the following: CallableStament.prepareCall()

What does Connection pooling mean?

The interaction with a database can be costly, regarding the opening and closing of database connections. Especially, when the number of database clients increases, this cost is very high and a large number of resources is consumed.A pool of database connections is obtained at start up by the application server and is maintained in a pool. A request for a connection is served by a connection residing in the pool. In the end of the connection, the request is returned to the pool and can be used to satisfy future requests.

What is RMI?

The Java Remote Method Invocation (Java RMI) is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection. Remote Method Invocation (RMI) can also be seen as the process of activating a method on a remotely running object.

RMI offers location transparency because a user feels that a method is executed on a locally running object. Check some RMI Tips here.

What is the basic principle of RMI architecture?

The RMI architecture is based on a very important principle, which states that the definition of the behavior and the implementation of that behavior, are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

What are the layers of RMI Architecture?

The RMI architecture consists of the following layers:

- **Stub and Skeleton layer**: This layer lies just beneath the view of the developer. This layer is responsible for intercepting method calls made by the client to the interface and redirect these calls to a remote RMI Service.
- **Remote Reference Layer**: The second layer of the RMI architecture deals with the interpretation of references made from the client to the server's remote objects. This layer interprets and manages references made from clients to the remote service objects. The connection is a one-to-one (unicast) link.
- **Transport layer**: This layer is responsible for connecting the two JVM participating in the service. This layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

What is the role of Remote Interface in RMI?

The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. A class that implements a remote interface should declare the remote interfaces being implemented, define the constructor for each remote object and provide an implementation for each remote method in all remote interfaces.

What is the role of the java.rmi.Naming Class?

The java.rmi.Naming class provides methods for storing and obtaining references to remote objects in the remote object registry. Each method of the Naming class takes as one of its arguments a name that is a String in URL format.

What does binding in RMI mean?

Binding is the process of associating or registering a name for a remote object, which can be used at a later time, in order to look up that remote object. A remote object can be associated with a name using the bind or rebind methods of the Naming class.

What is the difference between using bind() and rebind() methods of Naming Class?

The bind method bind is responsible for binding the specified name to a remote object, while the rebind method is responsible for rebinding the specified name to a new remote object. In case a binding exists for that name, the binding is replaced.

What are the steps involved to make work a RMI program?

The following steps must be involved in order for a RMI program to work properly:

- Compilation of all source files.
- Generatation of the stubs using rmic.
- Start the rmiregistry.
- Start the RMIServer.
- Run the client program.

What is the role of stub in RMI?

A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub, which is responsible for executing the method on the remote object. When a stub's method is invoked, it undergoes the following steps:

- It initiates a connection to the remote JVM containing the remote object.
- It marshals the parameters to the remote JVM.
- It waits for the result of the method invocation and execution.
- It unmarshals the return value or an exception if the method has not been successfully executed.
- It returns the value to the caller.

What is DGC? And how does it work?

DGC stands for Distributed Garbage Collection. Remote Method Invocation (RMI) uses DGC for automatic garbage collection. Since RMI involves remote object references across JVM's, garbage collection can be quite difficult. DGC uses a reference counting algorithm to provide automatic memory management for remote objects.

What is the purpose of using RMISecurityManager in RMI?

RMISecurityManager provides a security manager that can be used by RMI applications, which use downloaded code. The class loader of RMI will not download any classes from remote locations, if the security manager has not been set.

Explain Marshalling and demarshalling

When an application wants to pass its memory objects across a network to another host or persist it to storage, the in-memory representation must be converted to a suitable format. This process is called marshalling and revert operation is called demarshalling.

Explain Serialization and Deserialization.

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes and includes the object's data, as well as information about the object's type, and the types of data stored in the object. Thus, serialization can be seen as a way of flattening objects, in order to be stored on disk, and later, read back and reconstituted. Deserialisation is the reverse process of converting an object from its flattened state to a live object.

What is a Servlet?

The servlet is a Java programming language class used to process client requests and generate dynamic web content. Servlets are mostly used to process or store data submitted by an HTML form, provide dynamic content and manage state information that does not exist in the stateless HTTP protocol.

Explain the architecture of a Servlet

The core abstraction that must be implemented by all servlets is the javax.servlet.Servlet interface. Each servlet must implement it either directly or indirectly, either by extending javax.servlet.GenericServlet or javax.servlet.http.HTTPServlet. Finally, each servlet is

able to serve multiple requests in parallel using multithreading.

What is the difference between an Applet and a Servlet?

An Applet is a client side java program that runs within a Web browser on the client machine. On the other hand, a servlet is a server side component that runs on the web server. An applet can use the user interface classes, while a servlet does not have a user interface. Instead, a servlet waits for client's HTTP requests and generates a response in every request.

What is the difference between GenericServlet and HttpServlet?

GenericServlet is a generalized and protocol-independent servlet that implements the Servlet and ServletConfig interfaces. Those servlets extending the GenericServlet class shall override the service method. Finally, in order to develop an HTTP servlet for use on the Web that serves requests using the HTTP protocol, your servlet must extend the HttpServlet instead.

Explain the life cycle of a Servlet

On every client's request, the Servlet Engine loads the servlets and invokes its init methods, in order for the servlet to be initialized. Then, the Servlet object handles all subsequent requests coming from that client, by invoking the service method for each request separately. Finally, the servlet is removed by calling the server's destroy method.

What is the difference between doGet() and doPost()?

doGET: The GET method appends the name-value pairs on the request's URL. Thus, there is a limit on the number of characters and subsequently on the number of values that can

be used in a client’s request. Furthermore, the values of the request are made visible and thus, sensitive information must not be passed in that way.

doPOST: The POST method overcomes the limit imposed by the GET request, by sending the values of the request inside its body. Also, there are no limitations on the number of values to be sent across. Finally, the sensitive information passed through a POST request is not visible to an external client.

What does a Web Application mean?

A Web application is a dynamic extension of a Web or application server. There are two types of web applications: presentation-oriented and service-oriented. A presentation-oriented Web application generates interactive web pages, which contain various types of markup language and dynamic content in response to requests. On the other hand, a service-oriented web application implements the endpoint of a web service. In general, a Web application can be seen as a collection of servlets installed under a specific subset of the server’s URL namespace.

What is a Server Side Include (SSI)?

Server Side Includes (SSI) is a simple interpreted server-side scripting language, used almost exclusively for the Web, and is embedded with a servlet tag. The most frequent use of SSI is to include the contents of one or more files into a Web page on a Web server. When a browser accesses a Web page, the Web server replaces the servlet tag in that Web page with the hypertext generated by the corresponding servlet.

What is Servlet Chaining?

Servlet Chaining is the method where the output of one servlet is sent to a second servlet. The output of the

second servlet can be sent to a third servlet, and so on. The last servlet in the chain is responsible for sending the response to the client.

How do you find out what client machine is making a request to your servlet?

The ServletRequest class has functions for finding out the IP address or host name of the client machine. getRemoteAddr() gets the IP address of the client machine and getRemoteHost() gets the host name of the client machine. See example here.

What is the structure of the HTTP response?

The HTTP response consists of three parts:

- **Status Code**: describes the status of the response. It can be used to check if the request has been successfully completed. In case the request failed, the status code can be used to find out the reason behind the failure. If your servlet does not return a status code, the success status code, HttpServletResponse.SC_OK, is returned by default.

- **HTTP Headers**: they contain more information about the response. For example, the headers may specify the date/time after which the response is considered stale, or the form of encoding used to safely transfer the entity to the user. See how to retrieve headers in Servlet here.

- **Body**: it contains the content of the response. The body may contain HTML code, an image, etc. The body consists of the data bytes transmitted in an HTTP transaction message immediately following the headers.

What is a cookie? What is the difference between session and cookie?

A cookie is a bit of information that the Web server sends to the browser. The browser stores the cookies for each Web server in a local file. In a future request, the browser, along with the request, sends all stored cookies for that specific Web server. The differences between session and a cookie are the following:

- The session should work, regardless of the settings on the client browser. The client may have chosen to disable cookies. However, the sessions still work, as the client has no ability to disable them in the server side.

- The session and cookies also differ in the amount of information the can store. The HTTP session is capable of storing any Java object, while a cookie can only store String objects.

Which protocol browser and servlet to communicate will use?

The browser communicates with a servlet by using the HTTP protocol.

What is HTTP Tunneling?

HTTP Tunneling is a technique by which, communications performed using various network protocols are encapsulated using the HTTP or HTTPS protocols. The HTTP protocol therefore acts as a wrapper for a channel that the network protocol being tunneled uses to communicate. The masking of other protocol requests as HTTP requests is HTTP Tunneling.

What’s the difference between sendRedirect and forward methods?

The sendRedirect method creates a new request, while the forward method just forwards a request to a new target. The previous request scope objects are not available after a redirect, because it results in a new request. On the other hand, the previous request scope objects are available after forwarding. FInally, in general, the sendRedirect method is considered to be slower compare to the forward method.

What are URL Encoding and URL Decoding?

The URL encoding procedure is responsible for replacing all the spaces and every other extra special character of a URL, into their corresponding Hex representation. In correspondence, URL decoding is the exact opposite procedure.

What is a JSP Page?

A Java Server Page (JSP) is a text document that contains two types of text: static data and JSP elements. Static data can be expressed in any text-based format, such as HTML or XML. JSP is a technology that mixes static content with dynamically generated content.

How are the JSP requests handled?

On the arrival of a JSP request, the browser first requests a page with a .jsp extension. Then, the Web server reads the request and using the JSP compiler, the Web server converts the JSP page into a servlet class. Notice that the JSP file is compiled only on the first request of the page or if the JSP file has changed. The generated servlet class is invoked, in order to handle the browser’s request. Once the execution of the request is over, the servlet sends a response back to the client.

What are the advantages of JSP?

The advantages of using the JSP technology are shown below:

- JSP pages are dynamically compiled into servlets and thus, the developers can easily make updates to presentation code.
- JSP pages can be pre-compiled.
- JSP pages can be easily combined to static templates, including HTML or XML fragments, with code that generates dynamic content.
- Developers can offer customized JSP tag libraries that page authors access using an XML-like syntax.
- Developers can make logic changes at the component level, without editing the individual pages that use the application’s logic.

What are Directives? What are the different types of Directives available in JSP?

Directives are instructions that are processed by the JSP engine, when the page is compiled to a servlet. Directives are used to set page-level instructions, insert data from external files, and specify custom tag libraries. Directives are defined between < %@ and % >.The different types of directives are shown below:

- Include directive: it is used to include a file and merges the content of the file with the current page.
- Page directive: it is used to define specific attributes in the JSP page, like error page and buffer.
- Taglib: it is used to declare a custom tag library which is used in the page.

What are JSP actions?

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. JSP actions are executed when a JSP page is requested. They can be dynamically inserted into a file, re-use JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.Some of the available actions are listed below:

- jsp:include – includes a file, when the JSP page is requested.
- jsp:useBean – finds or instantiates a JavaBean.
- jsp:setProperty – sets the property of a JavaBean.
- jsp:getProperty – gets the property of a JavaBean.
- jsp:forward – forwards the requester to a new page.
- jsp:plugin – generates browser-specific code.

What are Scriptlets?

In Java Server Pages (JSP) technology, a scriptlet is a piece of Java-code embedded in a JSP page. The scriptlet is everything inside the tags. Between these tags, a user can add any valid scriplet.

What are Declaration's?

Declarations are similar to variable declarations in Java. Declarations are used to declare variables for subsequent use in expressions or scriptlets. To add a declaration, you must use the sequences to enclose your declarations.

What are Expressions?

A JSP expression is used to insert the value of a scripting language expression, converted into a string, into the data stream returned to the client, by the web server. Expressions are defined between <% = and %>tags.

What do implicit objects mean and what are they?

JSP implicit objects are those Java objects that the JSP Container makes available to developers in each page. A developer can call them directly, without being explicitly declared. JSP Implicit Objects are also called pre-defined variables. The following objects are considered implicit in a JSP page:

- Application
- Page
- Request
- Response
- Session
- Exception
- Out
- Config
- PageContext

What are the thread class methods?

public void start(): start a thread by calling its run() method

public void run(): Entry point for the thread

public final void setName(String name): Set the name of the thread

public final void setPriority(int priority): To set the priority of the thread

public final void setDaemon(boolean on): A parameter of true denotes this Thread as a daemon thread.

void join(long millisec): Wait for a thread to terminate. This method when called from the parent thread makes parent thread wait till child thread terminates. The current thread invokes this method on a second thread, causing the current

thread to block until the second thread terminates or the specified number of milliseconds passes.

public void interrupt(): Interrupts this thread, causing it to continue execution if it was blocked for any reason.

public final boolean isAlive(): Determine if a thread is still running

The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

public static void yield(): Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.

public static void sleep(long millisec): Causes the currently running thread to block for at least the specified number of milliseconds.

public static boolean holdsLock(Object x): Returns true if the current thread holds the lock on the given Object.

public static Thread currentThread(): Returns a reference to the currently running thread, which is the thread that invokes this method.

public static void dumpStack(): Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

getName(): It is used for Obtaining a thread’s name

void getPriority(): Obtain a thread’s priority

What are the MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY in

the context of the threads?

There are some methods that can be use by the threads to communicate with each other. They are as following,

wait(): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

notify(): wakes up the first thread that called wait() on the same object.

notifyAll(): wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5). Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

What is the thread synchronization?

When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. To understand synchronization java has a concept of monitor. Monitor can be thought of as a box, which can hold only one thread. Once a thread enters the monitor all the other threads have to

wait until that thread exits the monitor.

What is the inter-thread communication?

Inter thread communication is important when you develop an application where two or more threads exchange some information.

public void wait(): Causes the current thread to wait until another thread invokes the notify().

public void notify(): Wakes up a single thread that is waiting on this object's monitor.

public void notifyAll(): Wakes up all the threads that called wait() on the same object.

These methods have been implemented as final methods in Object, so they are available in all the classes. All 3 methods can be called only from within a synchronized context.

What is the thread deadlock?

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

What is the thread control?

Core Java provides a complete control over multithreaded program. You can develop a multithreaded program, which can be suspended, resumed or stopped completely based on your

requirements. There are various static methods, which you can use on thread objects to control their behavior.

public void suspend(): This method puts a thread in suspended state and can be resumed using resume() method.

public void stop(): This method stops a thread completely.

public void resume(): This method resumes a thread which was suspended using suspend() method.

public void wait(): Causes the current thread to wait until another thread invokes the notify().

public void notify(): Wakes up a single thread that is waiting on this object's monitor.

Be aware that latest versions of Java has deprecated the usage of suspend(), resume() and stop() methods and so you need to use available alternatives.

Briefly describe the thread life cycle

The start method creates the system resources, necessary to run the thread, schedules the thread to run, and calls the thread’s run method.

A thread becomes “Not Runnable” when one of these events occurs:

If sleep method is invoked.

The thread calls the wait method.

The thread is blocking on I/O.

A thread dies naturally when the run method exits.

Briefly describe the thread scheduling

Execution of multiple threads on a single CPU, in some order, is called scheduling.

In general, the runnable thread with the highest priority is active (running)

Java is priority-preemptive

If a high-priority thread wakes up, and a low-priority thread is running. Then the high-priority thread gets to run immediately.

Allows on-demand processing

Efficient use of CPU

What are the types of the thread scheduling?

Waiting and Notifying

Waiting [wait()] and notifying [notify(), notifyAll()] provides means of communication between threads that synchronize on the same object.

wait(): when wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately and moves the thread to the wait state.

notify(): This wakes up threads that called wait() on the same object and moves the thread to ready state.

notifyAll(): This wakes up all the threads that called wait() on the same object.

Running and Yielding

Yield() is used to give the other threads of the same priority a chance to execute i.e. causes current running thread to move to runnable state.

Sleeping and Waking up

nSleep() is used to pause a thread for a specified period of time i.e. moves the current running thread to Sleep state for a specified amount of time, before moving it to runnable state. Thread.sleep (no. Of milliseconds);

What is the thread priority?

When a Java thread is created, it inherits its priority from the thread that created it. You can modify a thread's priority at any time after its creation using the setPriority() method.

Thread priorities are integers ranging between MIN_PRIORITY (1) and MAX_PRIORITY (10). The higher the integer, the higher the priority. Normally the thread priority will be 5.

What's blocking thread?

When reading from a stream, if input is not available, the thread will block

Thread is suspended ("blocked") until I/O is available

Allows other threads to automatically activate

When I/O available, thread wakes back up again

Becomes "runnable" i.e. gets into ready state

What's grouping thread?

Thread groups provide a mechanism for collecting multiple

threads into a single object and manipulating those threads all at once, rather than individually.

To put a new thread in a thread group the group must be explicitly specified when the thread is created.

public Thread(ThreadGroup group, Runnable runnable)

public Thread(ThreadGroup group, String name)

public Thread(ThreadGroup group, Runnable runnable, String name)

A thread can’t be moved to a new group after the thread has been created. When a Java application first starts up, the Java runtime system creates a ThreadGroup named main. The java.lang.ThreadGroup class implements Java thread groups.

What’s daemon thread?

Daemon thread is a low priority thread (in context of JVM) that runs in background to perform tasks such as garbage collection (gc) etc.; they do not prevent the JVM from exiting (even if the daemon thread itself is running) when all the user threads (non-daemon threads) finish their execution. JVM terminates itself when all user threads (non-daemon threads) finish their execution, JVM does not care whether Daemon thread is running or not, if JVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself.

i. A newly created thread inherits the daemon status of its parent. That’s the reason all threads created inside main method (child threads of main thread) are non-daemon by default, because main thread is non-daemon.

ii. Methods of Thread class that are related to Daemon threads as following,

public void setDaemon(boolean status): This method is used for making a user thread to Daemon thread or vice versa. For example if I have a user thread t then t.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread td then by calling td.setDaemon(false) would make it normal thread(user thread/non-daemon thread).

public boolean isDaemon(): This method is used for checking the status of a thread. It returns true if the thread is Daemon else it returns false.

setDaemon(): method can only be called before starting the thread. This method would throw IllegalThreadStateException if you call this method after Thread.start() method. (refer the example)

Why the thread JOIN() method is used?

The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution).

Difference between the calling run and the start method.

We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of multithreading. When the run method gets called though start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

On the other hand if the run() method of these threads are being called directly then the execution of all of them is being handled by the same current thread and no multithreading will take place, hence the output would reflect the sequential execution of threads in the specified order.

What is Spring Framework?

Spring is the most broadly used framework for the development of Java Enterprise Edition applications. The core features of Spring can be used in developing any Java application.
We can use its extensions for building various web applications on top of the Java EE platform, or we may just use its dependency injection provisions in simple standalone applications.

What exactly is Field Injection and how to avoid it?

Injection Types

There are three options for how dependencies can be injected into a bean:

- Through a constructor
- Through setters or other methods
- Through reflection, directly into fields

You are using option 3. That is what is happening when you use @Autowired directly on your field.

Injection guidelines

A general guideline, which is recommended by Spring (see the sections on Constructor-based DI or Setter-based DI) is the following:

For mandatory dependencies or when aiming for immutability, use constructor injection
For optional or changeable dependencies, use setter injection
Avoid field injection in most cases
Field injection drawbacks

The reasons why field injection is frowned upon are as follows:

- You cannot create immutable objects, as you can with constructor injection

- Your classes have tight coupling with your DI container and cannot be used outside of it

- Your classes cannot be instantiated (for example in unit tests) without reflection. You need the DI container to instantiate them, which makes your tests more like integration tests

- Your real dependencies are hidden from the outside and are not reflected in your interface (either constructors or methods)

- It is really easy to have like ten dependencies. If you

were using constructor injection, you would have a constructor with ten arguments, which would signal that something is fishy. But you can add injected fields using field injection indefinitely. Having too many dependencies is a red flag that the class usually does more than one thing, and that it may violate the Single Responsibility Principle.

Conclusion

Depending on your needs, you should primarily use constructor injection or some mix of constructor and setter injection. Field injection has many drawbacks and should be avoided. The only advantage of field injection is that it is more convenient to write, which does not outweigh all the cons.

What are the benefits of using Spring?

Spring targets to make Java EE development easier. Here are the advantages of using it:

- **Lightweight:** there is a slight overhead of using the framework in development
- **Inversion of Control (IoC):** Spring container takes care of wiring dependencies of various objects, instead of creating or looking for dependent objects
- **Aspect Oriented Programming (AOP):** Spring supports AOP to separate business logic from system services
- **IoC container:** it manages Spring Bean life cycle and project specific configurations
- **MVC framework:** that is used to create web applications or RESTful web services, capable of returning XML/JSON

responses
- **Transaction management:** reduces the amount of boiler-plate code in JDBC operations, file uploading, etc., either by using Java annotations or by Spring Bean XML configuration file
- **Exception Handling:** Spring provides a convenient API for translating technology-specific exceptions into unchecked exceptions

What Spring sub-projects do you know? Describe them briefly.

- **Core** – a key module that provides fundamental parts of the framework, like IoC or DI
- **JDBC** – this module enables a JDBC-abstraction layer that removes the need to do JDBC coding for specific vendor databases
- **ORM integration** – provides integration layers for popular object-relational mapping APIs, such as JPA, JDO, and Hibernate
- **Web** – a web-oriented integration module, providing multipart file upload, Servlet listeners, and web-oriented application context functionalities
- **MVC framework** – a web module implementing the Model View Controller design pattern
- **AOP module** – aspect-oriented programming implementation allowing the definition of clean method-interceptors and pointcuts

What is Dependency Injection?

Dependency Injection, an aspect of Inversion of Control (IoC), is a general concept stating that you do not create your objects manually but instead describe how they should

be created. An IoC container will instantiate required classes if needed.
For more details, please refer here.

How can we inject beans in Spring?

A few different options exist:

- Setter Injection
- Constructor Injection
- Field Injection

The configuration can be done using XML files or annotations.
For more details, check this article.

Which is the best way of injecting beans and why?

The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. Constructor injection allows injecting values to immutable fields and makes testing easier.

What is the difference between *BeanFactory* and *ApplicationContext*?

BeanFactory is an interface representing a container that provides and manages bean instances. The default implementation instantiates beans lazily when *getBean()* is called.
ApplicationContext is an interface representing a container holding all information, metadata, and beans in the application. It also extends the *BeanFactory* interface but the default implementation instantiates beans eagerly when the application starts. This behavior can be overridden for

individual beans.
For all differences, please refer to the reference.

What is a Spring Bean?

The Spring Beans are Java Objects that are initialized by the Spring IoC container.

What is the default bean scope in Spring framework?

By default, a Spring Bean is initialized as a *singleton*.

How to define the scope of a bean?

To set Spring Bean's scope, we can use *@Scope* annotation or "scope" attribute in XML configuration files. There are five supported scopes:

- **singleton**
- **prototype**
- **request**
- **session**
- **global-session**

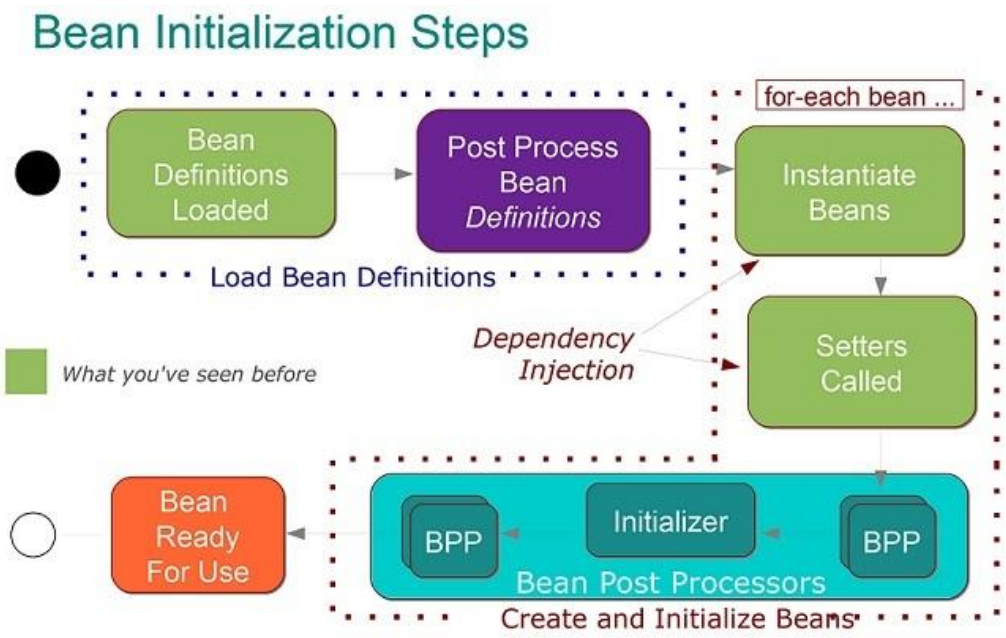
For differences, please refer here.

Are singleton beans thread-safe?

No, singleton beans are not thread-safe, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.

What does the Spring bean lifecycle look like?

First, a Spring bean needs to be instantiated, based on Java or XML bean definition. It may also be required to perform some initialization to get it into a usable state. After that, when the bean is no longer required, it will be removed from the IoC container.
The whole cycle with all initialization methods is shown on the image:



What is the Spring Java-Based Configuration?

It’s one of the ways of configuring Spring-based applications in a type-safe manner. It’s an alternative to the XML-based configuration.
Also, if you want to migrate your project from XML to Java config, please refer to this article.

Can we have multiple Spring configuration files in one project?

Yes, in large projects, having multiple Spring configurations is recommended to increase maintainability and modularity.
You can load multiple Java-based configuration files:

```
1  @Configuration
2  @Import({MainConfig.class, SchedulerConfig.class})
3  public class AppConfig {
```

Or load one XML file that will contain all other configs:

```
1  ApplicationContext context = new ClassPathXmlApplicationCo
```

And inside this XML file you'll have:

```
1  <import resource="main.xml"/>
2  <import resource="scheduler.xml"/>
```

What is Spring Security?

Spring Security is a separate module of the Spring framework that focuses on providing authentication and authorization methods in Java applications. It also takes care of most of the common security vulnerabilities such as CSRF attacks.
To use Spring Security in web applications, you can get started with a simple annotation: *@EnableWebSecurity*.
You can find the whole series of articles related to security on Baeldung.

What is Spring Boot?

Spring Boot is a project that provides a pre-configured set of frameworks to reduce boilerplate configuration so that you can have a Spring application up and running with the smallest amount of code.

Q17. Name some of the Design Patterns used in the Spring

Framework?

- **Singleton Pattern:** Singleton-scoped beans
- **Factory Pattern:** Bean Factory classes
- **Prototype Pattern:** Prototype-scoped beans
- **Adapter Pattern:** Spring Web and Spring MVC
- **Proxy Pattern:** Spring Aspect Oriented Programming support
- **Template Method Pattern:** *JdbcTemplate*, *HibernateTemplate,* etc.
- **Front Controller:** Spring MVC *DispatcherServlet*
- **Data Access Object:** Spring DAO support
- **Model View Controller:** Spring MVC

QHow does the scope *Prototype* work?

Scope *prototype* means that every time you call for an instance of the Bean, Spring will create a new instance and return it. This differs from the default *singleton* scope, where a single object instance is instantiated once per Spring IoC container.

How to Get *ServletContext* and *ServletConfig* Objects in a Spring Bean?

You can do either by:

1. Implementing Spring-aware interfaces. The complete list is available here.
2. Using *@Autowired* annotation on those beans:

```
1  @Autowired
2  ServletContext servletContext;
3
4  @Autowired
5  ServletConfig servletConfig;
```

What is a Controller in Spring MVC?

Simply put, all the requests processed by the *DispatcherServlet* are directed to classes annotated with *@Controller*. Each controller class maps one or more requests to methods that process and execute the requests with provided inputs.
If you need to take a step back, we recommend having a look at the concept of the Front Controller in the typical Spring MVC architecture.

How does the *@RequestMapping* annotation work?

The *@RequestMapping* annotation is used to map web requests to Spring Controller methods. In addition to simple use cases, we can use it for mapping of HTTP headers, binding parts of the URI with *@PathVariable,* and working with URI parameters and the *@RequestParam* annotation.
More details on *@RequestMapping* are available here.
For more Spring MVC questions, please check out Spring MVC Interview Questions article.

What is Spring *JDBCTemplate* class and how to use it?

The Spring JDBC template is the primary API through which we can access database operations logic that we're interested in:

- creation and closing of connections
- executing statements and stored procedure calls
- iterating over the *ResultSet* and returning results

To use it, we'll need to define the simple configuration of *DataSource*:

```
1   @Configuration
2   @ComponentScan("org.baeldung.jdbc")
3   public class SpringJdbcConfig {
4       @Bean
5       public DataSource mysqlDataSource() {
6           DriverManagerDataSource dataSource = new DriverMar
7           dataSource.setDriverClassName("com.mysql.jdbc.Driv
8           dataSource.setUrl("jdbc:mysql://localhost:3306/spr
9           dataSource.setUsername("guest_user");
10          dataSource.setPassword("guest_password");
11
12          return dataSource;
13      }
14  }
```

How would you enable t*ransactions* in Spring and what are their benefits?

There are two distinct ways to configure *Transactions* – with annotations or by using Aspect Oriented Programming (AOP) – each with their advantages.
The benefits of using Spring Transactions, according to the official docs, are:

- Provide a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO
- Support declarative transaction management
- Provide a simpler API for programmatic transaction management than some complex transaction APIs such as JTA
- Integrate very well with Spring's various data access

abstractions

What is Spring DAO?

Spring Data Access Object is Spring's support provided to work with data access technologies like JDBC, Hibernate, and JPA in a consistent and easy way.
You can, of course, go more in-depth on persistence, with the entire series discussing persistence in Spring.

What is Aspect-Oriented Programming?

Aspects enable the modularization of cross-cutting concerns such as transaction management that span multiple types and objects by adding extra behavior to already existing code without modifying affected classes.
Here is the example of aspect-based execution time logging.

What are *Aspect*, *Advice*, *Pointcut,* and *JoinPoint* in AOP?

- ***Aspect***: a class that implements cross-cutting concerns, such as transaction management
- ***Advice***: the methods that get executed when a specific *JoinPoint* with matching *Pointcut* is reached in the application
- ***Pointcut***: a set of regular expressions that are matched with *JoinPoint* to determine whether *Advice*needs to be executed or not
- ***JoinPoint***: a point during the execution of a program, such as the execution of a method or the handling of an exception

What is *Weaving*?

According to the official docs, *weaving* is a process that links aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs *weaving* at runtime.

What is reactive programming?

Reactive programming is about non-blocking, event-driven applications that scale with a small number of threads, with back pressure being a key ingredient that aims to ensure producers don't overwhelm consumers.
The primary benefits of reactive programming are:

- increased utilization of computing resources on multicore and multi-CPU hardware
- and increased performance by reducing serialization

Reactive programming is generally event-driven, in contrast to reactive systems, which are message-driven. Thus, using reactive programming does not mean we're building a reactive system, which is an architectural style.
However, reactive programming may be used as a means to implement reactive systems if we follow the Reactive Manifesto, which is quite vital to understand.
Based on this, reactive systems have four important characteristics:

- **Responsive**: the system should respond in a timely manner
- **Resilient**: in case the system faces any failure, it should stay responsive
- **Elastic**: reactive systems can react to changes and stay responsive under varying workload
- **Message-driven**: reactive systems need to establish a boundary between components by relying on asynchronous

message passing

What is *Spring WebFlux*?

Spring WebFlux is Spring's reactive-stack web framework, and it's an alternative to Spring MVC.
In order to achieve this reactive model and be highly scalable, the entire stack is non-blocking. Check out our tutorial on Spring 5 WebFlux for additional details.

What are the *Mono and Flux types*?

The WebFlux framework in Spring Framework 5 uses Reactor as its async foundation.
This project provides two core types: *Mono* to represent a single async value, and *Flux* to represent a stream of async values. They both implement the *Publisher* interface defined in the Reactive Streams specification.

Mono implements *Publisher* and returns 0 or 1 elements:

```
1 public abstract class Mono<T> implements Publisher<T> {...}
```

Also, *Flux* implements *Publisher* and returns *N* elements:

```
1 public abstract class Flux<T> implements Publisher<T> {...}
```

By definition, the two types represent streams, hence they're both lazy, which means nothing is executed until we consume the stream using the *subscribe()* method. Both types are immutable, therefore calling any method will return a new instance of *Flux* or *Mono*.

What is the *use of WebClient and WebTestClient*?

WebClient is a component in the new Web Reactive framework

that can act as a reactive client for performing non-blocking HTTP requests. Being a reactive client, it can handle reactive streams with back pressure, and it can take full advantage of Java 8 lambdas. It can also handle both sync and async scenarios.
On the other hand, the *WebTestClient* is a similar class that we can use in tests. Basically, it's a thin shell around the *WebClient*. It can connect to any server over an HTTP connection. It can also bind directly to WebFlux applications using mock request and response objects, without the need for an HTTP server.

What are the disadvantages of using *Reactive Streams*?

The major disadvantages of using reactive streams are:

- Troubleshooting a Reactive application is a bit difficult; be sure to check out our **tutorial on debugging reactive streams** for some handy debugging tips
- There is limited support for reactive data stores, as traditional relational data stores have yet to embrace the reactive paradigm
- There's an extra learning curve when implementing

Is Spring 5 compatible with older versions of Java?

In order to take advantage of Java 8 features, the Spring codebase has been revamped. This means older versions of Java cannot be used. Hence, the framework requires a minimum of Java 8.

How does Spring 5 integrate with JDK 9 modularity?

In Spring 5, everything has been modularized, thus we won't

be forced to import jars that may not have the functionalities we're looking for.

Please have a look at our guide to Java 9 modularity for an in-depth understanding of how this technology works.

Let's see an example to understand the new module functionality in Java 9 and how to organize a Spring 5 project based on this concept.

To start, let's create a new class that contains a single method to return a *String* "HelloWorld". We'll place this within a new Java project – *HelloWorldModule*:

```
package com.hello;
public class HelloWorld {
    public String sayHello(){
        return "HelloWorld";
    }
}
```

Then let's create a new module:

```
module com.hello {
    export com.hello;
}
```

Now, let's create a new Java Project, *HelloWorldClient*, to consume the above module by defining a module:

```
module com.hello.client {
    requires com.hello;
}
```

The above module will be available for testing now:

```
public class HelloWorldClient {
    public static void main(String[] args){
        HelloWorld helloWorld = new HelloWorld();
        log.info(helloWorld.sayHello());
    }
}
```

Can we use both Web MVC and WebFlux in the same application?

As of now, Spring Boot will only allow either Spring MVC or Spring WebFlux, as Spring Boot tries to auto-configure the context depending on the dependencies that exist in its classpath.

Also, Spring MVC cannot run on Netty. Moreover, MVC is a blocking paradigm and WebFlux is a non-blocking style, therefore we shouldn’t be mixing both together, as they serve different purposes.

Spring Core Annotations

1. Overview

We can leverage the capabilities of Spring DI engine using the annotations in the *org.springframework.beans.factory.annotation* and *org.springframework.context.annotation* packages.

We often call these “Spring core annotations” and we’ll review them in this tutorial.

2. DI-Related Annotations

2.1. *@Autowired*

We can use the *@Autowired* to **mark a dependency which Spring is going to resolve and inject.** We can use this annotation with a constructor, setter, or field injection.

Constructor injection:

```
class Car {
    Engine engine;

    @Autowired
    Car(Engine engine) {
        this.engine = engine;
    }
}
```

Setter injection:

```
class Car {
    Engine engine;

    @Autowired
    void setEngine(Engine engine) {
        this.engine = engine;
```

```
7       }
8   }
```

Field injection:

```
1   class Car {
2       @Autowired
3       Engine engine;
4   }
```

@Autowired has a *boolean* argument called *required* with a default value of *true*. It tunes Spring's behavior when it doesn't find a suitable bean to wire. When *true*, an exception is thrown, otherwise, nothing is wired.

Note, that if we use constructor injection, all constructor arguments are mandatory.

Starting with version 4.3, we don't need to annotate constructors with *@Autowired* explicitly unless we declare at least two constructors.

For more details visit our articles about *@Autowired* and constructor injection.

2.2. *@Bean*

@Bean marks a factory method which instantiates a Spring bean:

```
1   @Bean
2   Engine engine() {
3       return new Engine();
4   }
```

Spring calls these methods when a new instance of the return type is required.

The resulting bean has the same name as the factory method. If we want to name it differently, we can do so with the *name* or the *value* arguments of this annotation (the argument *value* is an alias for the argument *name*):

```
1  @Bean("engine")
2  Engine getEngine() {
3      return new Engine();
4  }
```

Note, that all methods annotated with *@Bean* must be in *@Configuration* classes.

2.3. *@Qualifier*

We use *@Qualifier* along with *@Autowired* to **provide the bean id or bean name** we want to use in ambiguous situations.

For example, the following two beans implement the same interface:

```
1  class Bike implements Vehicle {}
2
3  class Car implements Vehicle {}
```

If Spring needs to inject a *Vehicle* bean, it ends up with multiple matching definitions. In such cases, we can provide a bean's name explicitly using the *@Qualifier* annotation.

Using constructor injection:

```
1  @Autowired
2  Biker(@Qualifier("bike") Vehicle vehicle) {
3      this.vehicle = vehicle;
4  }
```

Using setter injection:

```
1  @Autowired
2  void setVehicle(@Qualifier("bike") Vehicle vehicle) {
3      this.vehicle = vehicle;
4  }
```

Alternatively:

```
1  @Autowired
2  @Qualifier("bike")
```

```
3  void setVehicle(Vehicle vehicle) {
4      this.vehicle = vehicle;
5  }
```

Using field injection:

```
1  @Autowired
2  @Qualifier("bike")
3  Vehicle vehicle;
```

For a more detailed description, please read this article.

2.4. *@Required*

@Required on setter methods to mark dependencies that we want to populate through XML:

```
1  @Required
2  void setColor(String color) {
3      this.color = color;
4  }
```

```
1  <bean class="com.baeldung.annotations.Bike">
2      <property name="color" value="green" />
3  </bean>
```

Otherwise, *BeanInitializationException* will be thrown.

2.5. *@Value*

We can use *@Value* for injecting property values into beans. It’s compatible with constructor, setter, and field injection.

Constructor injection:

```
1  Engine(@Value("8") int cylinderCount) {
2      this.cylinderCount = cylinderCount;
3  }
```

Setter injection:

```
1  @Autowired
2  void setCylinderCount(@Value("8") int cylinderCount) {
3      this.cylinderCount = cylinderCount;
4  }
```

Alternatively:

```
1  @Value("8")
2  void setCylinderCount(int cylinderCount) {
3      this.cylinderCount = cylinderCount;
4  }
```

Field injection:

```
1  @Value("8")
2  int cylinderCount;
```

Of course, injecting static values isn't useful. Therefore, we can use **placeholder strings** in *@Value* to wire values **defined in external sources**, for example, in *.properties* or *.yaml* files.

Let's assume the following *.properties* file:

```
1  engine.fuelType=petrol
```

We can inject the value of *engine.fuelType* with the following:

```
1  @Value("${engine.fuelType}")
2  String fuelType;
```

We can use *@Value* even with SpEL. More advanced examples can be found in our article about *@Value*.

2.6. *@DependsOn*

We can use this annotation to make Spring **initialize other beans before the annotated one**. Usually, this behavior is automatic, based on the explicit dependencies between beans.

We only need this annotation **when the dependencies are implicit**, for example, JDBC driver loading or static variable initialization.

We can use *@DependsOn* on the dependent class specifying the names of the dependency beans. The annotation's *value* argument needs an array containing the dependency bean names:

```
1  @DependsOn("engine")
2  class Car implements Vehicle {}
```

Alternatively, if we define a bean with the *@Bean* annotation, the factory method should be annotated with *@DependsOn*:

```
1  @Bean
2  @DependsOn("fuel")
3  Engine engine() {
4      return new Engine();
5  }
```

2.7. *@Lazy*

We use *@Lazy* when we want to initialize our bean lazily. By default, Spring creates all singleton beans eagerly at the startup/bootstrapping of the application context.

However, there are cases when **we need to create a bean when we request it, not at application startup.**

This annotation behaves differently depending on where we exactly place it. We can put it on:

- a *@Bean* annotated bean factory method, to delay the method call (hence the bean creation)
- a *@Configuration* class and all contained *@Bean* methods will be affected
- a *@Component* class, which is not a *@Configuration* class, this bean will be initialized lazily
- an *@Autowired* constructor, setter, or field, to load the dependency itself lazily (via proxy)

This annotation has an argument named *value* with the default

value of *true*. It is useful to override the default behavior.

For example, marking beans to be eagerly loaded when the global setting is lazy, or configure specific *@Bean* methods to eager loading in a *@Configuration* class marked with *@Lazy*:

```
1   @Configuration
2   @Lazy
3   class VehicleFactoryConfig {
4
5       @Bean
6       @Lazy(false)
7       Engine engine() {
8           return new Engine();
9       }
10  }
```

For further reading, please visit this article.

2.8. *@Lookup*

A method annotated with *@Lookup* tells Spring to return an instance of the method's return type when we invoke it.

Detailed information about the annotation can be found in this article.

2.9. *@Primary*

Sometimes we need to define multiple beans of the same type. In these cases, the injection will be unsuccessful because Spring has no clue which bean we need.

We already saw an option to deal with this scenario: marking all the wiring points with *@Qualifier* and specify the name of the required bean.

However, most of the time we need a specific bean and rarely the others. We can use *@Primary* to simplify this case: if **we mark the most frequently used bean with *@Primary*** it will be chosen on unqualified injection points:

```
1   @Component
2   @Primary
3   class Car implements Vehicle {}
4
5   @Component
6   class Bike implements Vehicle {}
7
8   @Component
9   class Driver {
10      @Autowired
11      Vehicle vehicle;
12  }
13
14  @Component
15  class Biker {
16      @Autowired
17      @Qualifier("bike")
18      Vehicle vehicle;
19  }
```

In the previous example *Car* is the primary vehicle. Therefore, in the *Driver* class, Spring injects a *Car* bean. Of course, in the *Biker* bean, the value of the field *vehicle* will be a *Bike* object because it’s qualified.

2.10. *@Scope*

We use *@Scope* to define the scope of a *@Component* class or a *@Bean* definition. It can be either *singleton, prototype, request, session, globalSession* or some custom scope.

For example:

```
1  @Component
2  @Scope("prototype")
3  class Engine {}
```

3. Context Configuration Annotations

We can configure the application context with the annotations described in this section.

3.1. *@Profile*

If we want Spring to **use a *@Component* class or a *@Bean* method only when a specific profile is active**, we can mark it with *@Profile*. We can configure the name of the profile with the *value* argument of the annotation:

```
1  @Component
2  @Profile("sportDay")
3  class Bike implements Vehicle {}
```

You can read more about profiles in this article.

3.2. *@Import*

We can use **specific *@Configuration* classes without component scanning** with this annotation. We can provide those classes with *@Import*‘s *value* argument:

```
1  @Import(VehiclePartSupplier.class)
2  class VehicleFactoryConfig {}
```

3.3. *@ImportResource*

We can **import XML configurations** with this annotation. We can specify the XML file locations with the *locations* argument, or with its alias, the *value* argument:

```
1  @Configuration
2  @ImportResource("classpath:/annotations.xml")
3  class VehicleFactoryConfig {}
```

3.4. *@PropertySource*

With this annotation, we can **define property files for application settings**:

```
1  @Configuration
2  @PropertySource("classpath:/annotations.properties")
3  class VehicleFactoryConfig {}
```

@PropertySource leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

```
1  @Configuration
2  @PropertySource("classpath:/annotations.properties")
3  @PropertySource("classpath:/vehicle-factory.properties")
4  class VehicleFactoryConfig {}
```

3.5. *@PropertySources*

We can use this annotation to specify multiple *@PropertySource* configurations:

```
1  @Configuration
2  @PropertySources({
3      @PropertySource("classpath:/annotations.properties"),
4      @PropertySource("classpath:/vehicle-factory.properties
5  })
6  class VehicleFactoryConfig {}
```

Note, that since Java 8 we can achieve the same with the repeating annotations feature as described above.

4. Conclusion

In this article, we saw an overview of the most common Spring core annotations. We saw how to configure bean wiring and application context, and how to mark classes for component scanning.

As usual, the examples are available over on GitHub.

Spring Web Annotations

1. Overview

In this tutorial, we'll explore Spring Web annotations from the *org.springframework.web.bind.annotation*package.

2. *@RequestMapping*

Simply put, *@RequestMapping* **marks request handler methods** inside *@Controller* classes; it can be configured using:

- *path,* or its aliases, *name,* and *value:* which URL the method is mapped to
- *method:* compatible HTTP methods
- *params:* filters requests based on presence, absence, or value of HTTP parameters
- *headers:* filters requests based on presence, absence, or value of HTTP headers
- *consumes:* which media types the method can consume in the HTTP request body
- *produces:* which media types the method can produce in the HTTP response body

Here's a quick example of what that looks like:

```
1  @Controller
```

```
2  class VehicleController {
3
4      @RequestMapping(value = "/vehicles/home", method = RequestMetho
5      String home() {
6          return "home";
7      }
8  }
```

We can provide **default settings for all handler methods in a *@Controller* class** if we apply this annotation on the class level. The only **exception is the URL which Spring won't override** with method level settings but appends the two path parts.

For example, the following configuration has the same effect as the one above:

```
1  @Controller
2  @RequestMapping(value = "/vehicles", method = RequestMetho
3  class VehicleController {
4
5      @RequestMapping("/home")
6      String home() {
7          return "home";
8      }
9  }
```

Moreover, *@GetMapping*, *@PostMapping*, *@PutMapping*, *@DeleteMapping*, and *@PatchMapping* are different variants of *@RequestMapping* with the HTTP method already set to GET, POST, PUT, DELETE, and PATCH respectively.

These are available since Spring 4.3 release.

3. *@RequestBody*

Let's move on to *@RequestBody* – which maps the **body of the HTTP request to an object**:

```
1  @PostMapping("/save")
2  void saveVehicle(@RequestBody Vehicle vehicle) {
3      // ...
4  }
```

The deserialization is automatic and depends on the content type of the request.

4. *@PathVariable*

Next, let’s talk about *@PathVariable*.

This annotation indicates that a **method argument is bound to a URI template variable.** We can specify the URI template with the *@RequestMapping* annotation and bind a method argument to one of the template parts with *@PathVariable*.

We can achieve this with the *name* or its alias, the *value* argument:

```
1  @RequestMapping("/{id}")
2  Vehicle getVehicle(@PathVariable("id") long id) {
3      // ...
4  }
```

If the name of the part in the template matches the name of the method argument, we don’t have to specify it in the annotation:

```
1  @RequestMapping("/{id}")
2  Vehicle getVehicle(@PathVariable long id) {
3      // ...
4  }
```

Moreover, we can mark a path variable optional by setting the argument *required* to false:

```
1  @RequestMapping("/{id}")
2  Vehicle getVehicle(@PathVariable(required = false) long id
3      // ...
4  }
```

5. *@RequestParam*

We use *@RequestParam* for **accessing HTTP request parameters**:

```
1  @RequestMapping
2  Vehicle getVehicleByParam(@RequestParam("id") long id) {
3      // ...
4  }
```

It has the same configuration options as the *@PathVariable* annotation.

In addition to those settings, with *@RequestParam* we can specify an injected value when Spring finds no or empty value in the request. To achieve this, we have to set the *defaultValue* argument.

Providing a default value implicitly sets *required* to *false:*

```
1  @RequestMapping("/buy")
2  Car buyCar(@RequestParam(defaultValue = "5") int seatCount
3      // ...
4  }
```

Besides parameters, there are **other HTTP request parts we can access: cookies and headers**. We can access them with the annotations ***@CookieValue*** **and** ***@RequestHeader*** respectively.

We can configure them the same way as *@RequestParam*.

6. Response Handling Annotations

In the next sections, we will see the most common annotations to manipulate HTTP responses in Spring MVC.

6.1. *@ResponseBody*

If we mark a request handler method with *@ResponseBody*, **Spring treats the result of the method**

as the response itself:

```
1  @ResponseBody
2  @RequestMapping("/hello")
3  String hello() {
4      return "Hello World!";
5  }
```

If we annotate a *@Controller* class with this annotation, all request handler methods will use it.

6.2. *@ExceptionHandler*

With this annotation, we can declare a **custom error handler method**. Spring calls this method when a request handler method throws any of the specified exceptions.

The caught exception can be passed to the method as an argument:

```
1  @ExceptionHandler(IllegalArgumentException.class)
2  void onIllegalArgumentException(IllegalArgumentException e
3      // ...
4  }
```

6.3. *@ResponseStatus*

We can specify the **desired HTTP status of the response** if we annotate a request handler method with this annotation. We can declare the status code with the *code* argument, or its alias, the *value* argument.

Also, we can provide a reason using the *reason* argument.

We also can use it along with *@ExceptionHandler*:

```
1  @ExceptionHandler(IllegalArgumentException.class)
2  @ResponseStatus(HttpStatus.BAD_REQUEST)
3  void onIllegalArgumentException(IllegalArgumentException e
```

```
4       // ...
5   }
```

For more information about HTTP response status, please visit this article.

7. Other Web Annotations

Some annotations don’t manage HTTP requests or responses directly. In the next sections, we’ll introduce the most common ones.

7.1. *@Controller*

We can define a Spring MVC controller with *@Controller*. For more information, please visit our article about Spring Bean Annotations.

7.2. *@RestController*

The *@RestController* **combines *@Controller* and *@ResponseBody*.**

Therefore, the following declarations are equivalent:

```
1   @Controller
2   @ResponseBody
3   class VehicleRestController {
4       // ...
5   }
```

```
1   @RestController
2   class VehicleRestController {
3       // ...
4   }
```

7.3. *@ModelAttribute*

With this annotation we can **access elements that are already in the model** of an MVC *@Controller,* by providing the model key:

```
1  @PostMapping("/assemble")
2  void assembleVehicle(@ModelAttribute("vehicle") Vehicle ve
3      // ...
4  }
```

Like with *@PathVariable* and *@RequestParam*, we don't have to specify the model key if the argument has the same name:

```
1  @PostMapping("/assemble")
2  void assembleVehicle(@ModelAttribute Vehicle vehicle) {
3      // ...
4  }
```

Besides, *@ModelAttribute* has another use: if we annotate a method with it, Spring will **automatically add the method's return value to the model**:

```
1  @ModelAttribute("vehicle")
2  Vehicle getVehicle() {
3      // ...
4  }
```

Like before, we don't have to specify the model key, Spring uses the method's name by default:

```
1  @ModelAttribute
2  Vehicle vehicle() {
3      // ...
4  }
```

Before Spring calls a request handler method, it invokes all *@ModelAttribute* annotated methods in the class.

More information about *@ModelAttribute* can be found in this article.

7.4. *@CrossOrigin*

@CrossOrigin **enables cross-domain communication** for the annotated request handler methods:

```
1  @CrossOrigin
2  @RequestMapping("/hello")
3  String hello() {
4      return "Hello World!";
5  }
```

If we mark a class with it, it applies to all request handler methods in it.

We can fine-tune CORS behavior with this annotation's arguments.

For more details, please visit this article.

8. Conclusion

In this article, we saw how we can handle HTTP requests and responses with Spring MVC.

As usual, the examples are available over on GitHub.

Spring Boot Annotations

1. Overview

Spring Boot made configuring Spring easier with its auto-configuration feature.

In this quick tutorial, we'll explore the annotations from the *org.springframework.boot.autoconfigure* and *org.springframework.boot.autoconfigure.condition* packages.

2. *@SpringBootApplication*

We use this annotation to **mark the main class of a Spring Boot application**:

```
1  @SpringBootApplication
2  class VehicleFactoryApplication {
3
4      public static void main(String[] args) {
5          SpringApplication.run(VehicleFactoryApplication.cl
6      }
7  }
```

@SpringBootApplication encapsulates ***@Configuration***, ***@EnableAutoConfiguration***, **and** ***@ComponentScan*** annotations with their default attributes.

3. *@EnableAutoConfiguration*

@EnableAutoConfiguration, as its name says, enables auto-configuration. It means that **Spring Boot looks for auto-configuration beans** on its classpath and automatically applies them.

Note, that we have to use this annotation with *@Configuration*:

```
1  @Configuration
2  @EnableAutoConfiguration
3  class VehicleFactoryConfig {}
```

4. Auto-Configuration Conditions

Usually, when we write our **custom auto-configurations**, we want Spring to **use them conditionally**. We can achieve this with the annotations in this section.

We can place the annotations in this section

on *@Configuration* classes or *@Bean* methods.

In the next sections, we'll only introduce the basic concept behind each condition. For further information, please visit this article.

4.1. *@ConditionalOnClass* and *@ConditionalOnMissingClass*

Using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation's **argument is present/absent**:

```
1  @Configuration
2  @ConditionalOnClass(DataSource.class)
3  class MySQLAutoconfiguration {
4      //...
5  }
```

4.2. *@ConditionalOnBean* and *@ConditionalOnMissingBean*

We can use these annotations when we want to define conditions based on the **presence or absence of a specific bean**:

```
1  @Bean
2  @ConditionalOnBean(name = "dataSource")
3  LocalContainerEntityManagerFactoryBean entityManagerFactor
4      // ...
5  }
```

4.3. *@ConditionalOnProperty*

With this annotation, we can make conditions on the **values of properties**:

```
1  @Bean
2  @ConditionalOnProperty(
```

```
3       name = "usemysql",
4       havingValue = "local"
5   )
6   DataSource dataSource() {
7       // ...
8   }
```

4.4. *@ConditionalOnResource*

We can make Spring to use a definition only when a specific **resource is present**:

```
1   @ConditionalOnResource(resources = "classpath:mysql.proper
2   Properties additionalProperties() {
3       // ...
4   }
```

4.5. *@ConditionalOnWebApplication* and *@ConditionalOnNotWebApplication*

With these annotations, we can create conditions based on if the current **application is or isn’t a web application**:

```
1   @ConditionalOnWebApplication
2   HealthCheckController healthCheckController() {
3       // ...
4   }
```

4.6. *@ConditionalExpression*

We can use this annotation in more complex situations. Spring will use the marked definition when the **SpEL expression is evaluated to true**:

```
1   @Bean
2   @ConditionalOnExpression("${usemysql} && ${mysqlserver ==
```

```
3  DataSource dataSource() {
4      // ...
5  }
```

4.7. *@Conditional*

For even more complex conditions, we can create a class evaluating the **custom condition**. We tell Spring to use this custom condition with *@Conditional*:

```
1  @Conditional(HibernateCondition.class)
2  Properties additionalProperties() {
3      //...
4  }
```

5. Conclusion

In this article, we saw an overview of how can we fine-tune the auto-configuration process and provide conditions for custom auto-configuration beans.

As usual, the examples are available over on GitHub.

Spring Scheduling Annotations

1. Overview

When single-threaded execution isn’t enough, we can use annotations from the *org.springframework.scheduling.annotation* package.

In this quick tutorial, we're going to explore the Spring Scheduling Annotations.

2. *@EnableAsync*

With this annotation, we can enable asynchronous functionality in Spring.

We must use it with *@Configuration*:

```
1  @Configuration
2  @EnableAsync
3  class VehicleFactoryConfig {}
```

Now, that we enabled asynchronous calls, we can use *@Async* to define the methods supporting it.

3. *@EnableScheduling*

With this annotation, we can enable scheduling in the application.

We also have to use it in conjunction with *@Configuration*:

```
1  @Configuration
2  @EnableScheduling
3  class VehicleFactoryConfig {}
```

As a result, we can now run methods periodically with *@Scheduled*.

4. *@Async*

We can define methods we want to **execute on a different thread**, hence run them asynchronously.

To achieve this, we can annotate the method with *@Async*:

```
1  @Async
2  void repairCar() {
```

```
3       // ...
4   }
```

If we apply this annotation to a class, then all methods will be called asynchronously.

Note, that we need to enable the asynchronous calls for this annotation to work, with *@EnableAsync* or XML configuration.

More information about *@Async* can be found in this article.

5. *@Scheduled*

If we need a method to **execute periodically**, we can use this annotation:

```
1   @Scheduled(fixedRate = 10000)
2   void checkVehicle() {
3       // ...
4   }
```

We can use it to execute a method at **fixed intervals**, or we can fine-tune it with **cron-like expressions**.

@Scheduled leverages the Java 8 repeating annotations feature, which means we can mark a method with it multiple times:

```
1   @Scheduled(fixedRate = 10000)
2   @Scheduled(cron = "0 * * * * MON-FRI")
3   void checkVehicle() {
4       // ...
5   }
```

Note, that the method annotated with *@Scheduled* should have a *void* return type.

Moreover, we have to enable scheduling for this annotation to work for example with *@EnableScheduling*or XML configuration.

For more information about scheduling read this article.

6. *@Schedules*

We can use this annotation to specify multiple *@Scheduled* rules:

```
@Schedules({
  @Scheduled(fixedRate = 10000),
  @Scheduled(cron = "0 * * * * MON-FRI")
})
void checkVehicle() {
    // ...
}
```

Note, that since Java 8 we can achieve the same with the repeating annotations feature as described above.

7. Conclusion

In this article, we saw an overview of the most common Spring scheduling annotations.

As usual, the examples are available over on GitHub.

Spring Data Annotations

1. Introduction

Spring Data provides an abstraction over data storage technologies. Therefore, our business logic code can be much more independent of the underlying persistence implementation. Also, Spring simplifies the handling of implementation-dependent details of data storage.

In this tutorial, we'll see the most common annotations of the Spring Data, Spring Data JPA, and Spring Data MongoDB projects.

2. Common Spring Data Annotations

2.1. *@Transactional*

When we want to **configure the transactional behavior of a method**, we can do it with:

```
1  @Transactional
2  void pay() {}
```

If we apply this annotation on class level, then it works on all methods inside the class. However, we can override its effects by applying it to a specific method.

It has many configuration options, which can be found in this article.

2.2. *@NoRepositoryBean*

Sometimes we want to create repository interfaces with the only goal of providing common methods for the child repositories.

Of course, we don't want Spring to create a bean of these repositories since we won't inject them anywhere. *@NoRepositoryBean* does exactly this: when we mark

a child interface of *org.springframework.data.repository.Repository*, Spring won't create a bean out of it.

For example, if we want an *Optional<T> findById(ID id)* method in all of our repositories, we can create a base repository:

```
@NoRepositoryBean
interface MyUtilityRepository<T, ID extends Serializable>
ID> {
    Optional<T> findById(ID id);
}
```

This annotation doesn't affect the child interfaces; hence Spring will create a bean for the following repository interface:

```
@Repository
interface PersonRepository extends MyUtilityRepository<Per
```

Note, that the example above isn't necessary since Spring Data version 2 which includes this method replacing the older *T findOne(ID id)*.

2.3. *@Param*

We can pass named parameters to our queries using *@Param*:

```
@Query("FROM Person p WHERE p.name = :name")
Person findByName(@Param("name") String name);
```

Note, that we refer to the parameter with the *:name* syntax.

For further examples, please visit this article.

2.4. *@Id*

@Id marks a field in a model class as the primary key:

```
class Person {

```

```
3       @Id
4       Long id;
5
6       // ...
7
8   }
```

Since it’s implementation-independent, it makes a model class easy to use with multiple data store engines.

2.5. *@Transient*

We can use this annotation to mark a field in a model class as transient. Hence the data store engine won’t read or write this field’s value:

```
1   class Person {
2
3       // ...
4
5       @Transient
6       int age;
7
8       // ...
9
10  }
```

Like *@Id*, *@Transient* is also implementation-independent, which makes it convenient to use with multiple data store implementations.

2.6. *@CreatedBy*, *@LastModifiedBy*, *@CreatedDate*, *@LastModifiedDate*

With these annotations, we can audit our model classes: Spring automatically populates the annotated fields with the principal who created the object, last modified it, and the

date of creation, and last modification:

```
1   public class Person {
2
3       // ...
4
5       @CreatedBy
6       User creator;
7
8       @LastModifiedBy
9       User modifier;
10
11      @CreatedDate
12      Date createdAt;
13
14      @LastModifiedDate
15      Date modifiedAt;
16
17      // ...
18
19  }
```

Note, that if we want Spring to populate the principals, we need to use Spring Security as well.

For a more thorough description, please visit this article.

3. Spring Data JPA Annotations

3.1. *@Query*

With *@Query*, we can provide a JPQL implementation for a repository method:

```
1  @Query("SELECT COUNT(*) FROM Person p")
2  long getPersonCount();
```

Also, we can use named parameters:

```
1  @Query("FROM Person p WHERE p.name = :name")
2  Person findByName(@Param("name") String name);
```

Besides, we can use native SQL queries, if we set the *nativeQuery* argument to *true*:

```
1  @Query(value = "SELECT AVG(p.age) FROM person p", nativeQu
2  int getAverageAge();
```

For more information, please visit this article.

3.2. *@Procedure*

With Spring Data JPA we can easily call stored procedures from repositories.

First, we need to declare the repository on the entity class using standard JPA annotations:

```
1   @NamedStoredProcedureQueries({
2       @NamedStoredProcedureQuery(
3           name = "count_by_name",
4           procedureName = "person.count_by_name",
5           parameters = {
6               @StoredProcedureParameter(
7                   mode = ParameterMode.IN,
8                   name = "name",
9                   type = String.class),
10              @StoredProcedureParameter(
11                  mode = ParameterMode.OUT,
12                  name = "count",
13                  type = Long.class)
14              }
15      )
16  })
17
18  class Person {}
```

After this, we can refer to it in the repository with the

name we declared in the *name* argument:

```
1  @Procedure(name = "count_by_name")
2  long getCountByName(@Param("name") String name);
```

3.3. *@Lock*

We can configure the lock mode when we execute a repository query method:

```
1  @Lock(LockModeType.NONE)
2  @Query("SELECT COUNT(*) FROM Person p")
3  long getPersonCount();
```

The available lock modes:

- *READ*
- *WRITE*
- *OPTIMISTIC*
- *OPTIMISTIC_FORCE_INCREMENT*
- *PESSIMISTIC_READ*
- *PESSIMISTIC_WRITE*
- *PESSIMISTIC_FORCE_INCREMENT*
- *NONE*

3.4. *@Modifying*

We can modify data with a repository method if we annotate it with *@Modifying*:

```
1  @Modifying
2  @Query("UPDATE Person p SET p.name = :name WHERE p.id = :i
3  void changeName(@Param("id") long id, @Param("name") Strin
```

For more information, please visit this article.

3.5. *@EnableJpaRepositories*

To use JPA repositories, we have to indicate it to Spring.

We can do this with *@EnableJpaRepositories*.

Note, that we have to use this annotation with *@Configuration*:

```
1  @Configuration
2  @EnableJpaRepositories
3  class PersistenceJPAConfig {}
```

Spring will look for repositories in the sub packages of this *@Configuration* class.

We can alter this behavior with the *basePackages* argument:

```
1  @Configuration
2  @EnableJpaRepositories(basePackages = "org.baeldung.persis
3  class PersistenceJPAConfig {}
```

Also note, that Spring Boot does this automatically if it finds Spring Data JPA on the classpath.

4. Spring Data Mongo Annotations

Spring Data makes working with MongoDB much easier. In the next sections, we'll explore the most basic features of Spring Data MongoDB.

For more information, please visit our article about Spring Data MongoDB.

4.1. *@Document*

This annotation marks a class as being a domain object that we want to persist to the database:

```
1  @Document
2  class User {}
```

It also allows us to choose the name of the collection we want to use:

```
1  @Document(collection = "user")
```

```
2  class User {}
```

Note, that this annotation is the Mongo equivalent of *@Entity* in JPA.

4.2. *@Field*

With *@Field*, we can configure the name of a field we want to use when MongoDB persists the document:

```
1   @Document
2   class User {
3
4       // ...
5
6       @Field("email")
7       String emailAddress;
8
9       // ...
10
11  }
```

Note, that this annotation is the Mongo equivalent of *@Column* in JPA.

4.3. *@Query*

With *@Query*, we can provide a finder query on a MongoDB repository method:

```
1  @Query("{ 'name' : ?0 }")
2  List<User> findUsersByName(String name);
```

4.4. *@EnableMongoRepositories*

To use MongoDB repositories, we have to indicate it to Spring. We can do this with *@EnableMongoRepositories*.

Note, that we have to use this annotation with *@Configuration*:

```
1  @Configuration
2  @EnableMongoRepositories
3  class MongoConfig {}
```

Spring will look for repositories in the sub packages of this *@Configuration* class. We can alter this behavior with the *basePackages* argument:

```
1  @Configuration
2  @EnableMongoRepositories(basePackages = "org.baeldung.repo
3  class MongoConfig {}
```

Also note, that Spring Boot does this automatically if it finds Spring Data MongoDB on the classpath.

5. Conclusion

In this article, we saw which are the most important annotations we need to deal with data in general, using Spring. In addition, we looked into the most common JPA and MongoDB annotations.

As usual, examples are available over on GitHub here for common and JPA annotations, and here for MongoDB annotations.

Spring Bean Annotations

1. Overview

In this article, we'll discuss the most **common Spring bean annotations** used to define different types of beans.

There're several ways to configure beans in a Spring container. We can declare them using XML configuration. We can declare beans using the *@Bean* annotation in a configuration class.

Or we can mark the class with one of the annotations from the *org.springframework.stereotype* package and leave the rest to component scanning.

2. Component Scanning

Spring can automatically scan a package for beans if component scanning is enabled.

@ComponentScan configures which **packages to scan for classes with annotation configuration**. We can specify the base package names directly with one of the *basePackages* or *value* arguments (*value* is an alias for *basePackages*):

```
1  @Configuration
2  @ComponentScan(basePackages = "com.baeldung.annotations")
3  class VehicleFactoryConfig {}
```

Also, we can point to classes in the base packages with the *basePackageClasses* argument:

```
1  @Configuration
2  @ComponentScan(basePackageClasses = VehicleFactoryConfig.c
3  class VehicleFactoryConfig {}
```

Both arguments are arrays so that we can provide multiple packages for each.

If no argument is specified, the scanning happens from the same package where the *@ComponentScan*annotated class is present.

@ComponentScan leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple

times:

```
1  @Configuration
2  @ComponentScan(basePackages = "com.baeldung.annotations")
3  @ComponentScan(basePackageClasses = VehicleFactoryConfig.c
4  class VehicleFactoryConfig {}
```

Alternatively, we can use *@ComponentScans* to specify multiple *@ComponentScan* configurations:

```
1  @Configuration
2  @ComponentScans({
3    @ComponentScan(basePackages = "com.baeldung.annotations"
4    @ComponentScan(basePackageClasses = VehicleFactoryConfig
5  })
6  class VehicleFactoryConfig {}
```

When **using XML configuration**, the configuring component scanning is just as easy:

```
1  <context:component-scan base-package="com.baeldung" />
```

3. *@Component*

@Component is a class level annotation. During the component scan, **Spring Framework automatically detects classes annotated with *@Component***.

For example:

```
1  @Component
2  class CarUtility {
3      // ...
4  }
```

By default, the bean instances of this class have the same name as the class name with a lowercase initial. On top of that, we can specify a different name using the optional *value* argument of this annotation.

Since *@Repository*, *@Service*, *@Configuration*, and *@Controller* are all meta-annotations of *@Component*, they

share the same bean naming behavior. Also, Spring automatically picks them up during the component scanning process.

4. *@Repository*

DAO or Repository classes usually represent the database access layer in an application, and should be annotated with *@Repository:*

```
1  @Repository
2  class VehicleRepository {
3      // ...
4  }
```

One advantage of using this annotation is that **it has automatic persistence exception translation enabled.** When using a persistence framework such as Hibernate, native exceptions thrown within classes annotated with *@Repository* will be automatically translated into subclasses of Spring's *DataAccessExeption*.

To enable exception translation, we need to declare our own *PersistenceExceptionTranslationPostProcessor* bean:

```
1  @Bean
2  public PersistenceExceptionTranslationPostProcessor excepti
3      return new PersistenceExceptionTranslationPostProcesso
4  }
```

Note, that in most cases, Spring does the step above automatically.

Or, via XML configuration:

```
1 <bean class=
2   "org.springframework.dao.annotation.PersistenceExceptionTr
```

5. *@Service*

The **business logic** of an application usually resides within the service layer – so we'll use the *@Service*annotation to indicate that a class belongs to that layer:

```
1  @Service
2  public class VehicleService {
3      // ...
4  }
```

6. *@Controller*

@Controller is a class level annotation which tells the Spring Framework that this class serves as a **controller in Spring MVC**:

```
1  @Controller
2  public class VehicleController {
3      // ...
4  }
```

7. *@Configuration*

Configuration classes can **contain bean definition methods** annotated with *@Bean*:

```
1  @Configuration
2  class VehicleFactoryConfig {
3
4      @Bean
5      Engine engine() {
6          return new Engine();
7      }
8
9  }
```

8. Stereotype Annotations and AOP

When we use Spring stereotype annotations, it’s easy to create a pointcut that targets all classes that have a particular stereotype.

For example, suppose we want to measure the execution time of methods from the DAO layer. We’ll create the following aspect (using AspectJ annotations) taking advantage of *@Repository* stereotype:

```
1   @Aspect
2   @Component
3   public class PerformanceAspect {
4       @Pointcut("within(@org.springframework.stereotype.Repo
5       public void repositoryClassMethods() {};
6
7       @Around("repositoryClassMethods()")
8       public Object measureMethodExecutionTime(ProceedingJoi
9         throws Throwable {
10          long start = System.nanoTime();
11          Object returnValue = joinPoint.proceed();
12          long end = System.nanoTime();
13          String methodName = joinPoint.getSignature().getNa
14          System.out.println(
15            "Execution of " + methodName + " took " +
16            TimeUnit.NANOSECONDS.toMillis(end - start) + " m
17          return returnValue;
18      }
19  }
```

In this example, we created a pointcut that matches all methods in classes annotated with *@Repository*. We used the *@Around* advice to then target that pointcut and determine the execution time of the intercepted methods calls.

Using this approach, we may add logging, performance management, audit, or other behaviors to each application layer.

9. Conclusion

In this article, we have examined the Spring stereotype annotations and learned what type of semantics these each represent.

We also learned how to use component scanning to tell the container where to find annotated classes.

Finally – we saw how these annotations **lead to a clean, layered design** and separation between the concerns of an application. They also make configuration smaller, as we no longer need to explicitly define beans manually.

What is Spring?

Spring is an open source development framework for Enterprise Java. The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make Java EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

What are benefits of Spring Framework?

a. **Lightweight:** Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 2MB.
b. **Inversion of control (IOC):** Loose coupling is achieved

in Spring, with the Inversion of Control technique. The objects give their dependencies instead of creating or looking for dependent objects.

c. **Aspect oriented (AOP):** Spring supports Aspect oriented programming and separates application business logic from system services.

d. **Container:** Spring contains and manages the lifecycle and configuration of application objects.

e. **MVC Framework:** Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks.

f. **Transaction Management:** Spring provides a consistent transaction management interface that can scale down to a local transaction and scale up to global transactions (JTA).

Exception Handling: Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO) into consistent, unchecked exceptions.

Which filter class is needed for spring security?

org.springframework.web.filter.DelegatingFilterProxy.

From the applications perspective, how many user roles needed in spring security. Three user roles are there in spring,

i. Supervisors
ii. Tellers
iii. Plain Users

What are access controls in spring security?

i. To access the account list, you must be authenticated.
ii. The files in the directory "/secure" should only be visible to authenticated users.
iii. The files in the directory "/secure/extreme" should only be visible to Supervisors.

iv. Only Tellers and Supervisors can make withdrawal and deposits.
v. Overdraft limit for an account can be exceeded only by Supervisors

How to restrict static resources processed by spring security filters?

```
<http pattern="/static/**" security="none" />
```

How to add security to method calls made on Spring beans in the application context?

```
<global-method-security pre-post-annotations="enabled" />
```

Will Spring Security secure all the applications?

No, in web application, we need to do some more things to secure full application to save from attackers.

What are all security layers in spring security framework?

i. Authentication
ii. Web request security
iii. Service layer and domain object security

Which java and spring version are needed for spring security?

Spring security 3.0 and jdk 1.5.

When I login in the application where spring security is applied and got the messages "Bad Credentials". What is wrong?

Authentication has failed for the given userid and password.

When I try to login, application goes in endless loop. What is wrong?

It happens when login page is secured resource. Login page should not be secured; it should be marked as ROLE_ANONYMOUS.
g.

Which are the Spring framework modules?

The basic modules of the Spring framework are as following,

a. Core module
b. Bean module
c. Context module
d. Expression Language module
e. JDBC module
f. ORM module
g. OXM module
h. Java Messaging Service(JMS) module
i. Transaction module
j. Web module
k. Web-Servlet module
l. Web-Struts module
m. Web-Portlet module

Explain the Core Container (Application context) module

This is the basic Spring module, which provides the fundamental functionality of the Spring framework. BeanFactory is the heart of any spring-based application. Spring framework was built on the top of this module, which makes the Spring container.

Provide a BeanFactory implementation example.

A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application’s configuration and dependencies from the actual

application code. The most commonly used BeanFactory implementation is the XmlBeanFactory class.

What's XMLBeanFactory ?

The most useful one is **org.springframework.beans.factory.xml.XmlBeanFactory**, which loads its beans based on the definitions contained in an XML file. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

Explain the AOP module

The AOP module is used for developing aspects for our Spring-enabled application. Much of the support has been provided by the AOP Alliance in order to ensure the interoperability between Spring and other AOP frameworks. This module also introduces metadata programming to Spring.

Explain the JDBC abstraction and DAO module

With the JDBC abstraction and DAO module we can be sure that we keep up the database code clean and simple, and prevent problems that result from a failure to close database resources. It provides a layer of meaningful exceptions on top of the error messages given by several database servers. It also makes use of Spring's AOP module to provide transaction management services for objects in a Spring application.

Explain the object/relational mapping integration module

Spring also supports for using of an object/relational mapping (ORM) tool over straight JDBC by providing the ORM module. Spring provides support to tie into several popular ORM frameworks, including Hibernate, JDO, and iBATIS SQL Maps. Spring's transaction management supports each of these

ORM frameworks as well as JDBC.

Explain the web module

The Spring web module is built on the application context module, providing a context that is appropriate for web-based applications. This module also contains support for several web-oriented tasks such as transparently handling multipart requests for file uploads and programmatic binding of request parameters to your business objects. It also contains integration support with Jakarta Struts.

Explain the Spring MVC module

MVC framework is provided by Spring for building web applications. Spring can easily be integrated with other MVC frameworks, but Spring's MVC framework is a better choice, since it uses IoC to provide for a clean separation of controller logic from business objects. With Spring MVC you can declaratively bind request parameters to your business objects.

Spring configuration file

Spring configuration file is an XML file. This file contains the classes information and describes how these classes are configured and introduced to each other.

What is Spring IoC container?

The Spring IoC is responsible for creating the objects,managing them (with dependency injection (DI)), wiring them together, configuring them, as also managing their complete lifecycle.

What are the benefits of IoC?

IOC or dependency injection minimizes the amount of code in an application. It makes easy to test applications, since no

singletons or JNDI lookup mechanisms are required in unit tests. Loose coupling is promoted with minimal effort and least intrusive mechanism. IOC containers support eager instantiation and lazy loading of services.

What are the common implementations of the ApplicationContext?

The **FileSystemXmlApplicationContext** container loads the definitions of the beans from an XML file. The full path of the XML bean configuration file must be provided to the constructor.
The **ClassPathXmlApplicationContext** container also loads the definitions of the beans from an XML file. Here, you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.

The **WebXmlApplicationContext:** container loads the XML file with definitions of all beans from within a web application.

What is the difference between Bean Factory and ApplicationContext?

Application contexts provide a means for resolving text messages, a generic way to load file resources (such as images), they can publish events to beans that are registered as listeners. In addition, operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context. The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable.

What does a Spring application look like?

a. An interface that defines the functions.
b. The implementation that contains properties, its setter and getter methods, functions etc.

c. Spring AOP
d. The Spring configuration XML file.
e. Client program that uses the function

What is Dependency Injection in Spring?

Dependency Injection, an aspect of Inversion of Control (IoC), is a general concept, and it can be expressed in many different ways.This concept says that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container (the IOC container) is then responsible for hooking it all up.

What are the different types of IoC (dependency injection)?

a. **Constructor-based dependency injection:** Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
b. **Setter-based dependency injection:** Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Which DI would you suggest Constructor-based or setter-based DI?

You can use both Constructor-based and Setter-based Dependency Injection. The best solution is using constructor arguments for mandatory dependencies and setters for optional dependencies.

What are Spring beans?

The Spring Beans are Java Objects that form the backbone of a Spring application. They are instantiated, assembled, and managed by the Spring IoC container. These beans are created with the configuration metadata that is supplied to the container, for example, in the form of XML <bean/> definitions.

Beans defined in spring framework are singleton beans. There is an attribute in bean tag named "singleton" if specified true then bean becomes singleton and if set to false then the bean becomes a prototype bean. By default it is set to true. So, all the beans in spring framework are by default singleton beans.

What does a Spring Bean definition contain?

A Spring Bean definition contains all configuration metadata, which is needed for the container to know how to create a bean, its lifecycle details and its dependencies.

How do you provide configuration metadata to the Spring Container?

There are 3 important methods to provide configuration metadata to the Spring Container:

a. XML based configuration file.
b. Annotation-based configuration
c. Java-based configuration

How do you define the scope of a bean?

When defining a ***<bean>*** in Spring, we can also declare a

scope for the bean. It can be defined through the scope attribute in the bean definition. For example, when Spring has to produce a new bean instance each time one is needed, the bean's scope attributes to be prototype. On the other hand, when the same instance of a bean must be returned by Spring every time it is needed, the the bean scope attribute must be set to singleton.

Explain the bean scopes supported by Spring.

There are five scoped provided by the Spring Framework supports following five scopes:

a. In **singleton** scope, Spring scopes the bean definition to a single instance per Spring IoC container.
b. In **prototype** scope, a single bean definition has any number of object instances.
c. In **request** scope, a bean is defined to an HTTP request. This scope is valid only in a web-aware Spring ApplicationContext.
d. In **session** scope, a bean definition is scoped to an HTTP session. This scope is also valid only in a web-aware Spring ApplicationContext.
e. In **global-session** scope, a bean definition is scoped to a global HTTP session. This is also a case used in a web-aware Spring ApplicationContext.
f. The default scope of a Spring Bean is Singleton.

Is Singleton beans thread safe in Spring Framework?

No, singleton beans are not thread-safe in Spring framework.

Explain Bean lifecycle in Spring framework.

a. The spring container finds the bean's definition from the XML file and instantiates the bean.
b. Spring populates all of the properties as specified in the bean definition (DI).
c. If the bean implements BeanNameAware interface, spring passes the bean's id to ***setBeanName()*** method.
d. If Bean implements BeanFactoryAware interface, spring passes the beanfactory to ***setBeanFactory()*** method.
e. If there are any bean BeanPostProcessors associated with the bean, Spring calls ***postProcesserBeforeInitialization()*** method.
f. If the bean implements ***IntializingBean***, its ***afterPropertySet()*** method is called. If the bean has init method declaration, the specified initialization method is called.
g. If there are any BeanPostProcessors associated with the bean, their ***postProcessAfterInitialization()*** methods will be called.
h. If the bean implements DisposableBean, it will call the ***destroy()*** method.

Which are the important beans lifecycle methods? Can you override them?

There are 2 important bean lifecycle methods. The 1st one is setup, which is called when the bean is loaded into the container. The 2nd method is the teardown method, which is called when the bean is unloaded from the container.

The bean tag has 2 important attributes (init-method and destroy-method) with which you can define your own custom initialization and destroy methods. There are also the corresponding annotations (@PostConstruct and @PreDestroy).

What are inner beans in Spring?

When a bean is only used as a property of another bean it can be declared as an inner bean. Spring's XML-based configuration metadata provides the use of ***<bean/>*** element

inside the ***<property/>*** or ***<constructor-arg/>*** elements of a bean definition, in order to define the so-called inner bean. Inner beans are always anonymous and they are always scoped as prototypes.

How can you inject a Java Collection in Spring?

Spring offers the following types of collection configuration elements:

a. The ***<list>*** type is used for injecting a list of values, in the case that duplicates are allowed.
b. The ***<set>*** type is used for wiring a set of values but without any duplicates.
c. The ***<map>*** type is used to inject a collection of name-value pairs where name and value can be of any type.
d. The ***<props>*** type can be used to inject a collection of name-value pairs where the name and value are both Strings.

What is bean wiring?

Wiring, or else bean wiring is the case when beans are combined together within the Spring container. When wiring beans, the Spring container needs to know what beans are needed and how the container should use dependency injection to tie them together.

What is bean auto wiring?

The Spring container is able to autowire relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for a bean by inspecting the contents of the BeanFactorywithout using <constructor-arg> and <property> elements.

Explain different modes of auto wiring?

The autowiring functionality has 5 modes which can be used to instruct Spring container to use autowiring for dependency injection:

a. **no:** This is default setting. Explicit bean reference should be used for wiring.
b. **byName:** When autowiring byName, the Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
c. **byType:** When autowiring by data type, the Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the bean's name in configuration file. If more than one such beans exist, a fatal exception is thrown.
d. **constructor:** This mode is similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
e. **autodetect:** Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

Are there limitations with autowiring?

Limitations of autowiring are:

a. **Overriding:** You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
b. **Primitive data types:** You cannot autowire simple properties such as primitives, Strings, and Classes.

c. **Confusing nature:** Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

Can you inject null and empty string values in Spring?

Yes, you can.

What is Spring Java-Based Configuration? Give some annotation example.

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.

An example is the @Configuration annotation that indicates that the class can be used by the Spring IoC container as a source of bean definitions. Another example is the@Bean annotated method that will return an object that should be registered as a bean in the Spring application context.

What is Annotation-based container configuration?

An alternative to XML setups is provided by annotation-based configuration, which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

How do you turn on annotation wiring?

Annotation wiring is not turned on in the Spring container by default. In order to use annotation based wiring we must enable it in our Spring configuration file by configuring ***<context:annotation-config/>*** element.

What’s @Required annotation?

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws BeanInitializationException if the affected bean property has not been populated.

What's @Autowired annotation?

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. It can be used to autowire bean on the setter method just like @Required annotation, on the constructor, on a property or pn methods with arbitrary names and/or multiple arguments.

What's @Qualifier annotation?

When there are more than one beans of the same type and only one is needed to be wired with a property, the @Qualifier annotation is used along with @Autowired annotation to remove the confusion by specifying which exact bean will be wired.

How can JDBC be used more efficiently in the Spring framework?

When using the Spring JDBC framework the burden of resource management and error handling is reduced. So developers only need to write the statements and queries to get the data to and from the database. JDBC can be used more efficiently with the help of a template class provided by Spring framework, which is the JdbcTemplate.

What's the JdbcTemplate?

JdbcTemplate class provides many convenience methods for doing things such as converting database data into

primitives or objects, executing prepared and callable statements, and providing custom database error handling.

What's the Spring DAO support?

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a consistent way. This allows us to switch between the persistence technologies fairly easily and to code without worrying about catching exceptions that are specific to each technology.

What are the ways to access Hibernate by using Spring?

There are 2 ways to access Hibernate with Spring:

i. Inversion of Control with a Hibernate Template and Callback.
j. Extending HibernateDAOSupport and Applying an AOP Interceptor node.

What's the ORM's Spring support?

Spring supports the following ORM's:

a. Hibernate
b. iBatis
c. JPA (Java Persistence API)
d. TopLink
e. JDO (Java Data Objects)
f. OJB

How can we integrate Spring and Hibernate using *HibernateDaoSupport*?

Use Spring's SessionFactory called LocalSessionFactory. The integration process is of 3 steps:

a. Configure the Hibernate SessionFactory
b. Extend a DAO Implementation from HibernateDaoSupport
c. Wire in Transaction Support with AOP

What are the types of the transaction management Spring support?

Spring supports 2 types of transaction management:

a. **Programmatic transaction management:** This means that you have managed the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
b. **Declarative transaction management:** This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

What are the benefits of the Spring Framework's transaction management?

a. It provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
b. It provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
c. It supports declarative transaction management.
d. It integrates very well with Spring's various data access abstractions.

Which Transaction management type is more preferable?

Most users of the Spring Framework choose declarative transaction management because it is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container. Declarative transaction management is preferable over programmatic transaction management though it is less

flexible than programmatic transaction management, which allows you to control transactions through your code.

Explain Aspect Oriented Programming (AOP)

Aspect-oriented programming, or AOP, is a programming technique that allows programmers to modularize crosscutting concerns, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management.

What's Aspect?

The core construct of AOP is the aspect, which encapsulates behaviors affecting multiple classes into reusable modules. It's a module which has a set of APIs providing crosscutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. In Spring AOP, aspects are implemented using regular classes annotated with the @Aspect annotation (@AspectJ style).

What is the difference between concern and cross-cutting concern in Spring AOP?

The Concern is behavior we want to have in a module of an application. A Concern may be defined as a functionality we want to implement.
The crosscutting concern is a concern, which is applicable throughout the application, and it affects the entire application. For example, logging, security and data transfer are the concerns, which are needed in almost every module of an application. Hence they are crosscutting concerns.

What's the Join point?

The join point represents a point in an application where we can plug-in an AOP aspect. It is the actual place in the

application where an action will be taken using Spring AOP framework.

What's the Advice?

The advice is the actual action that will be taken either before or after the method execution. This is actual piece of code that is invoked during the program execution by the Spring AOP framework.
Spring aspects can work with five kinds of advice:

a. **before:** Run advice before the a method execution.
b. **after:** Run advice after the a method execution regardless of its outcome.
c. **after-returning:** Run advice after the a method execution only if method completes successfully.
d. **after-throwing:** Run advice after the a method execution only if method exits by throwing an exception.
e. **around:** Run advice before and after the advised method is invoked.

What's the Pointcut?

The pointcut is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns.

What is Introduction?

An Introduction allows us to add new methods or attributes to existing classes.

What is Target object?

The target object is an object being advised by one or more aspects. It will always be a proxy object. It is also referred to as the advised object.

What is a Proxy?

A proxy is an object that is created after applying advice to a target object. When you think of client objects the target object and the proxy object are the same.

What are the different types of AutoProxying?

a. BeanNameAutoProxyCreator
b. DefaultAdvisorAutoProxyCreator
c. Metadata autoproxying

What is Weaving? What are the different points where weaving can be applied?

Weaving is the process of linking aspects with other application types or objects to create an advised object. Weaving can be done at compile time, at load time, or at runtime.

Explain XML Schema-based aspect implementation?

In this implementation case, aspects are implemented using regular classes along with XML based configuration.

Explain annotation-based (@AspectJ based) aspect implementation

This implementation case (@AspectJ based implementation) refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

What is Spring MVC framework?

Spring comes with a full-featured MVC framework for building

web applications. Although Spring can easily be integrated with other MVC frameworks, such as Struts, Spring's MVC framework uses IoC to provide a clean separation of controller logic from business objects. It also allows to declaratively binding request parameters to business objects.

What's DispatcherServlet?

The Spring Web MVC framework is designed around a ***DispatcherServlet*** that handles all the HTTP requests and responses.

What's WebApplicationContext?

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

What is Controller in Spring MVC framework?

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

What's the @Controller annotation?

The @Controller annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API.

What’s the @RequestMapping annotation?

@RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

What’s Inversion of Control (IoC)?

Inversion of control (IoC) is programming techniques in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis. This is the action of inverting something or the state of being inverted. The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code. For example, say your application has a text editor component and you want to provide spell checking.

```
public class TextEditor {
      private SpellChecker checker;
      public TextEditor(){
            this.checker = new SpellChecker();
      }
}
```

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

```
public class TextEditor {

       private ISpellChecker checker;
       public TextEditor(ISpellChecker checker){
             this.checker = checker;
       }
}
```

In the first code example we are instantiating SpellChecker (`this.checker = new SpellChecker();`), which means the

TextEditor class directly depends on the SpellChecker class. In the second code example we are creating an abstraction by having the TextEditor constructor signature as the SpellChecker dependency class (not initializing dependency in class). This allows us to call the dependency then pass it to the TextEditor class like so:

```
SpellChecker sc = new SpellChecker; // dependency TextEditor textEditor = new TextEditor(sc);
```

Now, the client creating the TextEditor class has the control over which SpellChecker implementation to use. We're injecting the TextEditor with the dependency. Inversion of Control is what you get when your program callbacks, e.g. like a GUI program.

For example, in an old school menu, you might have:

```
print "enter your name" read name print "enter your address" read address etc... store in database
```

thereby controlling the flow of user interaction.

In a GUI program or somesuch, instead we say

```
when the user types in field a, store it in NAME when the user types in field b, store it in ADDRESS when the user clicks the save button, call StoreInDatabase
```

So now control is inverted... instead of the computer accepting user input in a fixed order, the user controls the order in which the data is entered, and when the data is saved in the database.

Basically, anything with an event loops, callbacks, or execute triggers falls into this category. If you follow these simple two steps, you have done inversion of control:

1. Separate what-to-do part from when-to-do part.
2. Ensure that when part knows as *little* as possible

about what part; and vice versa.

There are several techniques possible for each of these steps based on the technology/language you are using for your implementation. The *inversion* part of the Inversion of Control (IoC) is the confusing thing; because *inversion* is the relative term. The best way to understand IoC is to forget about that word! Examples of the IoC are following,

- Event Handling. Event Handlers (what-to-do part) -- Raising Events (when-to-do part)
- Interfaces. Component client (when-to-do part) -- Component Interface implementation (what-to-do part)
- xUnit fixure. Setup and TearDown (what-to-do part) -- xUnit frameworks calls to Setup at the beginning and TearDown at the end (when-to-do part)
- Template method design pattern. Template method when-to-do part -- primitive subclass implementation what-to-do part
- DLL container methods in COM. DllMain, DllCanUnload, etc (what-to-do part) -- COM/OS (when-to-do part)

1. To me, inversion of control is turning your sequentially written code and turning it into a delegation structure. Instead of your program explicitly controlling everything, your program sets up a class or library with certain functions to be called when certain things happen.

2. It solves code duplication. For example, in the old days you would manually write your own event loop, polling the system libraries for new events. Nowadays, most modern APIs you simply tell the system libraries what events you're interested in, and it will let you know when they happen.

3. Inversion of control is a practical way to reduce code duplication, and if you find yourself copying an entire method and only changing a small piece of the code, you can consider tackling it with inversion of control. Inversion of control is made easy in many languages through the concept of delegates, interfaces, or even raw function pointers.

   It is not appropriate to use in all cases, because the flow of a program can be harder to follow when written this way. It's a useful way to design methods when writing a library that will be reused, but it should be used sparingly in the core of your own program unless it really solves a code duplication problem.

Before using Inversion of Control you should be well aware of the fact that it has its pros and cons and you should know why you use it if you do so.

Pros:

- Your code gets decoupled so you can easily exchange implementations of an interface with alternative implementations
- It is a strong motivator for coding against interfaces instead of implementations
- It's very easy to write unit tests for your code because it depends on nothing else than the objects it accepts in its constructor/setters and you can easily initialize them with the right objects in isolation.

Cons:

- IoC not only inverts the control flow in your program, it also clouds it considerably. This means you can no longer just read your code and jump from one place to another because the connections that would normally be in your code are not in the code anymore. Instead it is in XML

configuration files or annotations and in the code of your IoC container that interprets these metadata.

The first con is incorrect. Ideally there should only be 1 use of IOC container in your code, and that is your main method. Everything else should cascade down from there.

- There arises a new class of bugs where you get your XML config or your annotations wrong and you can spend a lot of time finding out why your IoC container injects a null reference into one of your objects under certain conditions.

Personally I see the strong points of IoC and I really like them but I tend to avoid IoC whenever possible because it turns your software into a collection of classes that no longer constitute a "real" program but just something that needs to be put together by XML configuration or annotation metadata and would fall (and falls) apart without it. Inversion of Control, (or IoC), is about getting freedom (You get married, you lost freedom and you are being controlled. You divorced; you have just implemented Inversion of Control. That's what we called, "decoupled". Good computer system discourages some very close relationship.) More flexibility (The kitchen in your office only serves clean tap water, that is your only choice when you want to drink. Your boss implemented Inversion of Control by setting up a new coffee machine. Now you get the flexibility of choosing either tap water or coffee.) and less dependency (Your partner has a job, you don't have a job, you financially depend on your partner, so you are controlled. You find a job; you have implemented Inversion of Control. Good computer system encourages in-dependency.)

When you use a desktop computer, you have slaved (or say, controlled). You have to sit before a screen and look at it. Using the keyboard to type and using the mouse to navigate.

And badly written software can slave you even more. If you replace your desktop with a laptop, then you somewhat inverted control. You can easily take it and move around. So now you can control where you are with your computer, instead of your computer controlling it. By implementing Inversion of Control, a software/object consumer gets more controls/options over the software/objects, instead of being controlled or having fewer options.

With the above ideas in mind. We still miss a key part of IoC. In the scenario of IoC, the software/object consumer is a sophisticated framework. That means yourself do not call the code you created. Now let's explain why this way works better for a web application.

Suppose your code is a group of workers. They need to build a car. These workers need a place and tools (a software framework) to build the car. A traditional software framework will be like a garage with many tools. So the workers need to make a plan themselves and use the tools to build the car. Building a car is not an easy business, it will be really hard for the workers to plan and cooperate properly. A modern software framework will be like a modern car factory with all the facilities and managers in place. The workers do not have to make any plan; the managers (part of the framework, they are the smartest people and made the most sophisticated plan) will help coordinate so that the workers know when to do their job (framework calls your code). The workers just need to be flexible enough to use any tools the managers give to them (by using Dependency Injection).

Although the workers give the control of managing the project on the top level to the managers (the framework). But it is good to have some professionals help out. This is the concept of **IoC** truly come from.

Modern Web applications with MVC architecture depend on the framework to do URL Routing and put Controllers in place for

the framework to call.

Dependency Injection and Inversion of Control are related. Dependency Injection is at the **micro** level and Inversion of Control is at the **macro** level. You have to eat every bite (implement DI) in order to finish a meal (implement IoC).

But I think you have to be very careful with it. If you will overuse this pattern, you will make very complicated design and even more complicated code. Like in this example with TextEditor: if you have only one SpellChecker maybe it is not really necessary to use IoC? Unless you need to write unit tests or something. Anyway: be reasonable. Design pattern are **good practices** but not Bible to be preached. Do not stick it everywhere. Inversion of Controls is about separating concerns.

Without IoC: You have a **laptop** computer and you accidentally break the screen. And darn, you find the same model laptop screen is nowhere in the market. So you're stuck.

With IoC: You have a **desktop** computer and you accidentally break the screen. You find you can just grab almost any desktop monitor from the market, and it works well with your desktop. Your desktop successfully implements **IoC** in this case. It accepts a variety type of monitors, while the laptop does not; it needs a specific screen to get fixed.

IoC / DI to me is pushing out dependencies to the calling objects. Super simple.

The non-techy answer is being able to swap out an engine in a car right before you turn it on. If everything hooks up right (the interface), you are good. Suppose you are an object. And you go to a restaurant:

Without IoC: you ask for "apple", and you are always served apple when you ask more.

With IoC: You can ask for "fruit". You can get different fruits each time you get served. for example, apple, orange, or water melon.
So, obviously, IoC is preferred when you like the varieties.

1. Inversion of control is a pattern used for decoupling components and layers in the system. The pattern is implemented through injecting dependencies into a component when it is constructed. These dependences are usually provided as interfaces for further decoupling and to support testability. IoC / DI containers such as Castle Windsor, Unity are tools (libraries) which can be used for providing IoC. These tools provide extended features above and beyond simple dependency management, including lifetime, AOP/ Interception, policy, etc.

2.
    a. Alleviates a component from being responsible for managing its dependencies.
    b. Provides the ability to swap dependency implementations in different environments.
    c. Allows a component be tested through mocking of dependencies.
    d. Provides a mechanism for sharing resources throughout an application.
3.
    a. Critical when doing test-driven development. Without IoC it can be difficult to test, because the components under test are highly coupled to the rest of the system.
    b. Critical when developing modular systems. A modular system is a system whose components can be replaced without requiring recompilation.
    c. Critical if there are many crosscutting concerns which need to address, partially in an enterprise application.

Dependency Injection (DI):

Dependency injection generally means passing an object on which method depends, as a parameter to a method, rather than having the method create the dependent object. What it means in practice is that the method does not depend directly on a particular implementation; any implementation that meets the requirements can be passed as a parameter.

Inversion of control as a design guideline serves the following purposes:

There is a decoupling of the execution of a certain task from implementation. Every module can focus on what it is designed for.
Modules make no assumptions about what other systems do but rely on their contracts. Replacing modules has no side effect on other modules
I will keep things abstract here, You can visit following links for detail understanding of the topic. Let to say that we make some meeting in some hotel. Many people, many carafes of water, many plastic cups. When somebody wants to drink, she fill cup, drink and throw cup on the floor. After hour or something we have a floor covered of plastic cups and water.

Let invert control.

The same meeting in the same place, but instead of plastic cups we have a waiter with one glass cup (Singleton) and she all of time offers to guests drinking. When somebody wants to drink, she gets from waiter glass, drink and return it

back to waiter. Leaving aside the question of the hygienic, last form of drinking process control is much more effective and economic.

And this is exactly what the Spring (another IoC container, for example: Guice) does. Instead of let to application create what it need using new keyword (taking plastic cup), Spring IoC container all of time offer to application the same instance (singleton) of needed object (glass of water). Think about yourself as organizer of such meeting. You need the way to message to hotel administration that meeting members will need glass of water but not piece of cake.

```
public class MeetingMember {

    private GlassOfWater glassOfWater;

    public void setGlassOfWater(GlassOfWater glassOfWater){
        this.glassOfWater = glassOfWater;
    }

    //your glassOfWater object initialized and ready to use...
    //spring IoC  called setGlassOfWater method itself in order to
    //offer to meetingMember glassOfWater instance
}
```

IoC provides the use of Interface as a way of specific something (such a field or a parameter) as a wildcard that can be used by some classes. It allows the re-usability of the code.

For example, let's say that we have two classes: Dog and Cat. Both share the same qualities/states: age, size, weight. So instead of creating a class of service called DogService and CatService, I can create a single one called AnimalService that allows using Dog and Cat only if they use the interface IAnimal.

However, pragmatically speaking, it has some backwards.

a. **Most of the developers don't know how to use it.** For example, I can create a class called **Customer** and **I can create automatically** (using the tools of the IDE) an interface called **ICustomer**. So, it's not rare to find a folder filled with classes and interfaces, no matter if the interfaces will be reused or not. It's called BLOATED. Some people could argue that "may be in the future we could use it". :-|

b. It has some limitings. For example, let's talk about the case of **Dog** and **Cat** and I want to add a new service (functionality) only for dogs. Let's say that I want to calculate the number of days that I need to train a dog (**`trainDays()`**), for cat it's useless, cats can't be trained (I'm joking).

    i. If I add `trainDays()` to the Service **AnimalService** then it also works with cats and it's not valid at all.
    ii. I can add a condition in `trainDays()` where it evaluates which class is used. But it will break completely the IoC.
    iii. I can create a new class of service called **DogService** just for the new functionality. But, it will increase the maintainability of the code because we will have two classes of service (with similar functionality) for **Dog** and it's bad.

Objects that are statically assigned to one another determine the flow of the business logic. With inversion of control, the flow depends on the object graph that is instantiated by the assembler and is made possible by object interactions being defined through abstractions. The binding process is achieved through **dependency injection**, although some argue that the use of a service locator also provides

inversion of control. Inversion of control as a ***design guideline*** serves the following purposes:

a. There is a decoupling of the execution of a certain task from implementation.
b. Every module can focus on what it is designed for.
c. Modules make no assumptions about what other systems do but rely on their contracts.
d. Replacing modules has no side effect on other modules.

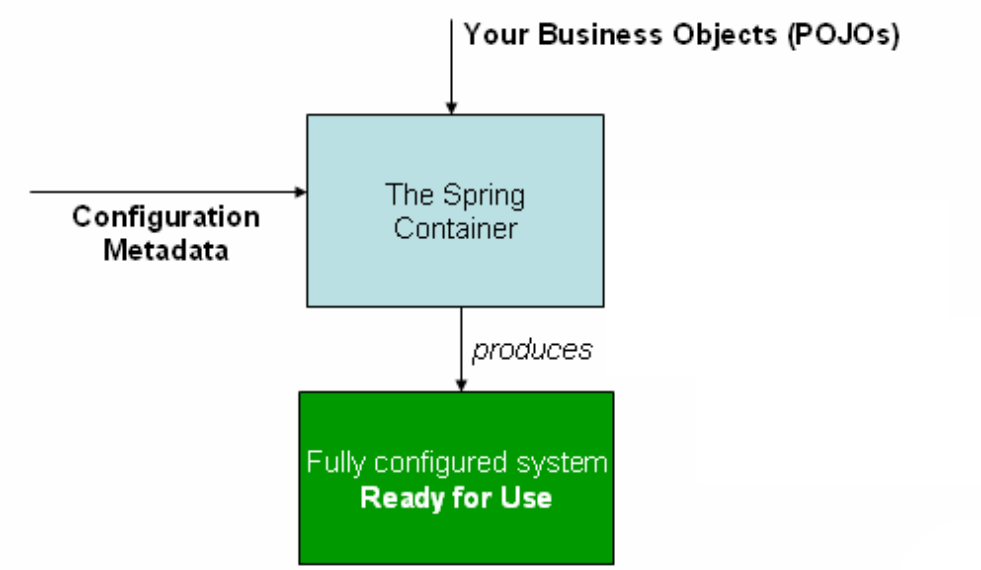
Inversion of control is a design paradigm with the goal of giving more control to the targeted components of your application, the ones getting the work done. Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compiles time which class will be used to provide that functionality. Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality.
The two concepts work together in this way to allow for much more flexible, reusable, and encapsulated code to be written. As such, they are important concepts in designing object-oriented solutions. In object-oriented programming, there are several basic techniques to implement inversion of control. These are:

a. Using a factory pattern
b. Using a service locator pattern
c. Using a **dependency injection** of any given below type:

    i. A constructor injection
    ii. A setter injection
    iii. An interface injection

The **org.springframework.beans** and **org.springframework.contex**t packages provide the basis for the Spring Framework's IoC container. The **BeanFactory** interface provides an advanced configuration mechanism capable of managing objects of any

nature. The **ApplicationContext** interface builds on top of the BeanFactory (it is a sub-interface) and adds other functionality such as easier integration with Spring's AOP features, message resource handling (for use in internationalization), event propagation, and application-layer specific contexts such as the WebApplicationContext for use in web applications. The **org.springframework.beans.factory.BeanFactory is the actual representation of the Spring IoC** container that is responsible for containing and otherwise managing the aforementioned beans. The BeanFactory interface is the central IoC container interface in Spring.



There are a number of implementations of the BeanFactory interface. The most commonly used BeanFactory implementation is the **XmlBeanFactory** class. Other commonly used class is **XmlWebApplicationContext**. Depending on the bean definition, the factory will return either an independent instance of a contained object (the Prototype design

pattern), or a single shared instance (a superior alternative to the Singleton design pattern, in which the instance is a singleton in the scope of the factory). Which type of instance will be returned depends on the bean factory configuration: the API is the same.

Before we dive into dependency injection types, let first identify the ways of creating a bean in spring framework, as it will help in understanding the things in next section.

A bean definition can be seen as a recipe for creating one or more actual objects. The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

When creating a bean using the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being created does not need to implement any specific interfaces or be coded in a specific fashion. Just specifying the bean class should be enough. When using XML-based configuration metadata you can specify your bean class like so:

```
<bean id="exampleBean"/>
```

When defining a bean, which is to be created using a static factory method, along with the class attribute, which specifies the class containing the static factory method, another attribute named factory-method is needed to specify the name of the factory method itself.

```
<bean id="exampleBean" factory-method="createInstance"/>
```

Spring expects to be able to call this method and get back a live object, which from that point on is treated as if it had been created normally via a constructor.

In a fashion similar to instantiation via a static factory

method, instantiation using an instance factory method is where the factory method of an existing bean from the container is invoked to create the new bean.

```
<bean id="myFactoryBean"  class="...">

<bean id="exampleBean"  factory-bean="myFactoryBean"
            factory-method="createInstance"></bean>
```

The basic principle behind Dependency Injection (DI) is that objects define their dependencies only through constructor arguments, arguments to a factory method, or properties, which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually inject those dependencies when it creates the bean. This is fundamentally the inverse, hence the name Inversion of Control (IoC).

Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

```
public class TestSetter {

    DemoBean demoBean = null;

    public void setDemoBean(DemoBean demoBean) {
        this.demoBean = demoBean;
    }
}
```

What’s Constructor injection?

Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator. Additionally, calling a static factory method with specific arguments to construct the bean, can be considered almost equivalent, and the rest of this text will

consider arguments to a constructor and arguments to a static factory method similarly.

Interface injection In this methodology we implement an interface from the IOC framework. IOC framework will use the interface method to inject the object in the main class. It is much more appropriate to use this approach when you need to have some logic that is not applicable to place in a property. Such as logging support.

```
public void SetLogger(ILogger logger){

  _notificationService.SetLogger(logger);
  _productService.SetLogger(logger);
}
```

What is difference between component and service?

A component is a glob of software that's intended to be used, without change, by an application that is out of the control of the writers of the component. By 'without change' means that the using application doesn't change the source code of the components, although they may alter the component's behavior by extending it in ways allowed by the component writers.

A service is similar to a component in that it's used by foreign applications. The main difference is that a component to be used locally (think jar file, assembly, dll, or a source import). A service will be used remotely through some remote interface, either synchronous or asynchronous (eg web service, messaging system, RPC, or socket.)

How DI is different from Service locator pattern?

The key benefit of a Dependency Injector is that it allows to plug-in a suitable implementation of a service according

to environment and usage. Injection isn't the only way to break this dependency, another is to use a service locator. The basic idea behind a service locator is to have an object that knows how to get hold of all of the services that an application might need. It then scans all such services and store them as a singleton Registry. When asked for a service implementation, a requester can query the registry with a token and get appropriate implementation.

Mostly these registries are populated via some configuration files. The key difference is that with a Service Locator every user of a service has a dependency to the locator. The locator can hide dependencies to other implementations, but you do need to see the locator.

Which one should be better to use i.e. service locator or dependency injection?

Well, it as I already said that key difference is that with a Service Locator every user of a service has a dependency to the locator. It means you must know the details of service locator in terms of input and output. So, it actually becomes the deciding factor which pattern to choose from.

If it is easy and necessary to maintain registry information then go for service locator, or else simply use dependency injection as it does not bother the users of service with any per-requisites.

Which is better constructor injection or setter injection?

The choice between setter and constructor injection is interesting as it mirrors a more general issue with object-oriented programming – should you fill fields in a constructor or with setters. Constructors with parameters give you a clear statement of

what it means to create a valid object in an obvious place. If there's more than one way to do it, create multiple constructors that show the different combinations. Another advantage with constructor initialization is that it allows you to clearly hide any fields that are immutable by simply not providing a setter. I think this is important – if something shouldn't change then the lack of a setter communicates this very well. If you use setters for initialization, then this can become a pain.

But If you have a lot of constructor parameters things can look messy, particularly in languages without keyword parameters. If you have multiple ways to construct a valid object, it can be hard to show this through constructors, since constructors can only vary on the number and type of parameters. Constructors also suffer if you have simple parameters such as strings. With setter injection you can give each setter a name to indicate what the string is supposed to do. With constructors you are just relying on the position, which is harder to follow.

My preference is to start with constructor injection, but be ready to switch to setter injection as soon as the problems I've outlined above start to become a problem.

What is Bean Factory?

A BeanFactory is like a factory class that contains a collection of beans. The BeanFactory holds Bean Definitions of multiple beans within itself and then instantiates the bean whenever asked for by clients.

BeanFactory is able to create associations between collaborating objects as they are instantiated. This removes the burden of configuration from bean itself and the beans client. BeanFactory also takes part in the life cycle of a bean, making calls to custom initialization and destruction

methods.

What is Application Context?

A bean factory is fine to simple applications, but to take advantage of the full power of the Spring framework, you may want to move up to Springs more advanced container, the application context. On the surface, an application context is same as a bean factory. Both load bean definitions, wire beans together, and dispense beans upon request. But it also provides:

- A means for resolving text messages, including support for internationalization
- A generic way to load file resources
- Events to beans that are registered as listeners

What are the common implementations of the Application Context?

The three commonly used implementation of 'Application Context' are

1) ClassPathXmlApplicationContext: It Loads context definition from an XML file located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code.

ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

2) FileSystemXmlApplicationContext : It loads context definition from an XML file in the filesystem. The application context is loaded from the file system by using the code.

ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");

3) XmlWebApplicationContext : It loads context definition from an XML file contained within a web application.

What should be used preferably BeanFactory or ApplicationContext?

A BeanFactory pretty much just instantiates and configures beans. An ApplicationContext also does that, and it provides the supporting infrastructure to enable lots of enterprise-specific features such as transactions and AOP. In short, favor the use of an ApplicationContext.

What's the dependency injection (DI)?

Basically, instead of having your objects creating a dependency or asking a factory object to make one for them, you pass the needed dependencies in to the constructor or via property setters, and you make it somebody else's problem (an object further up the dependency graph, or a dependency injector that builds the dependency graph). A dependency as I'm using it here is any other object the current object needs to hold a reference to.

One of the major advantages of dependency injection is that it can make testing lots easier. Suppose you have an object which in its constructor does something like

```
public SomeClass() {
    myObject = Factory.getObject();
}
```

This can be troublesome when all you want to do is run some

unit tests on SomeClass, especially if myObject is something that does complex disk or network access. So now you're looking at mocking myObject but also somehow intercepting the factory call. Hard. Instead, pass the object in as an argument to the constructor. Now you've moved the problem elsewhere, but testing can become lots easier. Just make a dummy myObject and pass that in. The constructor would now look a bit like:

```
1
2  public SomeClass (MyClass myObject) {
3
4      this.myObject = myObject;
5  }
6
```

MEMORY MANAGEMENT

What does the statement “memory is managed in Java” mean?

Memory is the key resource an application requires to run e ectively and like any resource, it is scarce. As such, its allocation and deallocation to and from applications or di erent parts of an application require a lot of care and consideration.

However, in Java, a developer does not need to explicitly

allocate and deallocate memory – the JVM and more speci cally the Garbage Collector – has the duty of handling memory allocation so that the developer doesn’t have to.

This is contrary to what happens in languages like C where a programmer has direct access to memory and literally references memory cells in his code, creating a lot of room for memory leaks.

What is Garbage Collection and what are its advantages?

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deallocation from us so that we can focus on solving the problem at hand.

Are there any disadvantages of Garbage Collection?

Yes. Whenever the garbage collector runs, it has an e ect on the application’s performance. This is because all other threads in the application have to be stopped to allow the garbage collector thread to e ectively do its work.

Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using di erent GC algorithms.

What is the meaning of the term “stop-the-world”?

When the garbage collector thread is running, other threads are stopped, meaning the application is stopped momentarily. This is analogous to house cleaning or fumigation where occupants are denied access until the process is complete.

Depending on the needs of an application, “stop the world” garbage collection can cause an unacceptable freeze. This is why it is important to do garbage collector tuning and JVM optimization so that the freeze encountered is at least acceptable.

What are stack and heap? What is stored in each of these memory structures, and how are they interrelated?

The stack is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap de ned in currently executing methods.

This structure allows the runtime to return from the method knowing the address whence it was called, and also clear all

local variables after exiting the method. Every thread has its own stack.

The heap is a large bulk of memory intended for allocation of objects. When you create an object with the new keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

What is generational garbage collection and what makes it a popular garbage collection approach?

Generational garbage collection can be loosely de ned as the strategy used by the garbage collector where the heap is divided into a number of sections called generations, each of which will hold objects according to their “age” on the heap.

Whenever the garbage collector is running, the rst step in the process is called marking. This is where the garbage collector identi es which pieces of memory are in use and which are not. This can be a very time-consuming process if all objects in a system must be scanned.

As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short-lived.

With generational garbage collection, objects are grouped according to their “age” in terms of how many garbage collection cycles they have survived. This way, the bulk of

the work spread across various minor and major collection cycles.

Today, almost all garbage collectors are generational. This strategy is so popular because, over time, it has proven to be the optimal solution.

Describe in detail how generational garbage collection works

To properly understand how generational garbage collection works, it is important to rst **remember how Java heap is structured** to facilitate generational garbage collection.

The heap is divided up into smaller spaces or generations. These spaces are Young Generation, Old or Tenured Generation, and Permanent Generation.

The **young generation hosts most of the newly created objects**. An empirical study of most applications shows that majority of objects are quickly short lived and therefore, soon become eligible for collection. Therefore, new objects start their journey here and are only “promoted” to the old generation space after they have attained a certain “age”. The term **“age”** in generational garbage collection **refers to the number of collection cycles the object has survived**.

The young generation space is further divided into three spaces: an Eden space and two survivor spaces such as Survivor 1 (s1) and Survivor 2 (s2).

The **old generation hosts objects that have lived in memory**

longer than a certain “age”. The objects that survived garbage collection from the young generation are promoted to this space. It is generally larger than the young generation. As it is bigger in size, the garbage collection is more expensive and occurs less frequently than in the young generation.

The **permanent generation or more commonly called, PermGen, contains metadata required by the JVM** to describe the classes and methods used in the application. It also contains the string pool for storing interned strings. It is populated by the JVM at runtime based on classes in use by the application. In addition, platform library classes and methods may be stored here.

First, **any new objects are allocated to the Eden space**. Both survivor spaces start out empty. When the Eden space lls up, a minor garbage collection is triggered. Referenced objects are moved to the rst survivor space. Unreferenced objects are deleted.

During the next minor GC, the same thing happens to the Eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1).

In addition, objects from the last minor GC in the rst survivor space (S0) have their age incremented and are moved to S1. Once all surviving objects have been moved to S1, both S0 and Eden space are cleared. At this point, S1

contains objects with di erent ages.

At the next minor GC, the same process is repeated. However this time the survivor spaces switch. Referenced objects are moved to S0 from both Eden and S1. Surviving objects are aged. Eden and S1 are cleared.

After every minor garbage collection cycle, the age of each object is checked. Those that have reached a certain arbitrary age, for example, 8, are promoted from the young generation to the old or tenured generation. For all subsequent minor GC cycles, objects will continue to be promoted to the old generation space.

This pretty much exhausts the process of garbage collection in the young generation. Eventually, a major garbage collection will be performed on the old generation which cleans up and compacts that space. For each major GC,

When does an object become eligible for garbage collection? Describe how the GC collects an eligible object?

An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.

The most straightforward case of an object becoming eligible for garbage collection is if all its references are null. Cyclic dependencies without any live external reference are also eligible for GC. So if object A references object B and object B references Object A and they don’t have any other

live reference then both Objects A and B will be eligible for Garbage collection.

Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

Q9. How do you trigger garbage collection from Java code?

You, as Java programmer, can not force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.

Before removing an object from memory garbage collection thread invokes finalize() method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.

Additionally, there are methods like System.gc() and Runtime.gc() which is used to send request of Garbage collection to JVM but it’s not guaranteed that garbage collection will happen.

What happens when there is not enough heap space to accommodate storage of new objects?

If there is no memory space for creating a new object in

Heap, Java Virtual Machine throws OutOfMemoryError or more speci cally **java.lang.OutOfMemoryError heap space.**

Is it possible to «resurrect» an object that became eligible for garbage collection?

When an object becomes eligible for garbage collection, the GC has to run the nalize method on it. The nalize method is guaranteed to run only once, thus the GC ags the object as nalized and gives it a rest until the next cycle.

In the nalize method you can technically “resurrect” an object, for example, by assigning it to a static eld. The object would become alive again and non- eligible for garbage collection, so the GC would not collect it during the next cycle.

The object, however, would be marked as nalized, so when it would become eligible again, the nalize method would not be called. In essence, you can turn this “resurrection” trick only once for the lifetime of the object. Beware that this ugly hack should be used only if you really know what you’re doing – however, understanding this trick gives some insight into how the GC works.

Describe strong, weak, soft and phantom references and their role in garbage collection.

Much as memory is managed in Java, an engineer may need to perform as much optimization as possible to minimize latency

and maximize throughput, in critical applications. Much as **it is impossible to explicitly control when garbage collection is triggered** in the JVM, **it is possible to in uence how it occurs as regards the objects we have created.**

Java provides us with reference objects to control the relationship between the objects we create and the garbage collector.

By default, every object we create in a Java program is strongly referenced by a variable:

```
StringBuilder sb = new StringBuilder();
```

In the above snippet, the new keyword creates a new StringBuilder object and stores it on the heap. The variable sb then stores a **strong reference** to this object. What this means for the garbage collector is that the particular StringBuilder object is not eligible for collection at all due to a strong reference held to it by sb. The story only changes when we nullify sb like this:

1 sb = null; After calling the above line, the object will then be eligible for collection.

We can change this relationship between the object and the garbage collector by explicitly wrapping it inside another reference object which is located inside java.lang.ref package.

A **soft reference** can be created to the above object like

this:

```
. 1  StringBuilder sb = new StringBuilder();

. 2      SoftReference<StringBuilder>     sbRef     =     new
     SoftReference<>(sb);

. 3  sb = null;
```

In the above snippet, we have created two references to the StringBuilder object. The rst line creates a **strong reference** sb and the second line creates a **soft reference** sbRef. The third line should make the object eligible for collection but the garbage collector will postpone collecting it because of sbRef.

The story will only change when memory becomes tight and the JVM is on the brink of throwing an OutOfMemory error. In other words, objects with only soft references are collected as a last resort to recover memory.

A **weak reference** can be created in a similar manner using WeakReference class. When sb is set to null and the StringBuilder object only has a weak reference, the JVM’s garbage collector will have absolutely no compromise and immediately collect the object at the very next cycle.

A **phantom reference** is similar to a weak reference and an object with only phantom references will be collected without waiting. However, phantom references are enqueued as soon as their objects are collected. We can poll the reference queue to know exactly when the object was

collected.

Suppose we have a circular reference (two objects that reference each other). Could such pair of objects become eligible for garbage collection and why?

Yes, a pair of objects with a circular reference can become eligible for garbage collection. This is because of how Java's garbage collector handles circular references. It considers objects live not when they have any reference to them, but when they are reachable by navigating the object graph starting from some garbage collection root (a local variable of a live thread or a static eld). If a pair of objects with a circular reference is not reachable from any root, it is considered eligible for garbage collection.

How are strings represented in memory?

A String instance in Java is an object with two elds: a char[] value eld and an int hash eld. The value eld is an array of chars representing the string itself, and the hash eld contains the hashCode of a string which is initialized with zero, calculated during the rst hashCode() call and cached ever since. As a curious edge case, if a hashCode of a string has a zero value, it has to be recalculated each time the hashCode() is called.

Important thing is that a String instance is immutable: you can't get or modify the underlying char[] array. Another feature of strings is that the static constant strings are

loaded and cached in a string pool. If you have multiple identical String objects in your source code, they are all represented by a single instance at runtime.

What is a StringBuilder and what are its use cases? What is the di erence between appending a string to a StringBuilder and concatenating two strings with a + operator? How does StringBuilder different from StringBuilder?

StringBuilder allows manipulating character sequences by appending, deleting and inserting characters and strings. This is a mutable data structure, as opposed to the String class which is immutable.

When concatenating two String instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. StringBuilder allows handling string manipulations much more e ciently.

StringBuffer is different from *StringBuilder* in that it is thread-safe. If you need to manipulate a string in a single thread, use StringBuilder instead.

StringBuffers are thread-safe, meaning that they have synchronized methods to control access so that only one thread can access a StringBuffer object's synchronized code at a time. Thus, StringBuffer objects are generally safe to use in a multi-threaded environment where multiple threads may be trying to access the same StringBuffer object at the same time.

`StringBuilder's` access is not synchronized so that it is not thread-safe. By not being synchronized, the performance of StringBuilder can be better than StringBuffer. Thus, if you are working in a single-threaded environment, using StringBuilder instead of StringBuffer may result in increased performance. This is also true of other situations such as a StringBuilder local variable (ie, a variable within a method) where only one thread will be accessing a StringBuilder object.

So, prefer `StringBuilder` because,

- Small performance gain.
- StringBuilder is a 1:1 drop-in replacement for the StringBuffer class.
- StringBuilder is not thread synchronized and therefore performs better on most implementations of Java

How does the **static allocation occurred in Java (heap, stack and permanent generation)?**

I have been lately reading a lot on memory allocation schemes in java, and there have been many doubts as I have been reading from various sources. I have collected my concepts, and I would request to go through all of the points and comment on them. I came to know that memory allocation is JVM specific, so I must say beforehand, that my question is Sun specific.

1. Classes (loaded by the classloaders) go in a special area on heap : Permanent Generation
2. All the information related to a class like name of the class, Object arrays associated with the class, internal objects used by JVM (like java/lang/Object) and

optimization information goes into the Permanent Generation area.

3. All the static member variables are kept on the Permanent Generation area again.
4. Objects go on a different heap : Young generation
5. There is only one copy of each method per class, be the method static or non-static. That copy is put in the Permanent Generation area. For non-static methods, all the parameters and local variables go onto the stack - and whenever there is a concrete invocation of that method, we get a new stack-frame associated with it. I am not sure where are the local variables of a static method are stored. Are they on the heap of Permanent Generation ? Or just their reference is stored in the Permanent Generation area, and the actual copy is somewhere else (Where ?)
6. I am also unsure where does the return type of a method get stored.
7. If the objects (in the young generation) needs to use a static member (in the permanent generation), they are given a reference to the static member && they are given enough memory space to store the return type of the method,etc.

First, as should be clear to you by now that there are very few people who can confirm these answers from first hand knowledge. Very few people have worked on recent HotSpot JVMs or studied them to the depth needed to really know. Most people here (myself included) are answering based on things they have seen written elsewhere, or what they have inferred. Usually what is written here, or in various articles and web pages, is based on other sources which may or may not be definitive. Often it is simplified, inaccurate or just plain wrong.

If you want definitive confirmation of your answers, you really need to download the OpenJDK sourcecode ... and *do*

your own research by reading and understanding the source code. Asking questions on SO, or trawling through random web articles is not a sound academic research technique.
Having said that ...

1) Classes (loaded by the classloaders) go in a special area on heap : Permanent Generation.

AFAIK, yes. (**Update**: see below.)

2) All the information related to a class like name of the class, Object arrays associated with the class, internal objects used by JVM (like java/lang/Object) and optimization information goes into the Permanent Generation area.

More or less, yes. I'm not sure what you mean by some of those things. I'm guessing that "internal objects used by JVM (like java/lang/Object)" means JVM-internal class descriptors.

3) All the static member variables are kept on the Permanent Generation area again.

The variables themselves yes. These variables (like all Java variables) will hold either primitive values or object references. However, while the static member variables are in a frame that is allocated in the permgen heap, the objects/arrays referred to by those variables may be allocated in *any* heap.

4) Objects go on a different heap : Young generation

Not necessarily. Large objects *may* be allocated directly into the tenured generation.

5) There is only one copy of each method per class, be the method static or non-static. That copy is put in the Permanent Generation area.

Assuming that you are referring to the code of the method, then AFAIK yes. It may be a little more complicated though. For instance that code may exist in bytecode and/or native code forms at different times during the JVM's life.

... For non-static methods, all the parameters and local variables go onto the stack - and whenever there is a concrete invocation of that method, we get a new stack-frame

associated with it.
Yes.

... I am not sure where are the local variables of a static method are stored. Are they on the heap of Permanent Generation ? Or just their reference is stored in the Permanent Generation area, and the actual copy is somewhere else (Where ?)
No. They are stored on the stack, just like local variables in non-static methods.

6) I am also unsure where does the return type of a method get stored.
If you mean the *value* returned by a (non-void) method call, then it is either returned on the stack or in a machine register. If it is returned on the stack, this takes 1 or two words, depending on the return type.
7) If the objects (in the young generation) nees to use a static member (in the permanent generation), they are given a reference to the static member && they are given enough memory space to store the return type of the method,etc.
That is inaccurate (or at least, you are not expressing yourself clearly).

If some method accesses a static member variable, what it gets is either a primitive value or an object **reference**. This may be assigned to an (existing) local variable or parameter, assigned to an (existing) static or non-static member, assigned to an (existing) element of a previously allocated array, or simply used and discarded.

- In no case does *new* storage need to be allocated to hold either a reference or a primitive value.
- Typically, one word of memory is all that is needed to store an object or array reference, and a primitive value typically occupies one or two words, depending on the hardware architecture.
- In no case does space need to be allocated by the caller to hold some object / array returned by a method. In Java, objects and arrays are always returned using pass-

> by-value semantics ... but that value that is is returned is an object or array reference.

What is the difference between PermGen and Metaspace?

Until Java 7 there was an area in JVM memory called PermGen, where JVM used to keep its classes. In Java 8 it was removed and replaced by area called Metaspace.

What are the most important differences between PermGen and Metaspace?

The only difference I know is that java.lang.OutOfMemoryError: PermGen space can no longer be thrown and the VM parameter MaxPermSize is ignored.

The main difference from a user perspective - which I think the previous answer does not stress enough - is that **Metaspace by default auto increases** its size (up to what the underlying OS provides), while PermGen always has a fixed maximum size. You can set a fixed maximum for Metaspace with JVM parameters, but you cannot make PermGen auto increase.
To a large degree it is just a change of name. Back when PermGen was introduced, there was no Java EE or dynamic class(un)loading, so once a class was loaded it was stuck in memory until the JVM shut down - thus *Permanent* Generation. Nowadays classes may be loaded and unloaded during the lifespan of the JVM, so Metaspace makes more sense for the area where the metadata is kept.
Both of them contain the `java.lang.Class` instances and both of them suffer from ClassLoader leaks. Only difference is that with Metaspace default settings, it takes longer until

you notice the symptoms (since it auto increases as much as it can), i.e. you just push the problem further away without solving it. OTOH I imagine the effect of running out of OS memory can be more severe than just running out of JVM PermGen, so I'm not sure it is much of an improvement.
Whether you're using a JVM with PermGen or with Metaspace, if you are doing dynamic class unloading, you should to take measures against classloader leaks.

PermGen has been completely removed.

Metaspace garbage collection - Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the `MaxMetaspaceSize`.
The space `Metadata` was held is no longer contiguous to the `Java heap`, The `metadata` has now moved to native memory to an area known as the `Metaspace`.

In Simple words,
Since the class metadata is allocated out of native memory, the max available space is the total available system memory. Thus, you will no longer encounter `OOM errors` and could end up spilling into the swap space.
The removal of `PermGen` doesn’t mean that your class loader leak issues are gone. So, yes, you will still have to monitor your consumption and plan accordingly, since a leak would end up consuming your entire native memory.