Core java interview questions

How to create a immutable object in Java? Count all benefits?

An immutable class is one whose state can not be changed once created. Here, state of object essentially means the values stored in instance variable in class whether they are primitive types or reference types.

To make a class immutable, below steps needs to be followed:

1. Don’t provide “setter” methods or methods that modify fields or objects referred to by fields. Setter methods are meant to change the state of object and this is what we want to prevent here.
2. Make all fields final and private. Fields declared private will not be accessible outside the class and making them final will ensure the even accidentally you can not change them.
3. Don’t allow subclasses to override methods. The simplest way to do this is to declare the class as final. Final classes in java can not be overridden.
4. Always remember that your instance variables will be either mutable or immutable. Identify them and return new objects with copied content for all mutable objects (object references). Immutable variables (primitive types) can be returned safely without extra effort.

Also, you should memorize following benefits of immutable class. You might need them during interview. Immutable classes --

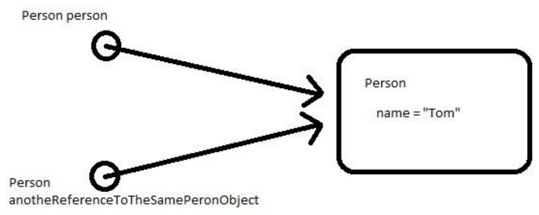
- are simple to construct, test, and use
- are automatically thread-safe and have no synchronization issues
- do not need a copy constructor
- do not need an implementation of clone
- allow hashCode to use lazy initialization, and to cache its return value
- do not need to be copied defensively when used as a field
- make good Map keys and Set elements (these objects must not change state while in the collection)
- have their class invariant established once upon construction, and it never needs to be checked again

- always have “failure atomicity” (a term used by Joshua Bloch) : if an immutable object throws an exception, it’s never left in an undesirable or indeterminate state.

Take a look an example written in **this post**.

Is Java Pass by Reference or Pass by Value?

The Java Spec says that ***everything in Java is pass-by-value***. There is no such thing as “pass-by-reference” in Java. These terms are associated with method calling and passing variables as method parameters. Well, primitive types are always pass by value without any confusion. But, the concept should be understood in context of method parameter of complex types.
In java, when we pass a reference of complex types as any method parameters, always the memory address is copied to new reference variable bit by bit. See in below picture:



In above example, address bits of first instance are copied to another reference variable, thus resulting both references to point a single memory location where actual object is stored. Remember, making another reference to null will not make first reference also null. But, changing state from either reference variable have impact seen in other reference also.
Read in detail here: **http://stackoverflow.com/questions/40480/is-java-pass-by-reference**

What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

Why there are two Date classes; one in java.util package and another in java.sql?

A java.util.Date represents date and time of day, a java.sql.Date only represents a date. The complement of java.sql.Date is java.sql.Time, which only represents a time of day. The java.sql.Date is a subclass (an extension) of java.util.Date. So, what changed in java.sql.Date:

-- toString() generates a different string representation: yyyy-mm-dd
-- a static valueOf(String) methods to create a Date from a String with above representation
-- the getters and setter for hours, minutes and seconds are deprecated

The java.sql.Date class is used with JDBC and it was intended to not have a time part, that is, hours, minutes, seconds, and milliseconds should be zero… but this is not enforced by the class.

Explain marker interfaces?

The marker interface pattern is a design pattern in computer science, used with languages that **provide run-time type information about objects**. It provides **a means to associate metadata with a class where the language does not have explicit support for such metadata.** In java, it is used as interfaces with no method specified.
A good example of use of marker interface in java is Serializable interface. A class implements this interface to indicate that its non-transient data members can be written to a byte steam or file system.
A *major problem* with marker interfaces is that an interface defines a contract for implementing classes, and that contract is inherited by all subclasses. This means that you cannot “un-implement” a marker. In the example given, if you create a subclass that

you do not want to serialize (perhaps because it depends on transient state), you must resort to explicitly throwing NotSerializableException.

Why main() in java is declared as public static void?

Why public? main method is public so that it can be accessible everywhere and to every object which may desire to use it for launching the application. Here, i am not saying that JDK/JRE had similar reasons because java.exe or javaw.exe (for windows) use Java Native Interface (JNI) calls to invoke method, so they can have invoked it either way irrespective of any access modifier.
Why static? Lets suppose we do not have main method as static. Now, to invoke any method you need an instance of it. Right? Java can have overloaded constructors, we all know. Now, which one should be used and from where the parameters for overloaded constructors will come.
Why void? Then there is no use of returning any value to JVM, who actually invokes this method. The only thing application would like to communicate to invoking process is: normal or abnormal termination. This is already possible using System.exit(int). A non-zero value means abnormal termination otherwise everything was fine.

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

When we create string with new() it's created in heap and also added into string pool, while String created using literal are created in String pool only which exists in Perm area of heap.

Well you really need to know the concept of string pool very deeply to answer this question or similar questions. My advise.. "Study Hard" about string class and string pool.

How does substring () inside String works?

String in java are like any other programming language, a sequence of characters. This is more like a utility class to work on that char sequence. This char sequence is maintained in following variable:

```
/** The value is used for character storage. */
private final char value[];
```

To access this array in different scenarios, following variables are used:

/** The offset is the first index of the storage that is used. */
private final int offset;
/** The count is the number of characters in the String. */
private final int count;
Whenever we create a substring from any existing string instance, substring() method only set's the new values of offset and count variables. The internal char array is unchanged. This is a possible source of memory leak if substring() method is used without care. Read more here

Explain the working of HashMap. How duplicate collision is resolved?

Most of you will agree that HashMap is most favorite topic for discussion in interviews now-a-days. If anybody asks me to describe "How HashMap works?", I simply answer: "On principles of Hashing". As simple as it is.

Now, Hashing in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/ algorithm on its properties.

A map by definition is : "An object that maps keys to values". Very easy.. right? So, HashMap has an inner class Entry, which looks like this:

```
static class Entry implements Map.Entry
{
final K key;
V value;
Entry next;
final int hash;
…//More code goes here
}
```

When, someone tries to store a key value pair in a HashMap, following things happen:

-- First of all, key object is checked for null. If key is null, value is stored in table[0] position. Because hash code for null is always 0.

-- Then on next step, a hash value is calculated using key's hash code by calling its hashCode() method. This hash value is used to calculate index in array for storing Entry object. JDK designers well assumed that there might be some poorly written hashCode() functions that can return very high or low hash code value. To solve this issue, they introduced another hash() function, and passed the object's hash code to this hash() function to bring hash value in range of array index size.

-- Now indexFor(hash, table.length) function is called to calculate exact index position for storing the Entry object.

-- Here comes the main part. Now, as we know that two unequal objects can have same hash code value, how two different objects will be stored in same array location [called bucket].
Answer is LinkedList. If you remember, Entry class had an attribute "next". This attribute always points to next object in chain. This is exactly the behavior of LinkedList.

So, in case of collision, Entry objects are stored in LinkedList form. When an Entry object needs to be stored in particular index, HashMap checks whether there is already an entry?? If there is no entry already present, Entry object is stored in this location.

If there is already an object sitting on calculated index, its next attribute is checked. If it is null, and current Entry object becomes next node in LinkedList. If next variable is not null, procedure is followed until next is evaluated as null.

What if we add the another value object with same key as entered before. Logically, it should replace the old value. How it is done? Well, after determining the index position of Entry object, while iterating over LinkedList on calculated index, HashMap calls equals method on key object for each Entry object. All these Entry objects in LinkedList will have similar hash code but equals() method will test for true equality. If key.equals(k) will be true then both keys are treated as same key object. This will cause the replacing of value object inside Entry object only.

In this way, HashMap ensure the uniqueness of keys.

Difference between interfaces and abstract classes?

This is very common question if you are appearing interview for junior level programmer. Well, most noticeable differences are as below:

- Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
- Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.
- Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected.
- Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
- A Java class can implement multiple interfaces but it can extend only one abstract class.
- Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
- Abstract class are slightly faster than interface because interface involves a search before calling any overridden method in Java. This is not a significant difference in most of cases but if you are writing a time critical application than you may not want to leave any stone unturned.

When do you override hashCode() and equals()?

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

hashCode() method is used to get a unique integer for given object. This integer is used for determining the bucket location, when this object needs to be stored in some HashTable like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.
equals() method, as name suggest, is used to simply verify the equality of two objects. Default implementation simply check the object references of two objects to verify their equality.

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

equals() must define an equality relation (it must be reflexive, symmetric, and transitive). In addition, it must be consistent (if the objects are not modified, then it must keep returning the same value). Furthermore, o.equals(null) must always return false.

hashCode() must also be consistent (if the object is not modified in terms of equals(), it must keep returning the same value).

The relation between the two methods is:

Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().

Why finalize() method should be avoided?

We all know the basic statement that finalize() method is called by garbage collector thread before reclaiming the memory allocated to the object. See this program which prove that finalize() invocation is not guaranteed at all. Other reasons can be:
1) finalize() methods do not work in chaining like constructors. It means like when you call a constructor then constructors of all super classes will be invokes implicitly. But, in case of finalize methods, this is not followed. Super class's finalize() should be called explicitly.

2) Any Exception thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact it will not be logged in your log files. So bad, isn't it?

3) Also, There is some performance penalty when finalize() in included in your class. In Effective java (2nd edition) Joshua bloch says,

"Oh, and one more thing: there is a severe performance penalty for using finalizers. On my machine, the time to create and destroy a simple object is about 5.6 ns.
Adding a finalizer increases the time to 2,400 ns. In other words, it is about 430 times slower to create and destroy objects with finalizers."

Why HashMap should not be used in multithreaded environment? Can it cause infinite loop as well?

We know that HashMap is non-synchronized collection where as its synchronized counter-part is HashTable. So, when you are accessing the collection in multithreaded environment and all threads are accessing a single instance of collection, then its safer to use HashTable for various obvious reasons e.g. to avoid dirty reads and to maintain data consistency. In worst case, this mutithreaded environment can result in infinite loop as well.

Yes, it is true. HashMap.get() can cause an infinite loop. Lets see how??

If you look at the source code HashMap.get(Object key) method, it looks like this:

```
public Object get(Object key) {

    Object k = maskNull(key);

    int hash = hash(k);

    int i = indexFor(hash, table.length);

    Entry e = table[i];

    while (true) {

        if (e == null)

            return e;

        if (e.hash == hash &amp;&amp; eq(k, e.key))

            return e.value;

        e = e.next;

    }

}
```

while(true) can always be a victim of infinite loop at runtime in multithreaded environment, IF, somehow e.next can point to itself. This will result in infinite loop. But, how e.next will point to itself (i.e. e).

This can happen in void transfer(Entry[] newTable) method, which is invoked at time the HashMap resizing is done.

```
do {

    Entry next = e.next;

    int i = indexFor(e.hash, newCapacity);

    e.next = newTable[i];

    newTable[i] = e;

    e = next;

} while (e != null);
```

This piece of code is prone to produce above condition, if resizing happen and at the same time other threads tried to modify the map instance.

Only way to avoid this scenario is to use synchronization in code, or better, used synchronized collection.

Explain abstraction and encapsulation? How are they related?

In simple words: “***Abstraction** captures only those details about an object that are relevant to the current perspective.*”
In object-oriented programming theory, abstraction involves the facility to define objects that represent abstract “actors” that can perform work, report on and change their state, and “communicate” with other objects in the system.
Abstraction in any programming language works in many ways. It can be seen from creating subroutines to defining interfaces for making low level language calls. Some abstractions try to limit the breadth of concepts a programmer needs, by completely hiding the abstractions they in turn are built on, e.g. design patterns.

Typically abstraction can be seen in two ways:

Data abstraction is the way to create complex data types and exposing only meaningful operations to interact with data type, where as hiding all the implementation details from

outside works. **Control abstraction** is the process of identifying all such statements and expose them as a unit of work. We normally use this feature when we create a function to perform any work.
Wrapping data and methods within classes in combination with implementation hiding (through access control) is often called encapsulation. The result is a data type with characteristics and behaviors. Encapsulation essentially has both i.e. information hiding and implementation hiding.
"***Whatever changes, encapsulate it***". It has been quoted as a famous design principle. For that matter in any class, changes can happen in data in runtime and changes in implementation can happen in future releases. So, encapsulation applies to both i.e. data as well as implementation.
SO, they can relate like following :

-- Abstraction is more about 'What' a class can do. [Idea] -- Encapsulation is more about 'How' to achieve that functionality. [Implementation]

Difference between interfaces and abstract classes?

Basic differences can be counted as follows:

- An interface cannot implement any methods, whereas an abstract class can
- A class can implement many interfaces but can have only one superclass (abstract or not)
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

You should remember that : "When you can fully describe the concept in terms of "***what it does***" without needing to specify any of "***how it does***", then you should use an interface. If you need to include some implementation details, then you will need to represent your concept in an abstract class."
Also, if i talk differently : Are there many classes that can be "*grouped together*" and described by one noun? If so, have an abstract class by the name of this noun, and inherit the classes from it. For example Cat and Dog can both inherit from abstract class Animal, and this abstract base class will implement a method void Breathe() which all animals will thus do in exactly the same fashion.
What kinds of verbs can be applied to my class, that might in general also be applied to others? Create an interface for each of these verbs. For example, All animals can be

fed, so I will create an interface called IFeedable and have Animal implement that. Only Dog and Horse are nice enough though to implement ILikeable, but some are not.

As said by someone: the main difference is where you want your implementation. By creating an interface, you can move your implementation to any class that implements your interface. By creating an abstract class, you can share implementation for all derived classes in one central place, and avoid lots of bad things like code duplication.

How StringBuffer save the memory?

A String is implemented as an immutable object; that is, when you initially decide to put something into a String object, the JVM allocates a fixed-width array of exactly the size of your initial value. This is then treated as a constant inside the JVM, which allows for very significant performance savings in the case where the String’s value is not changed. However, if you decide to change the String’s contents in any way, what the JVM then essentially does is copy the contents of the original String into a temporary space, make your changes, then save those changes into a whole new memory array. Thus, making changes to a String’s value after initialization is a fairly expensive operation.

StringBuffer, on the other hand, is implemented as a dynamically -- growable array inside the JVM, which means that any change operation can occur on the existing memory location, with new memory allocated only as-needed. However, there is no opportunity for the JVM to make optimizations around the StringBuffer, since its contents are assumed to be changeable at any instance.

Why wait and notify is declared in Object class instead of Thread?

Wait , notify , notifyAll methods are only required when you want your threads to access a shared resource and a shared resource could be any java object which is on the heap. So, these methods are defined on the core Object class so that each object has control of allowing Threads to wait on it’s monitor. Java doesn’t have any special object which is used for sharing a common resource. No such data structure is defined.So, onus is given on the Object class to be able to become shared resource providing it will helper methods like wait(),notify() and notifyAll().

Java is based on Hoare's monitors idea. In Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:

-- a Thread
-- a monitor (any object)

In the Java design, the thread can not be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on). This is a good design, because if we could make any other thread to wait on a desired monitor, this would lead to an "intrusion", posing difficulties on designing /programming concurrent programs. Remember that in Java all operations that are intrusive in another thread's execution are deprecated (e.g. stop()).

Write Java program to create deadlock in Java and fix it ?

In java, a deadlock is a situation where minimum two threads are holding lock on some different resource, and both are waiting for other resource to complete its task. And, none is able to leave the lock on resource it is holding.

To create a deadlock situation, and to know the solution : read full post "Writing a deadlock and resolving in java".

What happens if your Serializable class contains a member which is not serializable? How do you fix it?

In this case, **NotSerializableException** will be thrown at runtime. To fix this issue, a very simple solution is to mark such fields transient. It means these fields will not be serialized. If you want to save the state of these fields as well then you should consider reference variables which already implements serializable interface.

You also might need to use readResolve() and writeResolve() methods. Lets summarize this:

- First, make your non-serialisable field transient.
- In writeObject, first call defaultWriteObject on the stream to store all the non-transient fields, then call other methods to serialise the individual properties of your non-serialisable object.
- In readObject, first call defaultReadObject on the stream to read back all the non-transient fields, then call other methods (corresponding to the ones you added to writeObject) to deserialise your non-serialisable object.

Also, i will highly recommend to read **full guide on serialization in java**.

Explain transient and volatile keywords in java?

"*The* ***transient*** *keyword in Java is used to indicate that a field should not be serialized.*"
According to language specification: Variables may be marked transient to indicate that they are not part of the persistent state of an object. For example, you may have fields that are derived from other fields, and should only be done so programmatically, rather than having the state be persisted via serialization.
For example, in class BankPayment.java fields like principal and rate can be serialized while interest can be calculated any time even after de-serialization.

If we recall, each thread in java has its own local memory space as well and it does all read/write operations in its local memory. Once all operations are done, it write back the modified state of variable in main memory from where all threads access this variable. Normally, this is the default flow inside JVM. But, the volatile modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. It means every time thread want to read the state of variable, it must flush its local memory state and update the variable from main memory.

Volatile is most useful in lock-free algorithms. You mark the variable holding shared data as volatile when you are not using locking to access that variable and you want changes made by one thread to be visible in another, or you want to create a "happens-after" relation to ensure that computation is not re-ordered, again, to ensure changes become visible at the appropriate time.

The volatile should be used to safely publish immutable objects in a multi-threaded Environment. Declaring a field like public volatile ImmutableObject foo secures that all threads always see the currently available instance reference.

Difference between Iterator and ListIterator?

We can use Iterator to traverse a Set or a List or a Map. But ListIterator can only be used to traverse a List only. Other differences can be listed as below.

You can

1. iterate backwards.
2. obtain the index at any point.
3. add a new value at any point.
4. set a new value at that point

Deep copy and shallow copy?

A clone is an exact copy of the original. In java, it essentially means the ability to create an object with similar state as the original object. The clone() method provides this functionality.

Shallow copies duplicate as little as possible. By default, java cloning is shallow copy or ‘field by field copy’ i.e. as the Object class does not have idea about the structure of class on which clone() method will be invoked. So, JVM when called for cloning, do following things:

1) If the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned.

2) If the class contains members of any class type then only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated. Here, we want a clone which is independent of original and making changes in clone should not affect original.

Deep cloning requires satisfaction of following rules.

1. No need to separately copy primitives.
2. All the member classes in original class should support cloning and in clone method of original class in context should call super.clone() on all member classes.
3. If any member class does not support cloning then in clone method, one must create a new instance of that member class and copy all its attributes one by one to new member class object. This new member class object will be set in cloned object.

Read more about cloning here.

What is synchronization? Object level locking and class level locking?

Synchronization refers to multi-threading. A synchronized block of code can only be executed by one thread at a time. Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch.
Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method's object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.
Synchronization in java is achieved using synchronized keyword. You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.

Object level locking is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.
Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one

of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

Read more about synchronization here.

Difference between sleep() and wait()?

sleep() is a method which is used to hold the process for few seconds or the time you wanted but in case of wait() method thread goes in waiting state and it won't come back automatically until we call the notify() or notifyAll().

The major difference is that wait() releases the lock or monitor while sleep() doesn't releases any lock or monitor while waiting. Wait is used for inter-thread communication while sleep is used to introduce pause on execution, generally.

Thread.sleep() sends the current thread into the "Not Runnable" state for some amount of time. The thread keeps the monitors it has aquired — i.e. if the thread is currently in a synchronized block or method no other thread can enter this block or method. If another thread calls t.interrupt() it will wake up the sleeping thread. Note that sleep is a static method, which means that it always affects the current thread (the one that is executing the sleep method). A common mistake is to call t.sleep() where t is a different thread; even then, it is the current thread that will sleep, not the t thread.

object.wait() sends the current thread into the "Not Runnable" state, like sleep(), but with a twist. Wait is called on a object, not a thread; we call this object the "lock object." Before lock.wait() is called, the current thread must synchronize on the lock object; wait() then releases this lock, and adds the thread to the "wait list" associated with the lock. Later, another thread can synchronize on the same lock object and call lock.notify(). This wakes up the original, waiting thread. Basically, wait()/notify() is like sleep()/interrupt(), only the active thread does not need a direct pointer to the sleeping thread, but only to the shared lock object.

Read the difference in detail here.

Can you assign null to this reference variable?

NO. You can't. In java, left hand side of an assignment statement must be a variable. 'this' is a special keyword which represent the current instance always. This is not any variable.

Similarly, null can not be assigned to ‘super’ or any such keyword for that matter.

What if the difference between && and &??

& is bitwise and && is logical.

- & evaluates both sides of the operation.
- && evaluates the left side of the operation, if it’s true, it continues and evaluates the right side.

Read here for deep understanding.

How to override equals and hashCode() methods?

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

hashCode() method is used to get a unique integer for given object. This integer is used for determining the bucket location, when this object needs to be stored in some HashTable like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.

equals() method, as name suggest, is used to simply verify the equality of two objects. Default implementation simply check the object references of two objects to verify their equality.

Below are the important points to keep remember while overriding these functions.

1. Always use same attributes of an object to generate hashCode() and equals() both. As in our case, we have used employee id.
2. equals() must be consistent (if the objects are not modified, then it must keep returning the same value).
3. Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().
4. If you override one, then you should override the other.

Read more interesting facts and how to guide here.

Explain all access modifiers?

Java classes, fields, constructors and methods can have one of four different access modifiers:

private
If a method or variable is marked as private, then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.
If a class is marked as private then no external class an access the class. This doesn't really make so much sense for classes though. Therefore, the access modifier private is mostly used for fields, constructors and methods.

default
The default access level is declared by not writing any access modifier at all. Default access levels means that code inside the class itself + code inside classes in the same package as this class, can access the class, field, constructor or method. Therefore, the default access modifier is also sometimes called a package access modifier.

Subclasses cannot access methods and member variables in the superclass, if they have default accessibility declared, unless the subclass is located in the same package as the superclass.

protected
The protected acces modifier does the same as the default access, except subclasses can also access protected methods and member variables of the superclass. This is true even if the subclass is not located in the same package as the superclass.

public
The public access modifier means that all code can access the class, field, constructor or method, regardless of where the accessing code is located.

Modifiers	*Same Class*	*Same Package*	*Subclass*	*Other pa*
public	Y	Y	Y	Y

protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

What is garbage collection? Can we enforce it?

Garbage collection is an automatic memory management feature in many modern programming languages, such as Java and languages in the .NET framework. Languages that use garbage collection are often interpreted or run within a virtual machine like the JVM. In each case, the environment that runs the code is also responsible for garbage collection. A GC has two goals: any unused memory should be freed, and no memory should be freed unless the program will not use it anymore.

Can you force garbage collection?? Nope, System.gc() is as close as you can get. Your best option is to call System.gc() which simply is a hint to the garbage collector that you want it to do a collection. There is no way to force and immediate collection though as the garbage collector is non-deterministic. Also, under the documentation for OutOfMemoryError it declares that it will not be thrown unless the VM has failed to reclaim memory following a full garbage collection. So if you keep allocating memory until you get the error, you will have already forced a full garbage collection.

Read more about garbage collection here.

What is native keyword? Explain in detail?

The native keyword is applied to a method to indicate that the method is implemented in native code using JNI. It marks a method, that it will be implemented in other languages, not in Java.

Native methods were used in the past to write performance critical sections but with Java getting faster this is now less common. Native methods are currently needed when

- You need to call a library from Java that is written in other language.

- You need to access system or hardware resources that are only reachable from the other language (typically C). Actually, many system functions that interact with real computer (disk and network IO, for instance) can only do this because they call native code.

The downsides of using native code libraries are also significant:

1. JNI / JNA have a tendency to destabilize the JVM, especially if you try to do something complicated. If your native code gets native code memory management wrong, there's a chance that you will crash the JVM. If your native code is non-reentrant and gets called from more than one Java thread, bad things will happen … sporadically. And so on.
2. Java with native code is harder to debug than pure Java or pure C/C++.
3. Native code can introduce significant platform dependencies / issues for an otherwise platform independent Java app.
4. Native code requires a separate build framework, and that may have platform / portability issues as well.

What is serialization? Explain the catches?

In computer science, in the context of data storage and transmission, serialization is the process of translating data structures or object state into a format that can be stored and "resurrected" later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.

Java provides automatic serialization which requires that the object be marked by implementing the java.io.Serializable interface. Implementing the interface marks the class as "okay to serialize," and Java then handles serialization internally. There are no serialization methods defined on the Serializable interface, but a serializable class can optionally define methods with certain special names and signatures that if defined, will be called as part of the serialization/deserialization process.

Once an object is serialized, changes in its class break the de-serialization process. To identify the future changes in your class which will be compatible and others which will prove incompatible, please read the **full guide here**. In short, I am listing down here:

Incompatible changes

- Deleting fields

- Moving classes up or down the hierarchy
- Changing a non-static field to static or a non-transient field to transient
- Changing the declared type of a primitive field
- Changing the writeObject or readObject method so that it no longer writes or reads the default field data
- Changing a class from Serializable to Externalizable or vice-versa
- Changing a class from a non-enum type to an enum type or vice versa
- Removing either Serializable or Externalizable
- Adding the writeReplace or readResolve method to a class

Compatible changes

- Adding fields
- Adding/ Removing classes
- Adding writeObject/readObject methods [defaultReadObject or defaultWriteObject should be called first]
- Removing writeObject/readObject methods
- Adding java.io.Serializable
- Changing the access to a field
- Changing a field from static to non-static or transient to non transient