**Core Java Interview Questions**

Table of Contents

[1. What’s the difference between Association, Aggregation and Composition? 3](#_Toc486229571)

[2. What’s polymorphism in Java? 7](#_Toc486229572)

[3. What are advantages of JMS? 8](#_Toc486229573)

[4. What’s Abstraction? 9](#_Toc486229574)

[5. Difference between Abstract class and interface? 11](#_Toc486229575)

[6. What’s Encapsulation? 11](#_Toc486229576)

[7. Difference between Domain model and Data model? 11](#_Toc486229577)

[*1.* *Domain Model* 11](#_Toc486229578)

[*2. Data Model* 11](#_Toc486229579)

[8. What is Micro services Architecture? 12](#_Toc486229580)

[*Advantages:* 13](#_Toc486229581)

[*Disadvantages:* 13](#_Toc486229582)

[9. What are the methods in Object class? 13](#_Toc486229583)

[10. How cloning method works in Java? 14](#_Toc486229584)

[11. Difference between Shallow and Deep Copy in Java? 23](#_Toc486229585)

[12. How garbage collection works internally in Java? 24](#_Toc486229586)

[13. JVM (java virtual machine) in detail in java? 42](#_Toc486229587)

[14. How to override equals and hashCode in Java? 50](#_Toc486229588)

[Usage of hashCode() and equals() 50](#_Toc486229589)

[Overriding the default behavior 50](#_Toc486229590)

[Overriding hashCode() and equals() using Apache Commons Lang 53](#_Toc486229591)

[Important things to remember 55](#_Toc486229592)

[Special Attention When Using in *ORM* 55](#_Toc486229593)

[15. What is Polymorphism in Java? 56](#_Toc486229594)

[1. What is Polymorphism? 56](#_Toc486229595)

[2. How is Polymorphism Implemented in Java? 56](#_Toc486229596)

[3. Why is Polymorphism? 58](#_Toc486229597)

[16. What is Encapsulation in Java? 59](#_Toc486229598)

[1. What is Encapsulation? 59](#_Toc486229599)

[2. Why Is Encapsulation? 62](#_Toc486229600)

[3. How Is Encapsulation Done in Java? 63](#_Toc486229601)

[4. Encapsulation vs. Abstraction 63](#_Toc486229602)

[17. Difference between Loose Coupling and Tight Coupling in Java? 64](#_Toc486229603)

[1. Tight Coupling 64](#_Toc486229604)

[2. Loose Coupling 66](#_Toc486229605)

[Advantages of Loose coupling 67](#_Toc486229606)

# What’s the difference between Association, Aggregation and Composition?

* **Association**
  + In Object-oriented programming, one object is related to other to use functionality and service provided by that object.
  + This relationship between two object is known as association
  + Both Composition and Aggregation are form of association between two objects, but there is subtle difference between composition and aggregation
  + In UML notation, association is denoted by a simple arrow
* **Composition**
  + Association between two objects as composition, when one class owns other class and other class cannot meaningfully exists, when it's owner destroyed
  + For example Human class is composition of several body parts including Hand, Leg and Heart.
  + When human object dies, all it's body part ceased to exist meaningfully, this is one example of Composition
  + Composition is also very much preferred in object oriented design over inheritance
  + Composition represent strongest form of relationship and association being the most general form.
  + We can use final keyword to represent Composition.
  + Since in Composition, Owner object expect part object to be available and functions, by making it final, your provide guarantee that, when Owner will be created, this part object will exist
  + If one object is part-of another object e.g. Engine is part of Car, then association or relationship between them is Composition.
  + In UML notation, composition is denoted by a filled diamond
* **Example**

public class Car {

private String make;

private int year;

private final Engine engine;

public Car(String make, int year, int engineCapacity, int engineSerialNumber) {

this.make=make;

this.year=year;

// we create the engine using parameters passed in Car constructor

// only the Car instance has access to the engine instance

// when Car instance is garbage collected, the engine instance is garbage collected too

engine = new Engine(engineCapacity, engineSerialNumber);

}

public String getMake() {

return make;

}

public int getYear() {

return year;

}

public int getEngineSerialNumber() {

return engine.getEngineSerialNumber();

}

public int getEngineCapacity() {

return engine.getEngineCapacity();

}

}

public class Engine {

private int engineCapacity;

private int engineSerialNumber;

public Engine(int engineCapacity, int engineSerialNumber) {

this.engineCapacity = engineCapacity;

this.engineSerialNumber = engineSerialNumber;

}

public int getEngineCapacity() {

return engineCapacity;

}

public int getEngineSerialNumber() {

return engineSerialNumber;

}

}

* **Aggregation**
  + While an association is known as aggregation when one object uses other object.
  + While in case of Aggregation, including object can exists without being part of main object
  + For example a Player which is part of a Team, can exists without team and can become part of other teams as well
  + Aggregation is a lighter form of Composition, where sub-part object can meaningfully exits without main objects.
  + On the other hand if one object just has another object e.g. Car has driver than it's Aggregation.
  + In UML notation, aggregation is denoted by an empty diamond
  + Example

public class Car {

private String make;

private int year;

private Engine engine;

public Car(String make, int year, Engine engine) {

this.make = make;

this.year = year;

// the engine object is created outside and is passed as argument to Car constructor

// When this Car object is destroyed, the engine is still available to objects other than Car

// If the instance of Car is garbage collected the associated instance of Engine may not be garbage collected (if it is still referenced by other objects)

this.engine = engine;

}

public String getMake() {

return make;

}

public int getYear() {

return year;

}

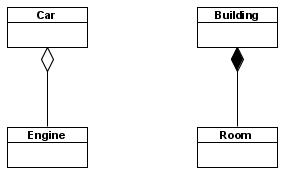
public Engine getEngine() {

return engine;

}

}

Class Diagram:



# What’s polymorphism in Java?

* + Polymorphism is the ability to create a variable, a function, or an object that has more than one form.
  + Polymorphism is the ability by which, we can create functions or reference variables, which behaves differently in different programmatic context.
  + Polymorphism is one of the major building blocks of object oriented programming along with inheritance, abstraction and encapsulation.
  + An example of polymorphism is referring the instance of subclass, with reference variable of super-class

Object o = new Object(); //o can hold the reference of any subtype

Object o = new String();

Object o = new Integer();

* + Here, String is subclass of Object class. This is basic example of polymorphism.
  + In java language, polymorphism is essentially considered into two versions.
    - Compile time polymorphism (static binding or method overloading)
    - Runtime polymorphism (dynamic binding or method overriding)
  + Compile time polymorphism (static binding or method overloading)
    - This is used to write the program in such a way, that flow of control is decided in compile time itself.
    - It is achieved using method overloading.
  + In method overloading, an object can have two or more methods with same name, BUT, with their method parameters different.
  + These parameters may be different on two bases:
    - Parameter type
    - Parameter count

# What are advantages of JMS?

* + Asynchronous communications :
    - An application need to notify another that an event has occurred with no need to wait for a response.
  + Reliability:
    - Ensure once-and-only-once message delivery. With your DB approach you have to "reinvent the wheel", specially if you have several clients reading the messages.
  + Loose coupling:
    - Not all systems can communicate using a database. So JMS is pretty good to be used in heterogeneous environments with decoupled systems that can communicate over system boundaries.
    - It is an implementation to handle the [Producer-consumer problem](https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem).

# What’s Abstraction?

* Abstraction is a way to segregate implementation from an interface
* Abstraction in Java is achieved by using interface and abstract class in Java.
* An interface or abstract class is something which is not concrete, something which is incomplete.
* In order to use interface or abstract class, we need to extend and implement an abstract method with concrete behavior.
* One example of Abstraction is creating interface to denote common behavior without specifying any details about how that behavior works e.g. You create an interface called Server which has the start() and stop() method.
* This is called abstraction of Server because every server should have a way to start and stop and details may differ.
* **Abstraction Using abstract class**
  + Java has a concept of abstract classes, abstract method but a variable cannot be abstract in Java.
  + An abstract method in Java doesn't have the body, it's just a declaration. In order to use an abstract method, you need to override that method in sub class.
  + When you know something needs to be there but not sure how exactly it should look like. E.g. when I am creating a class called Vehicle, I know there should be methods like start () and stop () but don't know how that start and stop method should work, because every vehicle can have different start and stop mechanism e.g. some can be started by kicking or some can be by pressing buttons
  + So the implementation of those start() and stop() methods should be left to their concrete implementation e.g. Scooter, MotorBike , Car etc
  + In Java, you cannot create an instance of the abstract class using the new operator, its compiler error. Though abstract class can have a constructor.
  + Abstract is a keyword in Java, which can be used with both class and method.
  + A class automatically becomes abstract class when any of its methods declared as abstract.
  + If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. Alternatively, this class can also be abstract.
* **Abstraction Using Interface**
  + In Java Interface is an another way of providing abstraction, Interfaces are by default abstract and only contains public, static, final constant or abstract methods.
  + It's very common interview question is that where should we use abstract class and where should we use Java Interfaces in my view this is important to understand to design better Java application, you can go for java interface if you only know the name of methods your class should have e.g. for Server it should have start () and stop () method but we don't know how exactly these start and stop method will work.
  + If you know some of the behavior while designing class and that would remain common across all subclasses add that into an abstract class.
  + An interface like Runnable interface is a good example of abstraction in Java which is used to abstract task executed by multiple threads. Callable is another good abstract of a task which can return value

# Difference between Abstract class and interface?

* Abstraction is a way to segregate implementation from an interface

# What’s Encapsulation?

* Abstraction is a way to segregate implementation from an interface

# Difference between Domain model and Data model?

## Domain Model

* + A domain model is generally implemented as an **object model** within a layer that uses a lower-level layer for persistence and "publishes" an API to a higher-level layer to gain access to the data and behavior of the model.
  + In the Unified Modeling Language (UML), a **class diagram** is used to represent the domain model.

## 2. Data Model

* + Data models define how the **logical structure** of a database is modeled. Data Models are fundamental entities to introduce abstraction in a DBMS. Data models define how data is **connected** to each other and how they are **processed and stored** inside the system.
  + The very first data model could be **flat data-models**, where all the data used are to be kept in the same plane. Earlier data models were not so scientific, hence they were prone to introduce lots of duplication and update anomalies

# What is Micro services Architecture?

* Micro services is a way of breaking large software projects into smaller, independent, and loosely coupled modules.
* Individual modules are responsible for highly defined and discrete tasks and communicate with other modules through simple, universally accessible APIs.
* The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.
* Microservices came about to help solve the frustrations developers were having with large applications that require change cycles to be tied together. In a monolithic application, any small change or update required building and deploying an entirely new application. This inevitably means any application development entails a certain amount of planning, preparation, time and, potentially, money.
* Microservices, on the other hand, require little of centralized management. Microservices applications are independently deployable and scalable. They enhance business capabilities with less planning and production than monoliths.
* In a microservices architecture, each service runs a unique process and usually manages its own database

## Advantages:

* + Are easily deployed.
  + Require less production time.
  + Can scale quickly.
  + Can be reused among different projects.
  + Work well with containers, such as Docker.
  + Complement cloud activities.

## Disadvantages:

* + Potentially too granular
  + Latency during heavy use
  + Testing can be complex

# What are the methods in Object class?

* equals (Object obj)
  + Checks whether the obj object is equal to the object on which the equals method is called .
* hashCode()
  + hashCode() is used for the HashTable . It returns the hash value of the object.
* getClass()
  + It returns the runtime class object .
* clone()
  + It creates and returns the copy of the object .
* notify()
  + It will wake up the thread waiting for the objects monitor.
* notifyAll()
  + It will wakes up all the thread that are waiting for the objects monitor .
* toString()
  + It will return the string representation of the object .
* wait()
  + This method causes the current thread to place itself in the wait set for this object and then to relinquish any and all synchronization claims on this object

# How cloning method works in Java?

* The clone() is a tricky method from java.lang.Object class, which is used to create a copy of an Object in Java.
* An object which is returned by the clone() method is known as a clone of original instance.
* A clone object should follow basic characteristics e.g. a.clone() != a, which means original and clone are two separate object in Java heap, a.clone().getClass() == a.getClass() and clone.equals(a), which means clone is exact copy of original object.
* java.lang.Object provides default implementation of clone() method in Java. It's declared as protected and native in Object class, so implemented in native code. Since its convention to return clone() of an object by calling super.clone() method, any cloning process eventually reaches to java.lang.Object clone() method.
* This method, first checks if the corresponding object implements Cloneable interface, which is a marker interface. If that instance doesn't implement Cloneable then it throws CloneNotSupportedException in Java, a checked exception, which is always required to be handled while cloning an object.
* If an object passes this check, than java.lang.Object's clone() method creates a shallow copy of the object and returned it to the caller.
* Since Object class' clone() method creates copy by creating new instance, and then copying field-by-field, similar to assignment operator, it's fine for primitives and Immutable object, but not suited if your class contains some mutable data structure e.g. Collection classes like ArrayList or arrays.
* In that case, both original object and copy of the object will point to the same object in the heap. You can prevent this by using the technique known as deep cloning, on which each mutable field is cloned separately. In short, here is how clone method works in Java:
* Any class calls clone() method on an instance, which implements Cloneable and overrides protected clone() method from Object class, to create a copy.
* Call to clone() method on Rectangle is delegated to super.clone(), which can be a custom superclass or by default java.lang.Object
* Eventually, call reaches to java.lang.Object's clone() method, which verify if the corresponding instance implements Cloneable interface, if not then it throws CloneNotSupportedException, otherwise it creates a field-by-field copy of the instance of that class and returned to the caller.
* Example

**import** **org.apache.log4j.Logger**;

/\*\*

  \* Simple example of overriding clone() method in Java to understand How Cloning of

  \* Object works in Java.

  \*

  \* @author

 \*/

**public** **class** **JavaCloneTest** {

**private** **static** **final** Logger logger = Logger.getLogger(JavaCloneTest.class);

**public** **static** **void** **main**(String args[]) {

        Rectangle rec = **new** Rectangle(**30**, **60**);

        logger.info(rec);

        Rectangle copy = **null**;

**try** {

            logger.info("Creating Copy of this object using Clone method");

            copy = rec.clone();

            logger.info("Copy " + copy);

        } **catch** (CloneNotSupportedException ex) {

            logger.debug("Cloning is not supported for this object");

        }

        //testing properties of object returned by clone method in Java

        logger.info("copy != rec : " + (copy != rec));

        logger.info("copy.getClass() == rec.getClass() : " + (copy.getClass() == rec.getClass()));

        logger.info("copy.equals(rec) : " + copy.equals(rec));

        //Updating fields in original object

        rec.setHeight(**100**);

        rec.setWidth(**45**);

        logger.info("Original object :" + rec);

        logger.info("Clonned object  :" + copy);

    }

}

**public** **class** **Rectangle** **implements** Cloneable{

**private** **int** width;

**private** **int** height;

**public** **Rectangle**(**int** w, **int** h){

        width = w;

        height = h;

    }

**public** **void** **setHeight**(**int** height) {

**this**.height = height;

    }

**public** **void** **setWidth**(**int** width) {

**this**.width = width;

    }

**public** **int** **area**(){

**return** widthheight;

    }

    @Override

**public** String **toString**(){

**return** String.format("Rectangle [width: %d, height: %d, area: %d]", width, height, area());

    }

    @Override

**protected** Rectangle **clone**() **throws** CloneNotSupportedException {

**return** (Rectangle) **super**.clone();

    }

    @Override

**public** **boolean** **equals**(Object obj) {

**if** (obj == **null**) {

**return** **false**;

        }

**if** (getClass() != obj.getClass()) {

**return** **false**;

        }

**final** Rectangle other = (Rectangle) obj;

**if** (**this**.width != other.width) {

**return** **false**;

        }

**if** (**this**.height != other.height) {

**return** **false**;

        }

**return** **true**;

    }

    @Override

**public** **int** **hashCode**() {

**int** hash = **7**;

        hash = **47**  hash + **this**.width;

        hash = **47**  hash + **this**.height;

**return** hash;

    }

}

*Output:*

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - Rectangle [*width:* **30**, *height:* **60**, *area:* **1800**]

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - Creating Copy of **this** object using Clone method

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - Copy Rectangle [*width:* **30**, *height:* **60**, *area:* **1800**]

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - copy != rec : **true**

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - copy.getClass() == rec.getClass() : **true**

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - copy.equals(rec) : **true**

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - Original object :Rectangle [*width:* **45**, *height:* **100**, *area:* **4500**]

**2013**-**05**-**20** **23**:**46**:**58**,**882** **0**    [main] INFO  JavaCloneTest  - Cloned object  :Rectangle [*width:* **30**, *height:* **60**, *area:* **1800**]

**Example for Deep Cloning with Mutable field**

* Since default implementation of clone() method only does shallow copy of objects, it can create issue, if original object contains mutable object or Collection classes.
* In our example, we have a class called Programmer, with String name, int age and List of Certifications.
* When we override clone() method inside Programmer class, we need to explicitly take care of this List, otherwise, both original and cloned object will point to same Collection in Java heap, which means, any change e.g. adding a new Certification in original object will also reflect in cloned object or vice-versa.
* Since an object should be independent of its clone, we need to fix this issue by applying deep cloning techniques.
* To fix this, we reassign certification fields of clone object by explicitly copying data, as shown in following line :

clone.certifications = new ArrayList(certifications); //deep copying

* **Example**

import java.util.ArrayList;

import java.util.List;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

/\*

\* Java program to show how to override clone method for deep copying.

\* This example includes a mutable filed in class to be cloned to show how you deal with

\* practical classes which contains both immutable and mutable fields.

\*

\*/

public class CloneTest {

private static final Logger logger = LoggerFactory.getLogger(Cloneclass);

public static void main(String args[]) {

Programmer javaguru = new Programmer("John", 31);

javaguru.addCertificates("OCPJP");

javaguru.addCertificates("OCMJD");

javaguru.addCertificates("PMP");

javaguru.addCertificates("CISM");

logger.debug("Real Java Guru : {}", javaguru);

Programmer clone = javaguru.clone();

logger.debug("Clone of Java Guru : {}", clone);

//let's add another certification to java guru

javaguru.addCertificates("Oracle DBA");

logger.debug("Real Java Guru : {}", javaguru);

logger.debug("Clone of Java Guru : {}", clone);

}

}

class Programmer implements Cloneable{

private static final Logger logger = LoggerFactory.getLogger(Programmer.class);

private String name;

private int age;

private List certifications ;

public Programmer(String name, int age) {

this.name = name;

this.age = age;

this.certifications = new ArrayList();

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void addCertificates(String certs){

certifications.add(certs);

}

@Override

public String toString() {

return String.format("%s, %d, Certifications: %s", name, age, certifications.toString());

}

@Override

protected Programmer clone() {

Programmer clone = null;

try{

clone = (Programmer) super.clone();

clone.certifications = new ArrayList(certifications); //deep copying

}catch(CloneNotSupportedException cns){

logger.error("Error while cloning programmer", cns);

}

return clone;

}

}

Output of Shallow copying :

[main] DEBUG CloneTest - Real Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM]

[main] DEBUG CloneTest - Clone of Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM]

[main] DEBUG CloneTest - Real Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM, Oracle DBA]

[main] DEBUG CloneTest - Clone of Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM, Oracle DBA]

After deep copying collection:

[main] DEBUG CloneTest - Real Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM]

[main] DEBUG CloneTest - Clone of Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM]

[main] DEBUG CloneTest - Real Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM, Oracle DBA]

[main] DEBUG CloneTest - Clone of Java Guru : John, 31, Certifications: [OCPJP, OCMJD, PMP, CISM]

* **Best Practices to follow while overriding clone method in Java**
  + Return Sub class from clone() method, instead of returning java.lang.Object
  + Use deep copy, if your class contains any mutable field.
  + Don't throw CloneNotSupportedException.

# Difference between Shallow and Deep Copy in Java?

* When we call Object.clone(), this method performs a shallow copy of object, by copying data field by field, and if we override this method and by convention first call super.clone(), and then modify some fields to "deep" copy, then we get deep copy of object. This modification is done to ensure that original and cloned object are independent to each other.
* In shallow copy main or parent object is copied, but they share same fields or children if fields are modified in one parent object other parent fields have automatic same changes occur, but in deep copy this is not the case.
* If our parent object contains only primitive value then shallow copy is good for making clone of any object because in new object value is copied but if parent object contains any other object then only reference value is copied in new parent object and both will point to same object so in that case according to our need we can go for deep copy.
* Deep copy is expensive as compare to shallow copy in terms of object creation, because it involves recursive copying of data from other mutable objects, which is part of original object.

# How garbage collection works internally in Java?

* All Java objects are always created on heap in java.
* What is GC (Garbage collection) process in java?
  + GC (Garbage collection) is the process by which JVM cleans objects (unused objects) from heap to reclaim heap space in java.
* What is Automatic Garbage Collection in JVM heap memory in java?
  + Automatic garbage collection is the process of Identifying objects which are in use in java heap memory and which objects are not in use in java heap memory and deleting the unused objects in java heap memory.
* How to identify objects which are in use in java heap memory?
  + Objects in use (or referenced objects) are those objects which is still needed by java program, some part of java program is still pointing to that object.
* Which objects are NOT in use in java heap memory?
  + Objects not in use (or unreferenced objects or unused objects) are those objects which is not needed by java program, no part of java program is pointing to that object. So, these unused objects can be cleaned in garbage collection process and memory used by an unreferenced object can be reclaimed.
* GC (Garbage collection) process automatically clears objects from heap to reclaim heap space. You just need to specify the type of garbage collector type you want to use at JVM startup.
* Gc (garbage collector) calls finalize method for garbage collection. Finalize method is called only once by garbage collector for an object in java.
* Daemon threads are low priority threads which runs intermittently in background for doing garbage collection (gc) in java.
* We can force early gc (garbage collection) in java by using following methods

System.gc();

Runtime.getRuntime().gc();

System.runFinalization();

Runtime.getRuntime().runFinalization();

* By calling these methods JVM runs the finalize () methods of any objects pending finalization i.e. objects which have been discarded but there finalize method is yet to be run. After finalize method is executed JVM reclaims space from all the discarded objects in java.
* Note: Calling these methods does not guarantee that it will immediately start performing garbage collection.
* Finalize method execution is not assured - We must not override finalize method to write some critical code of application because methods execution is not assured. Writing some critical code in finalize method and relying on it may make application to go horribly wrong in java.
* Dealing with OutOfMemoryError in java.
* WeakhashMap in java - java.util.WeakHashMap is hash table based implementation of the Map interface, with weak keys. An entry in a WeakHashMap will be automatically removed by garbage collector when its key is no longer in ordinary use.
* Object which is set explicitly set to null becomes eligible for gc (garbage collection) in java.
* Example 1

String s=”abc”; //s is currently not eligible for gc (garbage collection) in java.

s = null; //Now, s is currently eligible for gc (garbage collection) in java.

* Example 2

List list =new ArrayList(); //list is currently not eligible for gc (garbage collection).

list = null; //Now, list is currently eligible for gc (garbage collection).

* Difference in garbage collection in C/C++ and Java (Hint : In terms of memory allocation and deallocation of objects)?
  + In java garbage collection (memory allocation and deallocation of objects) is an automatic process.
  + But, In C and C++ memory allocation and deallocation of objects) is a manual process.
* All the variables declared inside block becomes eligible for gc (garbage collection) when we exit that block (As scope of those variable is only that block) in java.
* Example of garbage collection while using block in java -

**class** MyClass {

**public** **static** **void** main(String[] args) {

**boolean** var = **false**;

**if** (var) { // begin block 1

**int x = 1; // x is declared inside block**

                 //..........

                 //code inside block...

                 //..........

          } // end block 1 //And **now x is eligible for gc (garbage collection)**

**else** { // begin block 2

**int** y = 1;

                 //..........

                 //code inside block...

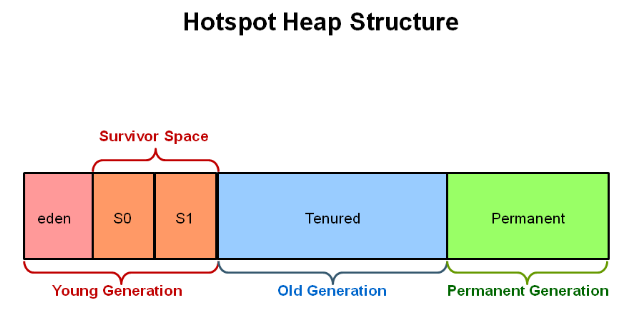
                 //..........

          } // end block 2 //And **now y is eligible for gc (garbage collection)**

   }

}

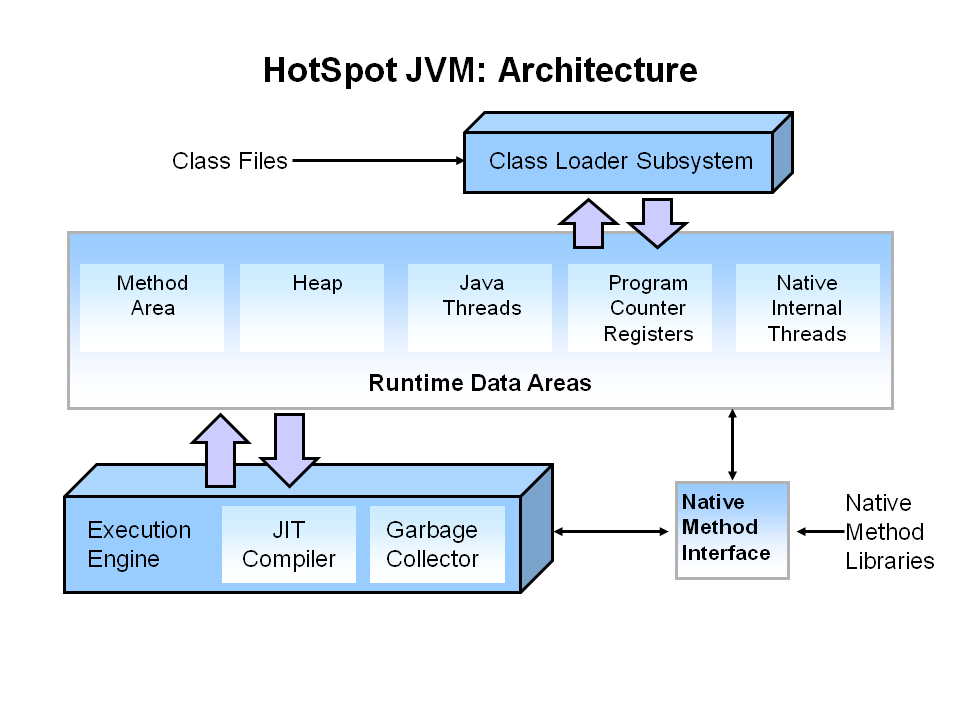
* **Terms frequently used in Garbage Collection (GC) in java**
* What is Throughput in gc(garbage collection) in java ?
  + In short, Throughput is the time not spent in garbage collection (GC) ( in percent).
  + Throughput focuses on maximizing the amount of work by an application in a specific period of time.
  + Examples of how throughput might be measured include
    - The number of transactions completed in a given time.
    - The number of jobs that a batch program can complete in an hour.
    - The number of database queries that can be completed in an hour.
* What are pauses in gc(garbage collection) in java ?
  + Pauses is applications pauses i.e. when application doesn’t gives any response because of garbage collection (GC).
* **JVM Heap memory with diagram in java**



* JVM Heap memory (Hotspot heap structure) in java consists of following elements
  + Young Generation
    - Eden,
    - S0 (Survivor space 0)
    - S1 (Survivor space 1)
  + Old Generation (Tenured)
  + Permanent Generation.
  + So, JVM Heap memory (Hotspot heap structure) is divided into three parts Young Generation, Old Generation (tenured) and Permanent Generation.
  + Young Generation is further divided into Eden, S0 (Survivor space 0) and S1 (Survivor space 1).
* **GARBAGE COLLECTION (Minor and major garbage collection) in JVM Heap memory (i.e. in young, old and permanent generation)**
* Young Generation (Minor garbage collection occurs in Young Generation)
  + New objects are allocated in Young generation.
  + Minor garbage collection occurs in Young Generation.
  + When minor garbage collection?
    - When the young generation fills up, this causes a minor garbage collection.
    - All the unreferenced (dead) objects are cleaned up from young generation.
  + When objects are moved from young to old generation in JVM heap?
    - Some of the objects which aren't cleaned up survive in young generation and gets aged. Eventually such objects are moved from young to old generation.
  + What is Stop the World Event?
    - Minor garbage collections are called Stop the World events.
    - All the non-daemon threads running in application are stopped during minor garbage collections (i.e. the application stops for while).
    - Daemon threads performs minor garbage collection. (Daemon threads are low priority threads which runs intermittently in background for doing garbage collection).
* Old Generation or (tenured generation) - (Major garbage collection occurs in Old Generation)
  + The Old Generation is used for storing the long surviving aged objects (Some of the objects which aren't cleaned up survive in young generation and gets aged. Eventually such objects are moved from young to old generation).
  + Major garbage collection occurs in Old Generation.
  + At what time (or what age) objects are moved from young to old generation in JVM heap?
    - There is some threshold set for young generation object and when that age is met, the object gets moved to the old generation during gc(garbage collection) in java.
  + What is major garbage collection in java?
    - When the old generation fills up, this causes a major garbage collection. Objects are cleaned up from old generation.
    - Major collection is much slower than minor garbage collection in jvm heap because it involves all live objects.
  + Major gc(garbage collections) in responsive applications in java?
    - Major garbage collections should be minimized for responsive applications because applications must not be stopped for long.
  + Optimizing Major gc(garbage collections) in responsive applications in java?
    - Selection of appropriate garbage collector for the old generation space affects the length of the “Stop the World” event for a major garbage collection.
* Permanent Generation or (Permgen) - (full garbage collection occurs in permanent generation in java).
  + Permanent generation Space contains metadata required by JVM to describe the classes and methods used in the application.
  + The permanent generation is included in a full garbage collection in java.
  + The permanent generation space is populated at runtime by JVM based on classes in use in the application.
  + The permanent generation space also contains Java SE library classes and methods in java.
  + JVM garbage collects those classes when classes are no longer required and space may be needed for other classes in java.
* **Most important VM (JVM) PARAMETERS in JVM Heap memory**
* -Xms : Xms is minimum heap size which is allocated at initialization of JVM.
  + Example: java -Xms512m MyJavaProgram
  + It will set the minimum heap size of JVM to 512 megabytes.
* -Xmx : Xmx is the maximum heap size that JVM can use.
  + Example: java -Xmx512m MyJavaProgram
  + It will set the maximum heap size of JVM to 512 megabytes.
* Young Generation(VM PARAMETERS for Young Generation)
  + -Xmn : -Xmn sets the size of young generation.
    - java -Xmn512m MyJavaProgram
  + -XX:NewRatio : NewRatio controls the size of young generation.
    - -XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3.
    - In other words, the combined size of the eden and survivor spaces will be one fourth of the total heap size.
  + -XX:NewSize - NewSize is minimum size of young generation which is allocated at initialization of JVM.
    - Note : If you have specified -XX:NewRatio than minimum size of the young generation is allocated automatically at initialization of JVM.
  + -XX:MaxNewSize - MaxNewSize is the maximum size of young generation that JVM can use.
  + -XX:SurvivorRatio : (for survivor space)
    - SurvivorRatio can be used to tune the size of the survivor spaces, but this is often not as important for performance
* Old Generation (tenured) - (VM PARAMETERS for Old Generation)
  + -XX:NewRatio : NewRatio controls the size of young and old generation.
* Permanent Generation (VM PARAMETERS for Permanent Generation)
  + -XX:PermSize: It’s is initial value of Permanent Space which is allocated at startup of JVM.
* **Different Garbage collectors in detail**
* **Serial collector / Serial GC**
  + Serial collector is also called Serial GC (Garbage collector) in java.
  + Serial GC (Garbage collector) is rarely used in java
  + Serial GC (Garbage collector) is designed for the single threaded environments in java.
  + In Serial GC (Garbage collector) , both minor and major garbage collections are done serially by one thread (using a single virtual CPU) in java.
  + Serial GC (Garbage collector) uses a mark-compact collection method. This method moves older memory to the beginning of the heap so that new memory allocations are made into a single continuous chunk of memory at the end of the heap. This compacting of memory makes it faster to allocate new chunks of memory to the heap in java.
  + The serial garbage collector is the default for client style machines in Java SE 5 and 6**.**
  + When to Use the Serial GC (garbage Collector) in java ?
    - The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work in java.
    - Serial GC (garbage collector) is also popular in environments where a high number of JVMs are run on the same machine. In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs in java.
  + Vm (JVM) option for enabling serial GC (garbage Collector) in java?
    - -XX:+UseSerialGC
    - Example of Passing Serial GC in Command Line for starting jar
    - java -Xms256m -Xmx512m -XX:+UseSerialGC -jar d:\MyJar.jar
* **Throughput GC (Garbage collector) or Parallel collector in java**
  + Throughput garbage collector is the default garbage collector for JVM in java.
  + Throughput garbage collector uses multiple threads to execute a minor collection and so reduces the serial execution time of the application in java.
  + The throughput garbage collector is similar to the serial garbage collector but uses multiple threads to do the minor collection in java.
  + This garbage collector uses a parallel version of the young generation garbage collector in java.
  + The tenured generation collector is the same as the serial garbage collector in java.
  + When to Use the Throughput GC (Garbage collector) in java ?
    - The Throughput garbage collector should be used when application can afford low pauses in java.
    - And application is running on host with multiple CPU’s (to derive advantage of using multiple threads for garbage collection) in java.
  + Vm (JVM) option for enabling throughput GC (Garbage collector) in java?
    - -XX:+UseParallelGC
    - Example of using throughput collector in Command Line for starting jar
    - java -Xms256m -Xms512m -XX:+UseParallelGC -jar d:\MyJar.jar
  + With this Vm (JVM) option you get a
    - Multi-threaded young generation garbage collector in java,
    - single-threaded old generation garbage collector in java and
    - Single-threaded compaction of old generation in java.
  + Vm (JVM) option for enabling throughput collector with n number of threads in java?
    - -XX:ParallelGCThreads=<numberOfThreads>
* **Incremental low pause garbage collector (train low pause garbage collector) in java**
  + Incremental low pause garbage collector is not used these days in java. Incremental low pause collector was used in Java 4.
* **Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java**
  + Concurrent Mark Sweep (CMS) collector collects the old/tenured generation in java.
  + Concurrent Mark Sweep (CMS) Collector minimize the pauses by doing most of the garbage collection work concurrently with the application threads in java.
  + Concurrent Mark Sweep (CMS) Collector on live objects?
    - Concurrent Mark Sweep (CMS) Collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap in java.
  + When to Use the Concurrent Low Pause Collector in java?
    - Concurrent Low Pause Collector should be used if your applications that require low garbage collection pause times in java.
    - Concurrent Low Pause Collector should be used when your application can afford to share processor resources with the garbage collector while the application is running in java.
    - Concurrent Low Pause Collector is beneficial to applications which have a relatively large set of long-lived data (a large tenured generation) and run on machines with two or more processors in java
  + Examples when to use Concurrent Mark Sweep (CMS) collector / concurrent low pause collector should be used for ?
    - Example 1 - Desktop UI application that respond to events,
    - Example 2 - Web server responding to a request and
    - Example 3 - Database responding to queries.
  + Vm (JVM) option for enabling Concurrent Mark Sweep (CMS) Collector in java?
    - -XX:+UseConcMarkSweepGC
    - Example of using Concurrent Mark Sweep (CMS) collector / concurrent low pause collector in Command Line for starting jar
    - java -Xms256m -Xms512m -XX:+UseConcMarkSweepGC -jar d:\MyJar.jar
  + Heap Structure for CMS garbage Collector?
    - CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.
    - Young Generation is further divided into Eden, S0 (Survivor space 0) and S1 (Survivor space 1).
  + Detailed Steps in GC (garbage collection) cycle in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java?
    - **Young Generation GC (garbage Collection) in java**
      * Live objects are copied from the Eden space and survivor space to the other survivor space.
      * Any older objects that have reached their aging threshold are promoted to old generation
    - **After Young generation GC (garbage Collection) in java**
      * After a young GC, the Eden space and one of the survivor spaces is cleared.
      * Promoted objects (older objects that have reached their aging threshold in young GC) are available in old generation.
    - **Old Generation GC (garbage Collection) with CMS in java**
      * Initial mark phase - (First pause happens/ stop the world event ) - mark live/reachable objects (Example - objects on thread stack, static objects etc.) and elsewhere in the heap (Example - the young generation).
      * Concurrent marking phase - (No pause phase ) - finds live objects while the application continues to execute.
      * Remark - (Second pause happens/ stop the world events) - It finds objects that were missed during the concurrent marking phase due to the concurrent execution of the application threads.
      * Sweep phase - do the concurrent sweep, memory is freed up.
      * Objects that were not marked in the previous phase are deallocated in place.
      * There is no compaction
      * Unmarked objects are equal to Dead Objects.
      * Reset phase - do the concurrent reset.
* **G1 Garbage Collector (or Garbage First) in java**
  + G1 garbage collector was introduced in Java 7
  + G1 garbage collector was designed to replace CMS collector (Concurrent Mark-Sweep garbage Collector).
  + G1 garbage collector is parallel, G1 garbage collector is concurrent, and G1 garbage collector is incrementally compacting low-pause garbage collector in java.
  + G1 garbage collector has much better layout from the other garbage collectors like serial, throughput and CMS garbage collectors in java.
  + G1(Garbage First) collector compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions.
  + G1(Garbage First) collector allows customizations by allowing users to specify pause times.
  + G1 Garbage Collector (or Garbage First) limits GC pause times and maximizes throughput.
  + Vm (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java?
    - -XX:+UseG1GC
    - Example of using G1 Garbage Collector in Command Line for starting jar>
    - java -Xms256m -Xms512m -XX:+UseG1GC -jar d:\MyJar.jar
  + **G1(Garbage First) collector functioning**
    - CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.
    - All memory objects end up in one of these three sections.
    - The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.
    - The heap is split/partitioned into many fixed sized regions (eden, survivor, old generation regions), but there is not a fixed size for them. This provides greater flexibility in memory usage.
  + When to use G1 garbage collector?
    - G1 must be used when applications that require large heaps with limited GC latency.
    - Example - Application that require
    - heaps around 5-6GB or larger and
    - pause time required below 0.5 seconds
  + **The G1(Garbage First) collector working Step by Step**
    - G1(Garbage First) garbage collector Heap Structure
      * The heap is split/partitioned into many fixed sized regions (eden, survivor, old generation regions), but there is not a fixed size for them. This provides greater flexibility in memory usage.
      * Each region’s size is chosen by JVM at startup.
      * Generally heap is divided into 2000 regions by JVM varying in size from 1 to 32Mb.
    - G1(Garbage First) garbage collector Heap Allocation
      * As mentioned above there are following region in heap Eden, survivor and old generation region. Also, Humongous and unused regions are there in heap.
    - Young Generation in G1 garbage collector
      * Generally heap is divided into 2000 regions by JVM.
      * Minimum size of region can be 1Mb and
      * Maximum size of region can be 32Mb.
      * Regions are not required to be contiguous like CMS garbage collector.
      * Young GC in G1 garbage collector
        + Live objects are copied or moved to survivor regions.
        + If objects aging threshold is met it get promoted to old generation regions.
        + It is STW (stop the world) event. Eden size and survivor size is calculated for the next young GC.
        + The young GC is done parallely using multiple threads.
      * End of a Young GC with G1 garbage collector
        + At this stage Live objects have been evacuated (copied or moved) to survivor regions or old generation regions.
    - Old Generation Collection with G1 garbage collector
      * G1 collector is low pause collector for old generation objects.
      * Initial Mark -
        + It is STW (stop the world) event.
        + With G1, it is piggybacked on a normal young GC. Mark survivor regions (root regions) which may have references to objects in old generation.
      * Root Region Scanning -
        + Scan survivor regions for references into the old generation.
        + This happens while the application continues to run. The phase must be completed before a young GC can occur.
      * Concurrent Marking -
        + Find live objects over the entire heap.
        + This happens while the application is running.
        + This phase can be interrupted by young generation garbage collections.
      * Remark (Stop the World Event) -
        + Completes the marking of live object in the heap.
        + Uses an algorithm called snapshot-at-the-beginning (SATB) which is much faster than algorithm used in the CMS collector.
      * Cleanup (Stop the World Event and Concurrent) -
        + Performs accounting on live objects and completely free regions. (Stop the world)
        + Young generation and old generation are reclaimed at the same time
        + Old generation regions are selected based on their liveness.
        + Scrubs the Remembered Sets. (Stop the world)
        + Reset the empty regions and return them to the free list. (Concurrent)

# JVM (java virtual machine) in detail in java?

* What is JVM (Java virtual machine) in java?
  + JVM is the virtual machine on which java code executes.
  + JVM is responsible for converting byte code into machine specific code.
* JVM (Java Virtual Machine) Architecture?

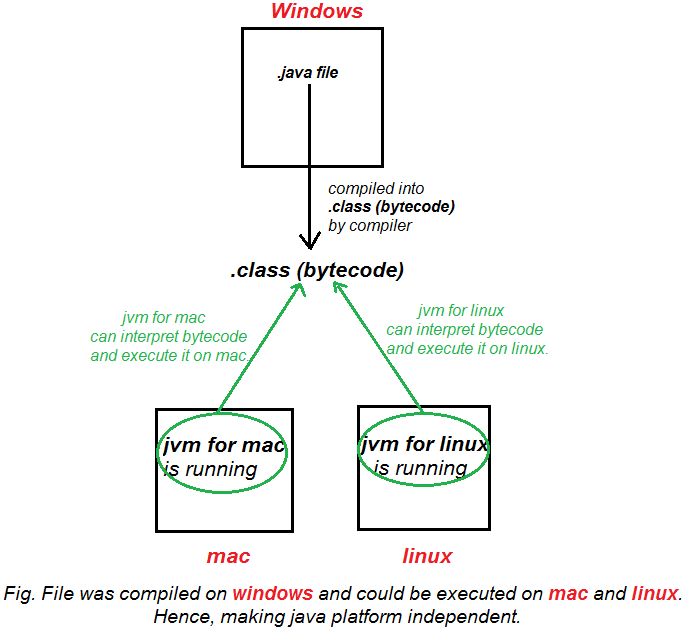


* + JVM (Java Virtual Machine) consists of Class Loader Subsystem, Runtime Data Areas and Execution Engine.
  + Class Loader Subsystem
    - Classloader is a subsystem of JVM.
    - Classloader is used to load class files.
    - Classloader verifies the class file using byte code verifier. Class file will only be loaded if it is valid.
  + Runtime Data Areas
    - Method Area
      * Method area is also called class area.
      * Method area stores data for each and every class like fields, constant pool, method’s data and information.
    - Heap
      * Heap is place where all objects are stored in JVM (java virtual machine).
      * Heap even contains arrays because arrays are objects.
    - Java Threads (Java thread Stacks)
      * You must know that each and every thread has its own stack.
      * How stack frames are created when thread calls new method?
        + As we know each and every thread has its own stack. Whenever Thread.start() method is called new stack frame is created and it is pushed on top of that thread's stack.
      * What does thread stack contains?
        + The stack contain
        + All the local variables,
        + All the parameters,
        + All the return address
      * Does stack stores/contains object OR what stack doesn’t contains?
        + Stack never stores object, but it stores object reference.
    - Program counter registers (PC Registers)
      * Program counter registers contains the address of instructions currently being executed and address of next instruction as well.
    - Native internal Threads (Native thread stack )
      * Native internal threads area contains all the informations related to native platform.
      * Example - If we are running JVM (java application) on windows, it will contain all information related to native platform i.e. windows.
      * If we are running JVM (java application) on linux, it will contain all information related to native platform i.e. linux.
  + Execution Engine
    - Execution Engine contains JIT (Just In Time) Compiler and Garbage collector compiler. Execution Engine also contains Interpreter.
    - JIT(Just In Time) compiler
      * JIT compiler compiles bytecodes to machine code at run time and improves the performance of Java applications.
      * JIT Compiler internal working
        + JIT compilation does require processor time and memory usage. When the JVM first starts up, lots of methods are called. Compiling all of these methods might can affect startup time significantly, though program ultimately may achieve good performance.
        + Methods are not compiled when they are called first time. For each and every method JVM maintains a call count, which is incremented every time the method is called. The methods are interpreted by JVM until call count not exceeds JIT compilation threshold (The JIT compilation threshold improves performance and helps the JVM to start quickly. The threshold has been selected carefully by java developers to obtain an optimal performances. Balance between startup times and long term performance is maintained).
        + Therefore, very frequently used methods are compiled as soon as JVM has started, and less used methods are compiled later.
      * How JIT improves performance of Most frequently used methods ?
        + After a method is compiled, its call count is reset to zero and subsequent calls to the method increment it call count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles method second time, applying more optimizations as compared to optimizations applied in previous compilation. This process is repeated until the maximum optimization level is reached. Most frequently used methods are always optimized to maximize the performance benefits of using the JIT compiler.
        + Example -

Let’s say JIT recompilation threshold = 2

After a method is compiled, its call count is reset to zero and subsequent calls to the method increment it call count. When the call count of a method reaches a 2 (i.e. JIT recompilation threshold), the JIT compiler compiles method second time, applying more optimizations as compared to optimizations applied in previous compilation.

* + - Garbage Collector
      * Garbage Collector Garbage collection is the process by which JVM clears objects (unused objects) from heap to reclaim heap space.
    - Interpreter
      * Interpreter is responsible for reading the bytecode and then executing the instructions.
  + Native method libraries
    - Native method interface is an interface that connects JVM with the native method libraries for executing native methods.
    - Example of Native method libraries>
      * If we are running JVM (java application) on windows, then Native method interface(window method interface) will connect JVM with the window methods libraries(native method libraries) for executing window methods (native methods).
    - You must know about JNI, What is Java Native Interface(JNI)?
      * You may write your application purely in java but there are certain situations where java code might need meet your requirement.
      * Programmers uses the JNI (Java Native Interface) to write the Java native methods when an application cannot be written purely in Java.
* JVM components related to performance
  + Three components Heap, JIT (Just In Time) Compiler and Garbage collector are related to JVM’s performance tuning.
  + Heap and Garbage collector for tuning JVM’s performance >
    - All the objects are stored in heap. Garbage collector manages the heap at JVM initialization.
    - There are many VM (JVM) options for Increasing and decreasing the heap size for managing object for best performance.
    - selecting the different garbage collector depending on your requirement.
  + JIT (Just In Time) Compiler for tuning JVM’s performance >
    - JIT compiler compiles bytecodes to machine code at run time and improves the performance of Java applications.
    - In newer versions of JVM tuning of JIT (Just In Time) Compiler is rarely needed.
* How is java platform independent language?
  + Once source code (i.e. .java file) is compiled on one platform(bytecode is formed). That bytecode can be executed (interpreted) on any other platform running a JVM.
  + Every platform have different JVM implementation.
  + JVM for windows platform is different from JVM for linux platform.



# How to override equals and hashCode in Java?

* [hashCode()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode%28%29) and [equals()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals%28java.lang.Object%29) methods have been defined in **Objectclass** which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods

## ****Usage of hashCode() and equals()****

* 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](https://en.wikipedia.org/wiki/Hash_table) 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.

## ****Overriding the default behavior****

* Everything works fine until you do not override any of these methods in your classes. But, sometimes application needs to change the default behavior of some objects.
* Lets take an example where your application has Employee object. Lets create a minimal possible structure of Employee class::

|  |
| --- |
| public class Employee  {      private Integer id;      private String firstname;      private String lastName;      private String department;        public Integer getId() {          return id;      }      public void setId(Integer id) {          this.id = id;      }      public String getFirstname() {          return firstname;      }      public void setFirstname(String firstname) {          this.firstname = firstname;      }      public String getLastName() {          return lastName;      }      public void setLastName(String lastName) {          this.lastName = lastName;      }      public String getDepartment() {          return department;      }      public void setDepartment(String department) {          this.department = department;      }  } |

* Above Employee class has some very basic attributes and there accessor methods. Now consider a simple situation where you need to compare two employee objects.

public class EqualsTest {

public static void main(String[] args) {

Employee e1 = new Employee();

Employee e2 = new Employee();

e1.setId(100);

e2.setId(100);

//Prints false in console

System.out.println(e1.equals(e2));

}

}

* No prize for guessing. Above method will print “false“. But, is it really correct after knowing that both objects represent same employee. In a real time application, this must return true.
* To achieve correct behavior, we need to **override equals method** as below:

public boolean equals(Object o) {

if(o == null)

{

return false;

}

if (o == this)

{

return true;

}

if (getClass() != o.getClass())

{

return false;

}

Employee e = (Employee) o;

return (this.getId() == e.getId());

}

* Add this method to your Employee class, and EqualsTest will start returning “true“.
* So are we done? Not yet. Lets test again above modified Employee class in different way.

import java.util.HashSet;

import java.util.Set;

public class EqualsTest

{

public static void main(String[] args)

{

Employee e1 = new Employee();

Employee e2 = new Employee();

e1.setId(100);

e2.setId(100);

//Prints 'true'

System.out.println(e1.equals(e2));

Set<Employee> employees = new HashSet<Employee>();

employees.add(e1);

employees.add(e2);

//Prints two objects

System.out.println(employees);

}

}

* Above class prints two objects in second print statement. If both employee objects have been equal, in a Set which stores only unique objects, there must be only one instance inside HashSet, after all both objects refer to same employee. What is it we are missing??
* We are missing the second important method hashCode(). As java docs say, **if you override equals() method then you must override hashCode() method**. So lets add another method in our Employee class.

@Override

public int hashCode()

{

final int PRIME = 31;

int result = 1;

result = PRIME \* result + getId();

return result;

}

* Once above method is added in Employee class, the second statement start printing only single object in second statement, and thus validating the true equality of e1 and e2.

## Overriding hashCode() and equals() using Apache Commons Lang

* Apache commons provide two excellent utility classes [HashCodeBuilder](https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/builder/HashCodeBuilder.html) and [EqualsBuilder](https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/builder/EqualsBuilder.html) for generating hash code and equals methods. Below is its usage:

import org.apache.commons.lang3.builder.EqualsBuilder;

import org.apache.commons.lang3.builder.HashCodeBuilder;

public class Employee

{

private Integer id;

private String firstname;

private String lastName;

private String department;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getFirstname() {

return firstname;

}

public void setFirstname(String firstname) {

this.firstname = firstname;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getDepartment() {

return department;

}

public void setDepartment(String department) {

this.department = department;

}

@Override

public int hashCode()

{

final int PRIME = 31;

return new HashCodeBuilder(getId()%2==0?getId()+1:getId(), PRIME).toHashCode();

}

@Override

public boolean equals(Object o) {

if (o == null)

return false;

if (o == this)

return true;

if (o.getClass() != getClass())

return false;

Employee e = (Employee) o;

return new EqualsBuilder().

append(getId(), e.getId()).

isEquals();

}

}

* *Alternatively*, if you are using any code editor, they also must be capable of generating some good structure for you. For example, **Eclipse IDE** has option under **right click on class >> source > Generate hashCode() and equals() …** will generate a very good implementation for you.

## ****Important things to remember****

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.

## ****Special Attention When Using in**** ORM

* If you’re dealing with an ORM, make sure to always use getters, and never field references in hashCode() and equals(). This is for reason, in ORM, occasionally fields are lazy loaded and not available until called their getter methods.
* For example, In our Employee class if we use *e1.id == e2.id*. It is very much possible that id field is lazy loaded. So in this case, one might be zero or null, and thus resulting in incorrect behavior.
* But if uses *e1.getId() == e2.getId()*, we can be sure even if field is lazy loaded; calling getter will populate the field first.

# What is Polymorphism in Java?

## ****What is Polymorphism?****

Polymorphism means ‘**many forms**’. In OOP, polymorphism means a type can point to different object at different time. In other words, the actual object to which a reference type refers, can be determined at runtime.

In Java, polymorphism is based on inheritance and overriding.

## ****How is Polymorphism Implemented in Java?****

In Java, you can implement polymorphism if you have a super class (or a super interface) with two or more sub classes.

Let’s understand by looking at some examples.

Suppose that we have the following interface and classes:

|  |
| --- |
| public interface Animal {      public void move();  }  public class Dog implements Animal {      public void move() {          System.out.print("Running...");      }  }    public class Bird implements Animal {      public void move() {          System.out.print("Flying...");      }  }    public class Fish implements Animal {      public void move() {          System.out.print("Swimming...");      }  } |

As you can see, we have Animal as the super interface, and 3 sub classes: Dog, Bird and Fish.

Because the Dog implements Animal, or Dog is an Animal, we can write:

|  |
| --- |
| Animal animal = new Dog(); |

Because Bird is an Animal, it’s legal to write:

|  |
| --- |
| Animal animal = new Bird(); |

Likewise, it’s perfect to write:

|  |
| --- |
| Animal animal = new Fish(); |

As you can see, we declare a reference variable called animal, which is of type Animal. Then we assign this reference variable to 3 different kinds of object: Dog, Bird and Fish.

You see? A reference type can take different objects (many forms). This is the simplest form of polymorphism, got it?

Now we come to a more interesting example to see the power of polymorphism.

Suppose that we have a trainer who teaches animals. We create the Trainer class as follows:

|  |
| --- |
| public class Trainer {      public void teach(Animal animal) {          animal.move();      }  } |

Notice that the teach() method accepts any kind of Animal. Thus we can pass any objects which are sub types of the Animal type. For example:

|  |
| --- |
| Trainer trainer = new Trainer();    Dog dog = new Dog();    Bird bird = new Bird();    Fish fish = new Fish();    trainer.teach(dog);  trainer.teach(bird);  trainer.teach(fish); |

Outputs:

|  |
| --- |
| Running…  Flying…  Swimming… |

Here, as you can see, the teach() method can accept ‘many forms’ of Animal: Dog, Bird, Fish,… as long as they are sub types of the Animal interface.

In the teach() method, the move() method is invoked on the Animal reference. And depending on the actual object type, the appropriate overriding method is called. Thus we see the outputs:

|  |
| --- |
| Running… (from the Dog object).  Flying… (from the Dog object).  Running… (from the Dog object). |

## ****Why is Polymorphism?****

Polymorphism is a robust feature of OOP. It increases the reusability, flexibility and extensibility of code. Take the above example for instance:

1. **Reusability**: the teach() method can be re-used for different kinds of objects as long as they are sub types of the Animal interface.
2. **Flexibility**: the actual object can be determined at runtime which allows the code run more flexibly.
3. **Extensibility**: when we want to add a new kind of Animal, e.g. Snake, we just pass an object of Snake into the teach() method without any modification.

# What is Encapsulation in Java?

## ****What is Encapsulation?****

It’s quite difficult to understand the concept of encapsulation from a definition. So let’s understand what encapsulation is by looking at some code examples.

Basically, there are two forms of encapsulation in OOP.

First, ***encapsulation is a technique that packages related data and behaviors into a single unit***. Let’s see an example:

|  |
| --- |
| class Person {      String name;      int age;        void talk() {      }        void think() {      }        void work() {      }        void play() {      }  } |

Here, the common characteristics and behaviors of a person are packaged into a single unit: the Person class. This is the process of encapsulation.

The Person class is an encapsulation unit. A Person object exposes its attributes and behaviors to the outside world:

|  |
| --- |
| Person you = new Person();  you.name = "John";  you.work(); |

Here, encapsulation hides implementation details of the Person class from other objects.

Likewise, creating an interface is also the process of encapsulation:

|  |
| --- |
| interface Human {      void eat();      void talk();      void think();  } |

Again, in terms of encapsulation, this interface groups the essential behaviors of human-being in a single unit.

Second, encapsulation is a technique for protecting data from misuse by the outside world, which is referred as ‘***information hiding***’ or ‘***data hiding***’.

In Java, the access modifier private is used to protect data and behaviors from outside. Let’s modify the Person class above to prevent the attributes name and age from being modified by other objects:

|  |
| --- |
| class Person {      private String name;      private int age;  } |

Here, the fields age and name can be only changed within the Person class. If someone attempts to make a change like this:

|  |
| --- |
| Person p = new Person();  p.name = "Tom"; // ERROR! |

The code won’t compile because the field name is marked as private.

But what if we want the other objects to be able to read the name and age? In this case, we provide methods whose name in the form of getXXX() - so called getters in Java. Hence we add two getters to the Person class:

|  |
| --- |
| class Person {      private String name;      private int age;        public String getName() {          return name;      }        public String getAge() {          return age;      }  } |

This is the process of hiding information. The other objects cannot access the data directly. Instead, they have to invoke the getters which are designed to protect the data from misuse or unwanted changes.

What if we want the outside world to be able to modify our data in a safety manner? In this case, we can provide methods in the pattern of setXXX() - the so called setters in Java. For example, creating a setter for the field name:

|  |
| --- |
| public void setName(String name) {      if (name == null || name.equals("")) {          throw new IllegalArgumentException("name cannot be null or empty!");      }        this.name = name;  } |

Here, someone can change the name of a Person object via this setter method, but he cannot set an empty or null name - as the setter will throw an exception if doing so. This protects data from misuse or malicious changes. For example:

|  |
| --- |
| Person p = new Person();    p.setName("");  // ERROR: IllegalArgumentException will be thrown    p.setName("Tom");   // OK, legal |

Likewise, we can implement a setter for the age attribute that allows the caller to set age in a valid range. For example:

|  |
| --- |
| public void setAge(int age) {      if (age < 18 || age > 55) {          throw IllegalArgumentException("Age must be from 18 to 55");      }      this.age = age;  } |

So far you’ve got an understanding about what encapsulation is in OOP and how it is implemented in Java. In short, encapsulation is a technique for hiding implementation details and restricting access for the outside world.

## ****2. Why Is Encapsulation?****

It’s because encapsulation has a number of advantages that increase the reusability, flexibility and maintainability of the code.

* 1. **Flexibility**: It’s more flexible and easy to change the encapsulated code with new requirements. For example, if the requirement for setting the age of a person changes, we can easily update the logic in the setter method setAge().
  2. **Reusability**: Encapsulated code can be reused throughout the application or across multiple applications. For example, the Person class can be reused whenever such type of object is required.
  3. **Maintainability**: Application code is encapsulated in separate units (classes, interfaces, methods, setters, getters, etc) so it’s easy to change or update a part of the application without affecting other parts, which reduces the time of maintenance.

## ****3. How Is Encapsulation Done in Java?****

As you can see in the above examples, encapsulation is implemented in Java using interfaces, classes, access modifiers, setters and getters.

A class or an interface encapsulates essential features of an object.

Access modifiers (private, public, protected) restrict access to data at different levels.

Setters and getters prevent data from misuse or unwanted changes from other objects.

That’s all for encapsulation topic today. Right now, take a pencil and a paper to take note what you have learned.

## ****4. Encapsulation vs. Abstraction****

So far you got a proper understanding about [abstraction](http://www.codejava.net/java-core/the-java-language/what-is-abstraction-in-java-the-why-and-the-truth) and encapsulation in Object-Oriented Programming with Java. To summary:

* Abstraction is the process of modeling real things in the real word into programming language. In Java, the process of abstraction is done using interfaces, classes, abstract classes, fields, methods and variables. Everything is an abstraction.
* Encapsulation is the process of hiding information details and protecting data and behavior of an object from misuse by other objects. In Java, encapsulation is done using access modifiers (public, protected, private) with classes, interfaces, setters, getters.

So, what are the commons and differences between abstraction and encapsulation?

Many programmers have been wondering about this.

If you are wondering about this too, here’s my answer:

Hey, there are NO commons and differences between abstraction and encapsulation. Why? It’s because they are not comparable.

They are about different things, so we cannot compare them.

As I told you in the article [What is Abstraction in Java](http://www.codejava.net/java-core/the-java-language/what-is-abstraction-in-java-the-why-and-the-truth) , abstraction is the fundamental concept on which other things rely on such as encapsulation, inheritance and polymorphism. In other words, if there is no abstraction, encapsulation would not exist.

Encapsulation is done based on abstraction. Imagine you are building a house. The house is made by bricks. Abstraction is the bricks and encapsulation is the house. Got it?

# Difference between Loose Coupling and Tight Coupling in Java?

First to understand what is Loose coupling and Tight coupling and also to know the advantage of loose coupling.

In short Loose coupling means reducing dependencies of a class that use different class directly. Tight coupling means classes and objects are dependent on one another. In general tight coupling is usually not good because it reduces flexibility and re-usability of code and it makes changes much more difficult and not easy to test.

## ****Tight Coupling****

Tightly coupled object is an object that needs to know quite a bit about other objects and are usually highly dependent on each other's interfaces. Changing one object in a tightly coupled application often requires changes to a number of other objects. In a small application we can easily identify the changes and there is less chance to miss anything. But in large applications these inter-dependencies are not always known by every programmer and there is chance of overlooking changes.

See below code is for tight coupling.

public class Journey {

Car car = new Car();

public void startJourney() {

car.travel();

}

}

public class Car {

public void travel () {

System.out.println("Travel by Car");

}

}

In the above code the Journey class is dependents on Car class to provide service to the end user(main class to call this Journey class).

In the above case Journey class is tightly coupled with Car class it means if any change in the Car class requires Journey class to change. For example if Car class travel() method change to journey() method then you have to change the startJourney() method will call journey() method instead of calling travel() method.

See below code,

public class Journey {

Car car = new Car();

public void startJourney() {

car.journey();

}

}

public class Car {

public void journey () {

System.out.println("Travel by Car");

}

}

The best example of tight coupling is RMI(Remote Method Invocation)(Now a days every where using web services and SOAP instead of using RMI, it has some disadvantages).

## ****Loose Coupling****

Loose coupling is a design goal that seeks to reduce the inter-dependencies between components of a system with the goal of reducing the risk that changes in one component will require changes in any other component. Loose coupling is much more generic concept intended to increase the flexibility of system, make it more maintainable and makes the entire framework more stable.

Below code is an example of loose coupling.

public interface Vehicle {

void start();

}

public class Car implements Vehicle {

@Override

public void start() {

System.out.println("Travel by Car");

}

}

public class Bike implements Vehicle {

@Override

public void start() {

System.out.println("Travel by Bike");

}

}

// create main class Journey

public class Journey {

public static void main(String[] args) {

Vehicle v = new Car();

v.start();

}

}

In the above example, Journey and Car objects are loosely coupled. It means Vehicle is an interface and we can inject any of the implemented classes at run time and we can provide service to the end user.

The examples of Loose coupling are Interface, JMS, Spring IOC(Dependency Injection, it can reduce the tight coupling).

## ****Advantages of Loose coupling****

A loosely coupled will help you when your application need to change or grow. If you design with loosely coupled architecture, only a few parts of the application should be affected when requirements change. With too a tight coupled architecture, many parts will need to change and it will be difficult to identify exactly which parts will be affected. In short,

1) It improves the testability.

2) It helps you follow the GOF principle of program to interfaces, not implementations.

3) The benefit is that it's much easier to swap other pieces of code/modules/objects/components when the pieces are not dependent on one another.

4) It's highly changeable. One module doesn't break other modules in unpredictable ways.