# How does HashMap class works in Java?

In this article I will explain, how does HashMap class of Java works internally? If you are aware how any class works internally then you can use that class more efficiently rather than just using it and making its performance worst and erroneous. After reading this article, you can understand what algorithms and techniques are used by it’s authors to write HashMap class. This article will explain you that, without knowing any class, how drastically performance of that class goes down. Therefore, in any interview, every smart interviewer wants to know how deeply you understand concepts and techniques used in popular classes, which are come to use in development very often. Before explain how HashMap class works internally, I would like to just brush up our general knowledge about HashMap class. I hope you are aware about these things before, but it would be better to remind them before going more deep. HashMap has implemented Map interface. It is allow all optional Map operations, which permits only key-value pair in this collection. HashMap allows only one null as key and multiple nulls as values. Keys must be unique for HashMap, that is why, only one null is allowed as key. This class is roughly equivalent to HashTable but there is one major difference between HashMap and HashTable classes, where all methods of HashTable are synchronized but HashMap methods are not synchronized. That means HashTable operations are thread-safe but HashMap operations are not thread-safe. HashMap neither gives guarantee of order nor sorting of elements. Now we can start digging in HashMap class. Most important thing to know about HashMap are data structures and algorithms used to write this class. As name of class "HashMap" is indicating that it works on hashing mechanism. This class basically uses two data structures, one is Array and other is Linked-List. HashMap internally create Array of Entry type objects. Entry is an inner class used by HashMap to stores Key and Value types objects. Whenever you put any key and value in HashMap class it internally create an Entry object to hold data inside. To place an Entry object in array, it need an index at which index this Entry object can store in array. This index is generated by hash code of Key object provide by user. Hash code of Key object can get by hashCode() method of Key object. After knowing index, Entry object can place in array. These array indexes are known as buckets. Now if you want to retrieve any object from HashMap, you need to provide Key object for that. This Key object is used to get hash code and generate an index. Now at this index or bucket there is an Entry object which contains key and value. After knowing Entry object it could return value from it. But what will happen, if more than one keys objects have same hash codes, in this situation more than one keys objects will generate same index for Entry array. This situation is known as collision. When more than one keys objects have same hash codes. In this circumstances on same index more than one Entry objects need to be stored, but in array you can only store one value at an index. Now Linked-List comes in picture. To solve this problem, Entry objects growing links as Linked-List, every collision on that index will be the next link of this Listed-List. During retrieval of object on basis of key object, if HashMap observe any collision on that index, class search that object in associated Linked-List instead of array. So this is the mechanism on which HashMap class works. Now question arise, what is the significance of knowing this internal behavior. The answer is, now you know, how efficiently you can use this class, not only this class all most every class which uses hashing mechanism to store and retrieve data. It is most essential to use key objects which generate almost unique hash codes. To achieve this you need to make sure that the class you are using as key must override equal() and hashCode() methods as such way that your key object generate unique hash code. You need to take care, very least collision should occur in your hash codes otherwise HashMap internal array structure would convert into Linked-List which is least efficient than array. Where big O notation of array is O(1) and Linked-List has O(n). So you need to take care, less collision will make your HashMap performance better that anything. Hope this article will help you to understand hashing tricks, how to use hashing classes more efficiently and get best outputs.

# Why HashTable class does not allow null key and null values?

If you will see carefully to HashTable class, you will notice that HashTable class is part of Java since JDK1.0. Its means this class is very old class. To understand this, first of all you need to understand comments written on this class by author.

"This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method."

HashTable class is implemented on hashing mechanism, that’s mean to store any key-value pair, its required hash code of key object. If key would be null, it will not able to given hash code. That is why HashTable refuse to accept null key and values to store in it.

But later on it was realized that null key and value has its own importance that is why one null key and multiple null values are allowed in later implemented classes like HashMap class.

# How to create Thread Pool in Java using Executors framework?

Since JDK1.5 one new concurrency package called java.util.concurrent was introduced in Java. This new package has multiple concurrent utilities to protect and help Java applications in concurrent environment. Thread Pooling is one of them to boost threading performances. Thread Pool has multiple benefits in multi-threading application. To understand Thread Pooling, you need to understand meaning of pooling. Pooling is one kind management of objects collection. This collection of objects, protect objects from garbage collection and reuse them to fulfill any specific purpose. Reusing live object is really good idea to save time and memory of application. For example, you might have use Connection Pooling to connect your application with database. Connection Pooling has same concept to reuse connection objects to connect with database every time. Now it is time to know the advantages of Thread Pooling in concurrent application. A marginal amount of time of CPU is consumed to initializing and distorting to a thread. So it is good idea to reuse existing thread instead of creating new thread every time. Therefore Thread Pooling is solution of this problem. To create thread pool, you can create fixed number of worker threads in Pool and give these worker threads to a corresponding work or task to perform. Now fixed number of worker threads will finish their task and will get reading to accept new task to perform, until whole tasks get finished. Thread Pool will save big about of time of CPU as well as memory, by not to create and destroy threads every time. There is one more benefit of pooling is that, by creating Thread Pool, it is easy to manage number to threads for an application. You can control how many threads are safe for an application instead of creating too many threads without know application capability. You can create fixed number of threads in a pool instead of just creating countless threads in application. If your application is not able to control number of threads for your application, then it could be a cause of OutOfMemoryError. Now you know what is Thread Pooling as well as benefits of Thread Pooling. But now question arise, how to create Thread Pool in Java? As I mentioned above, now Java has new concurrency package java.util.concurrent. This concurrency package has included, Executors framework to create and manage Thread Pool. Here I would like to show you Java programming example to show you, how Thread Pool get created and how to manage them. In this example, there are two Thread Pools one is Producer and other is Consumer. Producer Thread Pool will produce some output and Consumer Thread Pool will consume that output. Both Thread Pools have fixed number of worker threads to perform given tasks. Now take a look to this coding example.

**SharedPlace.java**

/\*\*

\* SharedPlace class is common shared place where Producer Thread Pool places its outputs

\* and Consumer Thread Pool fetch those outputs and consume them.

\*

\*/

**public** **class** SharedPlace {

**private** String text;

**public** String getText() {

**return** text;

}

**public** **void** setText(String text) {

**this**.text = text;

}

**public** **void** remove() {

**this**.text = **null**;

}

**public** **boolean** isEmpty() {

**return** (**this**.text == **null**) ? **true** : **false**;

}

}

**Producer.java**

/\*\*

\* Producer class is actually Producer Thread Pool, which would produce some output

\* <code>String </code> and place this <code>String</code> to a Shared Place.

\*/

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** Producer {

**private** SharedPlace sharedPlace;

**private** **final** **int** COUNT;

**public** Producer(SharedPlace sharedObj, **int** count) {

**this**.sharedPlace = sharedObj;

**this**.COUNT = count;

}

**public** **void** startProduction() {

ExecutorService executorService = Executors.newFixedThreadPool(5);

**for** (**int** i = 1; i <= COUNT; i++) {

executorService.execute(**new** Task(i));

}

executorService.shutdown();

}

**private** **class** Task **implements** Runnable {

**private** **final** **int** TASKNUM;

**public** Task(**int** taskNum) {

TASKNUM = taskNum;

}

**public** **void** run() {

String str = "TASK-"+TASKNUM;

**synchronized** (sharedPlace) {

**while** (!sharedPlace.isEmpty()) {

**try** {

sharedPlace.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

sharedPlace.setText(str);

sharedPlace.notifyAll();

}

}

}

}

**Consumer.java**

/\*\*

\* Consumer class is Consumer Thread Pool, who will fetch shared <code>String</code> objects from

\* SharedPlace class and consume them.

\*/

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** Consumer {

**private** **final** **int** COUNT;

**private** **final** SharedPlace sharedPlace;

**private** **int** workDone = 0;

**public** Consumer(SharedPlace sharedObj, **int** count) {

**this**.sharedPlace = sharedObj;

**this**.COUNT = count;

}

**public** **void** startConsumption() {

ExecutorService executorService = Executors.newFixedThreadPool(5);

**for** (**int** i = 0; i < COUNT; i++) {

executorService.execute(**new** Worker());

}

executorService.shutdown();

}

**private** **class** Worker **implements** Runnable {

**public** **void** run() {

**synchronized** (sharedPlace) {

**while** (sharedPlace.isEmpty()) {

**try** {

sharedPlace.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

String text = sharedPlace.getText();

sharedPlace.remove();

workDone++;

System.out.println("WORK DONE=" + workDone + " " + text);

sharedPlace.notifyAll();

}

}

}

}

**ThreadPool.java**

/\*\*

\* ThreadPool class is initializer class which would initialize both

\* Producer and Consumer Thread Pools and initiate their functioning.

\*/

**public** **class** ThreadPool {

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

SharedPlace sharedObj = **new** SharedPlace();

Producer producer = **new** Producer(sharedObj, 100);

Consumer consumer = **new** Consumer(sharedObj, 100);

producer.startProduction();

consumer.startConsumption();

}

}

Try this code and see how Thread Pool works in java. If you have any doubt or advise, please write and post your queries or comments below this blog. Hope this will help you.

# Printing Even and Odd using two Threads in Java

This is really very good programming question to know multithreading synchronization skill of candidates. Because anyone who is working on Java multithreading environment can easily understand that how unpredictable threads can execute. In other word, outputs of threads are not guaranteed. Therefore it is really tough task to synchronize two or more threads to get desired outputs from threads. Here you can see, how you can bring desired outputs from two different threads. We have two threads, one thread prints only even numbers and other thread prints only odd numbers. Now task is to synchronize both even and odd threads in that way, so they can print even and odd in natural consecutive manners. For example

**Odd Number: 1**

**Even Number: 2**

**Odd Number: 3**

**Even Number: 4**

**Odd Number: 5**

**Even Number: 6**

In this given example you can see, how you can synchronize two threads.

**public** **class** EvenOddTest {

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

EvenOddPrinter printer = **new** EvenOddPrinter();

Thread even = **new** Thread(**new** PrintTask(printer, 100, **true**));

Thread odd = **new** Thread(**new** PrintTask(printer, 100, **false**));

odd.start();

even.start();

}

}

**class** PrintTask **implements** Runnable{

**int** max = 0;

**boolean** isEven = **false**;

EvenOddPrinter p = **null**;

**int** num = 0;

**public** PrintTask(EvenOddPrinter p, **int** max, **boolean** isEven) {

**this**.p = p;

**this**.max = max;

**this**.isEven = isEven;

}

**public** **void** run() {

num = (isEven) ? 2 : 1;

**while** (num <= max) {

**if** (isEven){

p.printEven(num);

}

**else** {

p.printOdd(num);

}

num += 2;

}

}

}

**class** EvenOddPrinter {

**boolean** isOdd = **false**;

**public** **synchronized** **void** printEven(**int** number) {

**while** (!**this**.isOdd) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("Even Number : " + number);

**this**.isOdd = **false**;

notify();

}

**public** **synchronized** **void** printOdd(**int** number) {

**while** (**this**.isOdd) {

**try** {

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println("Odd Number : " + number);

**this**.isOdd = **true**;

notify();

}

}

Now try this program and you will see, how you can synchronize two threads for desired outputs. If you have any doubt please post your comments below this blog.

# How to make threads in sequence execution without using join method?

This is really very good programming question to know multithreading synchronization skill of candidates. Because anyone who is working on Java multithreading environment can easily understand that how unpredictable threads can execute. In other word, outputs of threads are not guaranteed. Therefore it is really tough task to synchronize two or more threads to get desired outputs from threads. Here you can see how you can bring desired outputs from two or more different threads. We have three threads T1, T2 and T3, which must be execute in sequence, like T1 -> T2 --> T3. This can be done by join() method of java.lang.Thread class easily but without using join() method this is quite complicated in Java multi-threading. Now this can be done with wait() andnotify() methods. For example

Job:1 : JOB A DONE...

Job:2 : JOB B DONE...

Job:3 : JOB C DONE...

In this given example you can see, how you can synchronize thread threads without using join() method.

/\*\*

\* How could you make sure that thread A ,B and C run sequentially without using join method?

\*

\*/

**public** **class** ThreadJoiningWithoutJoin {

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

**final** JoinTask task = **new** JoinTask();

Thread A = **new** Thread(){

**public** **void** run(){

task.doJob(1, "JOB A DONE...");

}

};

Thread B = **new** Thread(){

**public** **void** run(){

task.doJob(2, "JOB B DONE...");

}

};

Thread C = **new** Thread(){

**public** **void** run(){

task.doJob(3, "JOB C DONE...");

}

};

C.start();

B.start();

A.start();

}

}

// Shared Class or Object

**class** JoinTask {

**private** **int** currentRank = 1;

**public** **void** doJob(**int** rank, String msg) {

**synchronized**(**this**) {

**while** (rank != currentRank) {

**try** {

wait();

}**catch**(InterruptedException ex) {

ex.printStackTrace();

}

}

System.out.println("Job:" + currentRank + " : " + msg );

currentRank++;

notifyAll();

}

}

}

Now try this program and you will see, how you can synchronize three threads for desired outputs without using join() method.

# User Defined Marker Interface in Java

Marker Interfaces are nothing but an Interface with no method declared in it. Basically Marker Interfaces was designed so that JVM can check whether any given object is instance of Marker Interface (with instanceof keyword in Java). So that JVM can treat it with special manner. Some of Marker Interfaces Examples are:

* java.lang.Cloneable
* java.io.Serializable
* java.util.EventListener

Yes you can define user defined Marker Interfaces in Java. You can create Interface without any method and can check whether any given object is instance of that Marker Interface and take your decision on basis of this, whatever you want to do. But for this purpose to check instance type of Marker Interface is old way. Now instead of Marker Interface you can define annotations and can check annotations instead of instance type. If you want to know much about annotations in Java programming language. Please read my blog ["How to create and implement custom annotations in Java?"](http://www.somanyword.com/2014/02/how-to-create-and-implement-custom-annotations-in-java/). This will help you to understand concept behind annotations.

# Rules for method Overriding in Java

Overriding is very significant and confusing part of Java language. Sometime experienced developers get confused on Overridden rules. In this blog you can understand these rules with good and simple illustrations. Overridden rules in Java are given below:

**1. Arguments:** Arguments must not change during overriding any method. Even you cannot use subtype of any class as argument in case of overridden method. You must have use exact same argument type used in Super method. For example:

**class** SuperClass {

**public** **void** method(SuperClass args) {

System.out.println(args);

}

}

**class** SubClass **extends** SuperClass {

//Invalid Overridden method, because argument type must be same.

@Override

**public** **void** method(SubClass args){

System.out.println(args);

}

//Valid Overridden method, because argument is same here.

@Override

**public** **void** method(SuperClass args){

System.out.println(args);

}

}

**2. Return types:**Return type of overridden method cannot change except subtype or sub-class of return type. This was not valid before Java5. But now it is valid since Java5. For example:

**class** SuperClass {

**public** SuperClass method() {

**return** **null**;

}

}

**class** SubClass **extends** SuperClass {

// Valid Overridden method, because return type is same as in overridden method,

@Override

**public** SuperClass method() {

**return** **null**;

}

// Valid Overridden method, because return type is sub-class of overridden method return type (Only valid Java 5 and upper versions)

@Override

**public** SubClass method() {

**return** **null**;

}

// Invalid Overridden method, because return type is neither same type of overridden method return type nor sub-class of Overridden method return type.

@Override

**public** String method() {

**return** **null**;

}

}

**3. Access Modifier:** Access modifier of overridden method cannot be more restrictive. But it can be less restrictive. Default access modifier can be Protected or Public and Protected access modifier can be Public. But Public cannot be Protected or Default. For example:

**class** SuperClass {

**protected** **void** method() {

}

}

**class** SubClass **extends** SuperClass {

// Valid Overridden method, because Protected modifier can be Public.

@Override

**public** **void** method() {

}

// Invalid Overridden method, because Protected modifier cannot be Default.

@Override

**void** method() {

}

}

**4. Method Invocation:** Which method would be called is depends on object type at runtime. For example:

**class** SuperClass {

**public** **void** method() {

System.out.println("SuperClass method called!");

}

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

SuperClass superObj = **new** SubClass();

superObj.method();

}

}

**class** SubClass **extends** SuperClass {

@Override

**public** **void** method() { // Valid Overridden method

System.out.println("SubClass method called!");

}

}

Output of this program is “SubClass method called!” because at runtime Object of SubClass was created by SubClass constructor.

**5. Exceptions:** Rules related to exceptions during method overriding is quite complicated. So you need to understand it more carefully. Most important points are given below:

• Exception can reduce during override.

• Exception can eliminate also.

• Must not throw new checked exception.

• Must not throw new broader checked exception.

• New RuntimeException can be thrown and can be eliminate.

For example:

**class** SuperException **extends** Exception {};

**class** SubException **extends** SuperException {};

**class** AnyException **extends** Exception {};

**class** OneMoreException **extends** Exception {};

**class** NewRuntimeException **extends** RuntimeException {};

**class** SuperClass {

**public** **void** method() **throws** SuperException, AnyException {

System.out.println("Hello!");

}

}

**class** Reduced **extends** SuperClass {

// Valid Overriding

@Override

**public** **void** method() **throws** SuperException { // Reduced

System.out.println("Hello!");

}

}

**class** Eliminated **extends** SuperClass {

// Valid Overriding

@Override

**public** **void** method() { // Reduced

System.out.println("Hello!");

}

}

**class** Narrow **extends** SuperClass {

// Valid Overriding

@Override

**public** **void** method() **throws** SubException { // Narrow

System.out.println("Hello!");

}

}

**class** NewException **extends** SuperClass {

// Invalid Overriding, Compiler error

@Override

**public** **void** method() **throws** OneMoreException { // New Checked Exception

System.out.println("Hello!");

}

}

**class** Broader **extends** SuperClass {

// Invalid Overriding, Compiler error

@Override

**public** **void** method() **throws** Exception { // New Broader Checked Exception

System.out.println("Hello!");

}

}

**class** NewRuntime **extends** SuperClass {

// Valid Overriding

@Override

**public** **void** method() **throws** NewRuntimeException { // New RuntimeException

System.out.println("Hello!");

}

}

This is all, how method overriding works in Java

# How many types of Design Patterns exist?

There are basically three categories of design patterns which are described by GOF (Gang of Four) and used for software development.  These categories are:

1. Creational Patterns
2. Structural Patterns
3. Behavioral Patterns

**Creational Patterns:**Creational patterns are those design patterns which deals with object creation.  All the design patterns which are implements to create objects and decouple client from creating and initializing objects are known as creational patterns. According to definition:

“*Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner which is suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.”*

Design patterns which are come in creational pattern categories are given below: Factory, Abstract Factory, Builder, Prototype and Singleton pattern.

**Structural Patterns:**Structural patterns describe how objects and classes can be combined to form a structure and provide suitable solution for a particular problem. Structural patterns are further divided in two types of patterns object patterns and class patterns. The difference is that class patterns describe relationships and structures with the help of inheritance. Object patterns, other hand, describe how object can be associated and aggregated to form larger, more complex structures. According to definition:

*“In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.”*

Design patterns which are come in Structural patterns categories are given below: Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy pattern.

**Behavioral Patterns:** Behavioral patterns are patterns that focus on the interactions between cooperating objects. The interactions between cooperating objects should be such that they are communicating while maintaining as loose coupling as possible. The loose coupling is the key to n-tier architectures. In this, the implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.

“*In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.”*

Design patterns which are come in behavioral pattern categories are given below: Chain-of-responsibility, Command, Iterator, Mediator, Memento, Observer, State and Strategy pattern.

# Difference between Factory and Abstract Factory Patterns

When we talk about Factory and Abstract Factory Patterns, conceptually both patterns are equal, because Factory and Abstract Factory are creational patterns, which mean these design patterns deals with object creation.  Both design patterns decouple client from object creation directly. Factory and Abstract Factory patterns return desired object on the basis of client requirements without giving any information to object creation to client.  Client does not need to bother about how to create their desired complex object. These patterns take responsibility of creating desired object on client requirements and return it to client.

But there are significant differences between Factory and Abstract Factory patterns despite of these similarities. There is one major difference is that Factory pattern can only create with Class Creational Pattern, where Abstract Factory pattern can also created with Object Creational Pattern.  The difference is that Class Creational patterns describe relationships and structures with the help of inheritance. Object Creational patterns, other hand, describe how object can be associated and aggregated to form larger, more complex structures. Therefore, Factory pattern can only implemented with the help of inheritance and Abstract Factory pattern can implement with inheritance but Abstract Factory is not limited with inheritance only, it may implemented with other patterns like Builder or Prototype patterns.

Abstract Factory pattern has one more layer of abstraction which does not exists in Factory pattern. One more difference is that Factory pattern returns products types where Abstract Factory pattern return factory types, later on this factory will return their corresponding products to clients. For example, a restaurant serves different kind of burgers, now if we create factory pattern of burger types, it will create a Burger Factory, this Burger Factory will gives different kind of objects of burgers based on client requirements, like veg burger, chicken burger or hamburger. Therefore these returned burger object is final product. But if we talk about Abstract Factory pattern, it returns factory types first instead of products. In same example, in case of Abstract Factory pattern, first of all a Restaurant Factory will be created, now this factory can have multiple food items like burgers, finger fries and cold drinks. This Restaurant Factory will return single food type or drink type object on the basis of client demand, now this food type or drink type object will return final product type object. So you can see Abstract Factory has one more level abstraction on Factory pattern.

If you want to know these patterns more detailed you can read my other blogs related to these patterns. Where you can see the practical examples and understand these better way.

# How to design Factory Pattern using Java

Factory design pattern is creational design pattern, which deals with object creation. Factory pattern decouple client from creating object directly. This design pattern take responsibility to create object behalf of client and return internally constructed object back to client. This design pattern does not disclose complex object construction to client, which is not important for client to know about. Factory pattern is Class Creational Pattern which describes relationships and structures with the help of inheritance. To understand it better, let’s take an example and design a factory pattern for this. Suppose there is a McDonald's restaurant which serves different types of burgers, either it could be veg-burgers or non-veg burgers. These burgers types could have multiple types of veg or non-veg burgers. Now we have create factory patterns which create a Burger Factory, this factory internally create burger objects on the basis of client requirement and return suitable constructed object to client. Now take a look to example. **Burger.java**

**public** **class** Burger {

**protected** String burgerName;

**protected** String[] ingredients;

**public** Burger(String burgerName, String[] ingredients) {

**this**.burgerName = burgerName;

**this**.ingredients = ingredients;

}

**public** String getBurgerName(){

**return** **this**.burgerName;

}

**public** String[] getIngredients() {

**return** **this**.ingredients;

}

}

**VegBurger.java**

**public** **class** VegBurger **extends** Burger {

**public** VegBurger(String burgerName, String[] ingredients) {

**super**(burgerName, ingredients);

}

**public** String getBurgerName(){

**return** **this**.burgerName;

}

**public** String[] getIngredients() {

**return** **this**.ingredients;

}

}

**NonVegBurger.java**

**public** **class** NonVegBurger **extends** Burger {

**public** NonVegBurger(String burgerName, String[] ingredients) {

**super**(burgerName, ingredients);

}

**public** String getBurgerName(){

**return** **this**.burgerName;

}

**public** String[] getIngredients() {

**return** **this**.ingredients;

}

}

**BurgerFactory.java**

**public** **class** BurgerFactory {

**public** **static** **final** **int** VEG\_BURGER = 1;

**public** **static** **final** **int** NON\_VEG\_BURGER = 2;

**public** **static** Burger getBurgerInstance(**int** burgerType) {

**if** (burgerType == VEG\_BURGER) {

**return** **new** VegBurger("Mc Vegi Burger", **new** String[] {"Bread", "Potato", "Peas", "Onion"});

} **else** **if** (burgerType == NON\_VEG\_BURGER) {

**return** **new** NonVegBurger("Mc Chicken Burger", **new** String[] {"Bread", "Chicken", "Oil", "Onion"});

} **else** {

**return** **null**;

}

}

}

**TestFactoryPattern.java**

**public** **class** TestFactoryPattern {

// Check how to use Factory pattern.

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

Burger burger1 = BurgerFactory.getBurgerInstance(BurgerFactory.VEG\_BURGER);

Burger burger2 = BurgerFactory.getBurgerInstance(BurgerFactory.NON\_VEG\_BURGER);

System.out.println(burger1.getBurgerName());

String[] vegBuegerInredients = burger1.getIngredients();

**for** (String ingredient : vegBuegerInredients){

System.out.println(ingredient);

}

System.out.println(burger2.getBurgerName());

String[] nonVegBuegerInredients = burger2.getIngredients();

**for** (String ingredient : nonVegBuegerInredients){

System.out.println(ingredient);

}

}

}

Please look this example carefully, how factory pattern construct object internally for client without give any information to client about object construction. Hope this blog will help you to understand factory pattern easily.

# What is ThreadLocal in java?

Most of Java developers do not know much about ThreadLocal. Let me put some light on this topic. TheadLocal is a helping class in Java multithreading (java.lang.ThreadLocal). This class can be used for data confinement and synchronize threads. Most important things to know about ThreadLocal is, access and scope of this class. This could be little confusing but I would help you to make it easy to understand with relevant examples. Now, let us talk about ThreadLocal more deeply. ThreadLocal has both scopes, global as well as local. These both terms, global and local create confusion, because how a class can have global and local both scope. Therefore, we need to understand it more carefully. ThreadLocal has a local scope for an individual thread; no other thread can interfere in this. If any thread has stored some data in ThreadLocal class, that particular thread can only retrieve that data from ThreadLocal. No other thread can interfere in it and retrieve that data. This is why we are saying ThreadLocal has local scope. But now question is how this class can be access globally. As you know any global variable or class can be assess from anywhere in application. Same here this ThreadLocal can be access from anywhere in a particular thread. For example, there is a thread, which is using method a() of class A and method b() of class B in its execution ( Thead -> a() -> b() ). These methods can access ThreadLocal anywhere for that particular thread execution. This is how its work globally. Here, I am showing you a coding example, how to implement and use ThreadLocal in Java. **CustomThreadLocal.java**

**public** **class** CustomThreadLocal {

**private** **static** ThreadLocal<String> threadLocal = **new** ThreadLocal<String>();

**public** **static** **void** add(String id) {

threadLocal.set(id);

}

**public** **static** String get() {

**return** threadLocal.get();

}

}

**Display.java**

**public** **class** Display {

**public** **void** displayId() {

System.out.println(Thread.currentThread().getName() + ": "+ CustomThreadLocal.get());

}

}

**WorkerThread.java**

**public** **class** WorkerThread **extends** Thread {

**public** **void** run() {

**long** id = Thread.currentThread().getId();

String threadId = "TID-"+id;

CustomThreadLocal.add(threadId);

System.out.println(Thread.currentThread().getName() + " has alocated id: " + threadId);

Display display = **new** Display();

display.displayId();

}

}

**TestThreadLocal.java**

**public** **public** **class** TestThreadLocal{

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

WorkerThread firstThread = **new** WorkerThread();

WorkerThread secondThread = **new** WorkerThread();

WorkerThread thirdThread = **new** WorkerThread();

firstThread.start();

secondThread.start();

thirdThread.start();

}

}

**Output:**

Thread-0 has alocated id: TID-9

Thread-1 has alocated id: TID-10

Thread-2 has alocated id: TID-11

Thread-0: TID-9

Thread-1: TID-10

Thread-2: TID-11

Output of this program is interesting, in which you can see, ThreadLocal did not share data between different threads, ThreadLocal is very synchronized and safe.

# Print pairs of numbers from sorted array, whose sum is any given number

Here we are writing algorithm for a given problem. Problem is suppose, these is a sorted array of integer numbers and a given number S. You have to write possible combinations of integer numbers, whose sum is S. You should not repeat same digits again and complexity of algorithm must not be more that O(n). For example given numbers in array are {1,2,4,5,7,8,9,10,11,12,13,15,16,17} and sum should be 15(SUM). There for all possible combination would be:

Possible combination of numbers in given array, whose sum is: 15

2 + 13 = 15

4 + 11 = 15

5 + 10 = 15

7 + 8 = 15

Let us see real time implementation in java.

**public** **class** Algorithm {

/\*\*

\* Print the pair here such that a[i]+a[j] = sum

\* @param array : Sorted integer array.

\* @param sum : Integer value to check sum.

\* @Efficiency : O(n)

\*/

**public** **void** printPair(**int**[] array, **int** sum)

{

**int** length = array.length;

**int** i = 0, j = length - 1;

**while** (i < j) {

**int** number = sum - array[i];

**if** (array[j] == number) {

System.out.println(array[i] + " + " + array[j] + " = " + sum);

j--;

}**else** **if** (array[j] < number) {

i++;

}**else** {

j--;

}

}

}

**public** **static** **void** main(String... args) {

**final** **int**[] arr = **new** **int**[] {1,2,4,5,7,8,9,10,11,12,13,15,16,17};

**final** **int** SUM = 15;

Algorithm algo = **new** Algorithm();

System.out.println("Possible combination of numbers in given array, whose sum is: " + SUM);

algo.printPair(arr, SUM);

}

}

Try this algorithm, whose complexity is O(n). Hope you find it useful.

# How to print unique numbers from two sorted arrays.

In given problem we have to write unique numbers from two sorted arrays, which do not exists in each other. For example, you have two sorted arrays A and B. A = [1, 3, 4, 6,8,10, 17, 34] B = [2, 8, 17, 33, 44, 66, 89, 100, 123] Write a program to print all those numbers which are 1) in A and not in B 2) in B and not in A  Output should be: {1 , 3 , 4 , 6 , 10, 33, 34, 44,, 66, 89, 100, 123 } I am implementing this algorithm in Java.

**public** **class** Algorithm{

**public** **void** printUnique(**int**[] arrA, **int**[] arrB) {

**int** aLength = arrA.length;

**int** bLength = arrB.length;

**int** a = 0;

**int** b = 0;

**while** (**true**) {

**if** (a == aLength){

**for** (;b < bLength; b++)

System.out.print(arrB[b] + " ");

**break**;

}

**if** (b == bLength) {

**for** (;a < aLength; a++)

System.out.print(arrA[a] + " ");

**break**;

}

**if** (arrA[a] < arrB[b]) {

System.out.print(arrA[a] + " ");

a++;

}

**else** **if** (arrA[a] > arrB[b]) {

System.out.print(arrB[b] + " ");

b++;

}

**else** {

a++;

b++;

}

}

}

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

**int**[] arrA = **new** **int**[]{1, 3, 4, 6,8,10, 17, 34};

**int**[] arrB = **new** **int**[] {2, 8, 17, 33, 44, 66, 89, 100, 123};

**new** Algorithm().printUnique(arrA, arrB);

}

}

Here I am showing you output of this algorithm. **Output is:**

1 2 3 4 6 10 33 34 44 66 89 100 123

Hope you find it useful.

# Print diagonals of a given matrix

Printing diagonals of a matrix is quite easy, but you need to understand the relation between matrix and diagonals. If you want perfect diagonals from matrix then your matrix must be square, I mean your matrix should be NxN. For example 2x2, 4x4 or 9x9 etc. Because in rectangular matrix (2x3 or 5x7) it is not possible to find perfect diagonals. 1      2      3      4 5      6      7      8 9      10   11    12 13    14   15    16 Now if you see above matrix carefully there are two diagonals in this one is 1, 6, 11, 16 and other is 4, 7, 10, 13. Therefore you can write your algorithm with O(log n) complexity.

**public** **class** Algorithm {

/\*\*

\* Print diagonals of matrix.

\*/

**public** **void** printDiagonalsOfMatrix() {

**int**[][] matrix = **new** **int**[][]{{1,2,3,4},{5,6,7,8},{9,10,11,12}, {13,14,15,16}};

**int** length = matrix.length;

**for** (**int** i = 0; i < length; i++ ) {

System.out.println("Diagonal-1 :" + matrix[i][i] + " Diagonal-2 : " + matrix[i][(length-1)-i]);

}

}

**public** **static** **void** main(String... args) {

Algorithm algo = **new** Algorithm();

algo.printDiagonalsOfMatrix();

}

}

**Output of this algorithm is:**

Diagonal-1 :1 Diagonal-2 : 4

Diagonal-1 :6 Diagonal-2 : 7

Diagonal-1 :11 Diagonal-2 : 10

Diagonal-1 :16 Diagonal-2 : 13

Hope you will find this algorithm easy and useful.

# What is stored procedure in MySQL?

**Introduction** Stored procedure is a collection of SQL commands (statements) compiled under one name. A stored procedure can be called simply by EXECUTE command followed by stored procedure name and parameters ( *if any*), which we will see further in an example EXECUTE [*Stored Procedure Name*] *@parameter1,@parameter2* A stored procedure can be invoked from an application created in another language like C#, VB, Java etc. , from a trigger or from another stored procedure. **Advantages of Stored Procedure**

* **It reduces the data traffic between application and database server**- As we have seen to call a stored procedure only name and parameters need to be sent instead of sending long and lengthy SQL query statements. This reduces the data traffic.
* **Procedures are pre-compiled statements-**Like SQL queries, stored procedures are not required to compile every time before executing it on database server. Instead of stored procedures are pre-compiled statements, which get compiled once on database and exist there forever. Therefore stored procedures get execute more faster than SQL queries, because SQL queries travels from application to database server via networks and then get compiled on database server after that it get executed. On other hand stored procedures already existing on database server and already complied, which has to just execute on database server. Therefore stored procedures save lot of time, which are required for traveling and compiling SQL statements.
* **More logic can be applied at database end**- If there are choices between queries to run on database server, depending on the data of a table, it can be done inside one stored procedure, instead of returning the result to application and to check the condition there and then sent appropriate query. For example – *IF (Select sum(Quantity)>10) Begin Query1 End Else Begin Query 2 End*If stored procedure is NOT used in the above example, then value of *Sum(Quantity)* has to be sent to application. In application the conditional logic needs to be applied which would decide the appropriate query to be called depending on the logic. This advantage of stored procedure reduces application’s line of code, data traffic and hence overall execution time.
* **They are more secure**- Stored procedures are secure. Database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permission on the underlying database tables.
* **Reusable Code**- Stored procedures are reusable. As it’s a collection of code statements under one name, it can be reused by any number of applications by calling it with respective name and suitable parameters if any.

**Disadvantage of Stored Procedures**

* **Increasing Load on database-**As I have explained above, stored procedures exists on database server, which increase significant size of database on database server.
* **Sufficient Knowledge-**You need to have sufficient knowledge of database and stored procedures before create a good stored procedures.
* **Migration-** Stored procedures are database dependents if you have created some stored procedures on MySql and you want to migrate them to Oracle or MS-Sql database, then you can not migrate them directly. You need to do compatible changes on them and recompile them on other databases.

**Create Stored Procedure** Now let us see step by step, how to create a stored procedure in MySql. We are going to create a procedure named *GetAllEmployees().* This procedure will select all the employees from *EmployeeInfo* table. Below is the general syntax to create procedure in left column and actual SQL statements by which our *GetAllEmployees()*procedure would be created in right column. Open MYSQL query window and write the statements from right column and execute it. It would create a procedure named *GetAllEmployees()*in your current database.

|  |  |
| --- | --- |
| Syntax | Actual Statements |
| DELIMITER [*New delimiter*]CREATE PROCEDURE [*Stored Procedure Name*]BEGIN SQL QueryEND DELIMITER [*Old delimiter*] | DELIMITER //CREATE PROCEDURE *GetAllEmployees()* BEGIN *Select \* from Employeeinfo;*END // DELIMITER ; |

  Now let’s understand each statement in detail

* First statement DELIMITER // is used to change the standard delimiter, which is semicolon ( ; ) to any other delimiter. In above query delimiter ( ; ) is changed to (//). This is done, so that MySql could take the stored procedure as whole, instead of interpretating each statement ending with ( ; ) one by one at a time. At the end of the stored procedure, that is After END statement, new delimiter ( // ) is applied to indicate the end of the stored procedure. The last statement DELIMITER ; restores the MySql delimiter back to standard one. Although to create a stored procedure in MySql this DELIMITER is not mandatory.
* CREATE PROCEDURE statement is used to create a new stored procedure. The name of the stored procedure is specified after CREATE PROCEDURE statement. The name is followed by parenthesis, same as we do with any function name in most of the languages. In the above example name of procedure is *GetAllEmployees* and it is followed by *()*as a part of the syntax.
* The body of the stored procedure is written between BEGIN and END statement. Declarative SQL statements are written in the body to handle business logic. As in above procedure we using SELECT statement to query the data from *Employeeinfo* table.

**Calling Stored Procedure** A CALL statement is used to call any stored procedure. CALL followed by stored procedure name. Below is the general syntax to call a stored procedure in left column and actual SQL statements by which our *GetAllEmployees()*is called in right column.

|  |  |
| --- | --- |
| Syntax | Actual Statements |
| CALL [*Stored Procedure Name*] | CALL *GetAllEmployees()* |

  If CALL *GetAllEmployees()*is executed, the body of the procedure would be executed and as result we will get all the records from *Employeeinfo*table. Till now we have learned how to create any stored procedure using CREATE PROCEDURE Statement and how to call them using CALL statement to see the result. Now let’s see further about Variables and parameters in MySql Stored Procedure. Stored Procedure Variables A variable is local to a stored procedure, and it is used to store any intermediate results while execution of stored procedure body. **Declare Variable** A variable must be declared before using it. A DECLARE statement is used to declare any variable, followed by variable name, its type and its default value if any. We are going to create a variable named *MyInt*of int type. Below is the syntax and actual statements of a variable declaration.

|  |  |
| --- | --- |
| Syntax | Actual Statements |
| DECLARE [*Variable Name*] [*Data Type*] DEFAULT [*Default value*] | DECLARE *MyInt INT*Default 0 |

  Now let’s understand each statement in detail

* DECLARE keyword followed by variable name. Variable name must follow the naming rules specified by MySql.
* Secondly, Data Type of the variable should be specified and its size. Any MySql data type such as Int, Varchar, Datetime etc. goes with variable.
* DEFAULT keyword assigns any default value to the variable. For example a default value of 0 is assigned to *MyInt*variable. If default value is not assigned, then the initial value of a variable is NULL.

More than one variable of same type can be declared in one statement as follows. DECLARE *MyInt1*, *MyInt2* INT DEFAULT 0 **Assign Variable** Once a variable is declared, SET statement is used to assign any other value to that variable. We would assign the value of ’10’ to the previously declared *MyInt* variable. Below is the syntax in left column to use SET statement and in the right column the actual example.

|  |  |
| --- | --- |
| Syntax | Actual Statements |
| SET [*Variable Name*] *Value* | SET *MyInt* =10 |

Value of *MyInt* after the execution of this statement would be 10. **Scope of Variables** A variable declared between any BEGIN and END statement would lose its scope after END statement. The variable with same name could be declared between two different sets of BEGIN and END statements. A variable beginning with @ symbol is session variable, and its remains in scope till session ends. **Conclusion**In this blog, we have learned basics of stored procedure, what are stored procedures? How to create them? What are advantage and disadvantage of stored procedures? How to call them? Hopefully this blog will encourage you to used stored procedures to solve your relevant problems in future.

# Difference between Class.forName() and ClassLoader.loadClass() methods in Java?

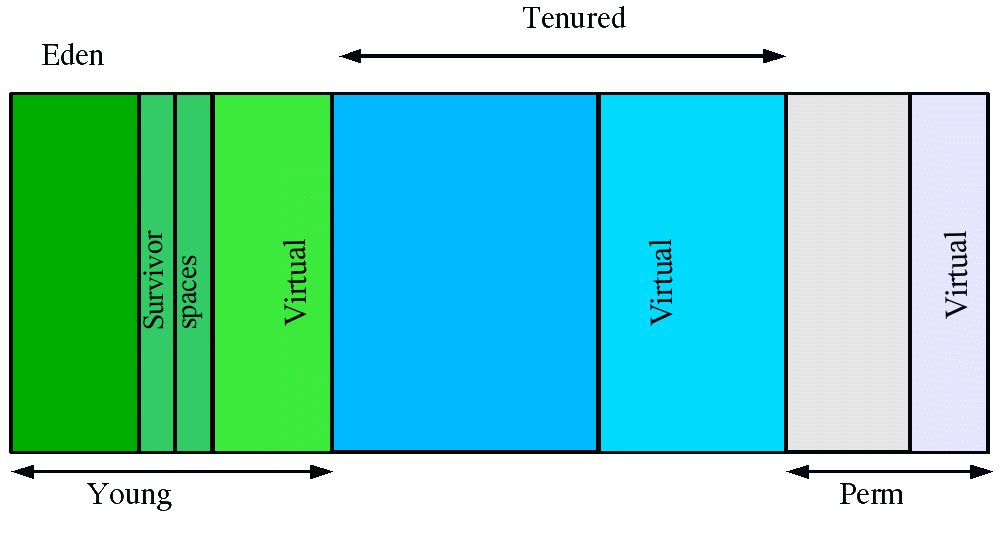
This is one of the most frequently asked questions in Java interviews. This question describes how deeply you know about class loading, dynamic class loading mechanism and class initialization. If you are interested to know about [class loading mechanism in Java](http://www.somanyword.com/2014/04/classloading-mechanism-in-java/), please read my article [here](http://www.somanyword.com/2014/04/classloading-mechanism-in-java/). Both Class.forName(String className) and ClassLoader.loadClass(String className) methods try to dynamically load class with given string class name and both methods return java.lang.Class objects for given string class name. But now question arise that if both methods have same behavior than what is the difference between them? There is one visible difference is that, where Class.forName(String className) method is static method of java.lang.Class class, in other hand, ClassLoader.loadClass(String className) method is instance method (non-static method). Therefore, if you want to use and call ClassLoader.loadClass(String className) method, you need any java.lang.ClassLoader instance to load any class with this method. One another difference is ClassLoader used by both methods to load given class. Where Class.forName(String className) method use the same ClassLoader to load the class, which is used by it's caller. For example, if you have created a class XYZ in which you have used Class.forName(String className) method to load any given class, in this case, Class.forName(String className) method will use same ClassLoader with which XYZ class has loaded. Whereas ClassLoader.loadClass(String className) method use that ClassLoader on which ClassLoader instance this method is called. If you are concerned about any specific or your own ClassLoader to load any class then you should use ClassLoader.loadClass(String className) method to load any given class. Let us discuss one major difference between both methods. The major difference between both methods is class initialization after loading any given class. Class.forName(String className) method load given class dynamically and initialize it statically. It initialize all static fields of this class and it's super class static fields recursively. Whereas ClassLoader.loadClass(String className) method load the class dynamically but delay the initialization of given class. Therefore, choice of use from both methods depends on situations, if you know, static initialization of any class is costly, then you could choose delay initialization using ClassLoader.loadClass(String className) method. But there is one more method in java.lang.Class class, in which you can specify your own ClassLoader to load any class and delay class initialization as well. There is one method with three parameters in which you can specify your ClassLoader and set Boolean flag, whether you want to initialize that class or not. For example, Class.forName(String className, boolean initialize, ClassLoader loader). This method can be use for more flexibility and customization like initialization and ClassLoader. I hope now you are bit more clear about differences in both methods which generally look alike. If you know any more differences please share with us.

# How memory management (Garbage Collection) works in Java and difference types of Garbage Collectors in Java

**Introduction** Garbage Collector is heart of JVM and Java language. Because this is one the most important features provide by Java for memory management. Garbage collection has played a significant role in popularity of Java in software industries. Garbage collector automatically de-allocates (free) memory after reclaiming dead objects from heap and release these reclaimed memory back to JVM to allocate new objects. Dead objects are those objects which are not in use and further referenced in program. Other programming languages before Java like C and C++, there are no garbage collectors kind of things available to manage memory. Memory management is required to be done explicitly by developer. This is the programmer’s responsibility to initialize new object and de-allocate memory after use. The complexity of that task leads to many common errors that can cause unexpected or erroneous program behavior and crashes. As a result, a large proportion of developer time is often spent debugging and trying to correct such errors. That is why garbage collection is one of the major reasons of popularity of Java. Before going deep to understand different types of garbage collectors, you must aware about JVM heap memory isolation in generation. There are basically three types of isolated generations on heap, which are young generation, old generation and permanent generation. But there are few terms, which are required to be known before reading this blog. These terms are mentioned below:

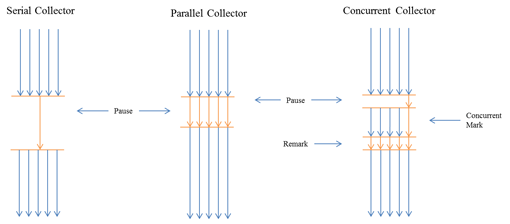
1. **Live Objects:**Live objects are those objects which are still referenced in program and could be use later on in program.
2. **Dead Objects:**Dead objects are those objects which are no longer used and referenced in program or out of scope of program. These dead objects are termed as garbage.
3. **Fragmentation:**This is the limitation of garbage collection. When the memory for garbage objects is freed, the free space may appear in small chunks in various areas such that there might not be enough space in any one contiguous area to be used for allocation of large object.
4. **Compaction:**This is an approach to eliminating fragmentation from memory and known as compaction. Compaction moves all live objects to one side of memory and other side of memory is free to allocate new objects.
5. **Throughput:**The percentage of total time not spent on garbage collection, considered over long periods of time. In other word, effectively time spent by application without garbage collection overhead.
6. **Garbage Collection Overhead:** The inverse of throughput, that is, the percentage of total time spent in garbage collection.
7. **Pause Time:** The length of time during which application execution is stopped while garbage collection is occurring.
8. **Stop-The-World:** Stop-the-world is term used for time period, while application is totally halt for garbage collection.
9. **Frequency of Collection:** How often garbage collection occurs, relative to application execution.
10. **Footprint:** Measurement of heap size, which describes used and free space available in heap.
11. **Promptness:**The time between when an object becomes garbage and when the memory becomes available.

**Generations** Heap memory is divided into multiple parts which are termed as generations. Garbage collectors works differently on different generations. There are mainly three generations isolated into heap. These generations have different memory pools holding objects of different ages. Different garbage collector algorithms can be used to perform garbage collection. There are few important observations which are good to know that most allocated objects are not referred for much longer and they died at young stage. Only few objects get survive from this stage and get alive. Few references from older to younger objects also exist. Three generations are mentioned bellow, which holds allocated objects of different ages:



1. **Young Generation (Eden + 2 Survivors):** Most of new objects are allocated in this generation (Eden). Young generation garbage collection occurred relatively more frequently than other generations and this is faster than other generation because young generation is comparatively small and most of objects are no longer referred. Only few survive objects are promoted to next survivor area. There are two survivor areas, which hold those objects which are not garbage and survived from garbage collection. Objects those survived some number of garbage collections are eventually promoted or tenured to the old generation.
2. **Old Generation (Tenured):** All survived objects from young generation are promoted to old generation (Tenured) and some large objects which are directly allocated in old generation. Old generation is larger than young generation. Garbage collection is less frequent in this generation and also takes significantly longer to complete collection.
3. **Permanent Generation (PermGen):** PermGen contains all relevant data or meta-data of JVM such as classes and methods objects. PermGen is divided into two areas, read-only and read-write areas, which contain different types of objects.

**Fast Memory Allocation** As you know garbage collection and memory allocation to objects are equally important processes. Therefore, objects allocation must be as faster as we expect garbage collection. There are few techniques used to fast the allocation, one is ***Bump-The-Pointer*** technique. In this technique one pointer always track end of the previously allocated object and check whether new object is satisfying or fit for available memory, if yes, update the pointer and initialize the object. In other hand, for multithreaded application, objects allocation operation need to be multithread safe. If global locks were used to ensure this, allocation into a generation would degrade performance. Therefore, to solve this issue, another technique is used for multithreaded applications. This technique is called ***Thread-Local-Allocation-Buffers* (TLABs)**. This improves multithreaded allocation throughput by giving each thread its own buffer (i.e., a small portion of the generation) from which to allocate. Since only one thread can be allocating into each TLAB, allocation can take place quickly by utilizing the Bump-The-Pointer technique, without requiring any locking. Only infrequently, when a thread fills up its TLAB and needs to get a new one, must synchronization be utilized. Several techniques to minimize space wastage due to the use of TLABs are employed. For example, TLABs are sized by the allocator to waste less than 1% of Eden, on average. The combination of the use of TLABs and linear allocations using the Bump-The-Pointer technique enables each allocation to be efficient, only requiring around 10 native instructions. **Design and Algorithms** Garbage collection use different types of designs and algorithms, which would be better to know before drill down garbage collector types.



1. **Serial versus Parallel:**In *serial* collector, only one thing happens at a time. Even when multiple CPUs are available, only one is utilized to perform to collection. When *parallel* collection is used, garbage collection is split into parts and those subparts are executed simultaneous on different CPUs.
2. **Concurrent versus Stop-The-World:**When *Stop-The-World*garbage collection takes place, it suspends application execution and performs garbage collection. Whereas, *concurrent*garbage collection and application execution are work concurrently. *Concurrent* garbage collection does most of its works concurrently with application execution, that is why, its throughput is much higher that Stop-The-World. But occasionally its do few Stop-The-World pauses during concurrent garbage collection. *Stop-The-World*is simpler than *concurrent*garbage collection because *Stop-The-World* stop application execution and freeze heap and objects do not change during collection. In other hand, *concurrent*garbage collection need extra care during collection because might be objects which are going to collect are updated at time of collection by application.
3. **Compacting versus Non-compacting versus Copying:**When garbage collector analysis which objects are live and garbage, then all live objects are moved towards start of memory, which is called *compaction* and rest of objects (garbage) reclaimed to release the memory. After *compaction* it is easy to allocate new objects in memory because Bump-The-Pointer technique track last memory available after compacted objects. Opposite to compacting collection, a non-compacting collection does not move live objects, instead of this it remove all garbage from memory and leave fragmented space in heap. That is why non-compacting collection is much faster than compacting collection but it is expensive to allocate new objects on heap. A third alternative is a *copying* collector, which copies or evacuates live objects to a different memory area. The benefit is that source area can then be considered empty and available for fast and easy subsequent allocations. But the drawback is additional time required for copying and the extra space that may be required.

**Types of Garbage Collectors** There are five types of garbage collectors available in JVM.

**1. Serial Collector:**With the serial collector, both young and old collections are done serially (using single CPU), in a stop-the-world fashion. That is application execution is halted while collection is taking place. Throughput of serial collector is very low with high pause time, because during collection every task is done with stop-the-world fashion. [Click here to read more about serial collector.](http://www.somanyword.com/2014/01/serial-garbage-collector-in-java/)

**2. Parallel Collector:**You know these days many machines run on lots of memory and multiple CPUs. Therefore parallel collector take advantage of multiple CPUs with parallel execution of garbage collection with multiple threads. Therefor throughput of parallel collector is better than serial collector because most of work are done parallel rather that idle the CPU when one garbage collection is working. [Click here to read more about parallel collector](http://www.somanyword.com/2014/01/parallel-garbage-collector-in-java/).

**3. Parallel Compacting (Old) Collector:** Parallel compacting collector was introduced since Java 5. Parallel compacting collector works similar way as parallel collector work, but it use one new algorithms to compact the live objects in one side to remove fragmentation from heap. Parallel compacting collector is collect garbage in Stop-The-World fashion and compaction of live objects are done parallel with this. This is much more complex that parallel collector. Throughput of parallel compacting is less than parallel collector but new object allocation is faster. [Click here to read more about parallel compacting collector](http://www.somanyword.com/2014/01/parallel-compacting-garbage-collector-in-java/).

**4. Concurrent Mark-Sweep (CMS) Collector:**For many applications those required high throughput and less pause time, concurrent mark-sweep (CMS) is much beneficial for these types of applications. CMS collector is inspired by parallel collector but it also work concurrently with application throughput as well. Therefore, it introduce very short pause time (Stop-The-World) in application execution and provide good throughput. [Click here to read more about Concurrent Mark Sweep garbage collector](http://www.somanyword.com/2014/01/concurrent-mark-sweep-cms-garbage-collector-in-java/).

**5. Garbage First (G1) Collector:**The Garbage First Garbage Collector (G1 GC) is latest garbage collection implementation in Java HotSpot VM. This garbage collector (G1) in introduced since Java 6 for testing phase. But since Java 7, Garbage First (G1) is officially supported. G1 is low-pause, server style garbage collector, which is design to achieve more throughputs from application. G1 determine the areas where large amount of garbage (dead objects) exists, G1 collect those area first. That is why, this garbage collector is known as Garbage First (G1). Throughput of G1 is much higher with low pause time. G1 algorithm does not work on generations (young, old and permanent) as previous algorithms works. Instead of it G1 break down whole heap in small equal sizes memory reasons. These individual reason hold objects of different ages. Now these individual reason logically hold young, survivor and tenured objects.

**Conclusion** Now you know garbage collection is not just a feature of JVM. It is one of most significant feature of Java and one of the biggest thing to play a big role in success of Java. With deep understanding of objects age  generations on heap and all available garbage collector types, you can give a new life and good performance to your software. With the help of this article now you can select your own garbage collector, which suites you best for your application. I hope you will take full advantage of this GC customization.

# Difference between Enumeration, Iterator and ListIterator in Java

Enumeration, Iterator and ListIterator are used to iterate and fetch elements from its implemented classes. But there are few differences between these implementation. Let us discuss some of the important differences over here.

1. Enumeration interface implement on legacy classes. Iterator is implemented on all Java collection classes ( Set, List & Map ). Iterator allows safe removal of elements from collection during iteration. ListIterator is implemented only for List type classes.

2. Enumeration is read only iteration. Elements are not allowed to remove during iteration. Iterator & ListIterator can remove element from collection along with iteration. There is remove() method to remove element from collection.

3. Enumeration uses elements() method. Iterator uses iterator() method. ListIterator uses listIterator() method.

4. Enumeration & Iterator can traverse unidirectional (forward directional) only. Whereas ListIterator can traverse bidirectional, mean forward and backward both directions.

5. Iterator & ListIterator iteration are fail-fast iteration. But Enumerations returned by the methods of classes like HashTable, Vector are not fail-fast. These classes provide safety to its data with synchronization. The nextElement() method that locks or synchronize the current Vector object. But synchronization slow down the performance of classes.

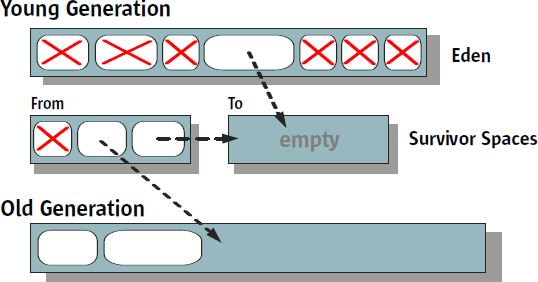
# What is difference between fail-fast and fail-safe iterations in Java?

Whenever you iterate any collection class to retrieve data from this, it is necessary to think about what could be the implication of iteration in multithreading programming and what kind of iteration that class supports? After understanding this you can write you code according to your requirement and bug free. There are two types of iterators fail-fast and fail-safe. Perhaps, initially these both terms would look little complicated in listening. But believe me this is not like that. **Fail-fast Itreator** Fail-fast iteration as name is suggesting that fail-fast type iteration fails as soon as it sees any modification in collection during iteration. There could be any kind of changes addition, removal or updating of elements in collection. For example if one thread (Thread-1) is iterating on collection and one another thread (Thread-2) added one more element in collection and modified it actual structure on which Thread-1 is iterating. In this case Thread-1 can fail and throw ConcurrentModificationException. The fail-fast behavior of iterators are implemented by keeping Modification Count (modCount) and Expected Modification Count (expectedModCount) of elements present in collection at time of iteration. If that iteration realize any miss match in modCount and expectedModCount during iteration, then it try to fail that iteration as soon as it detect that changes and throw ConcurrentModificationException. That is why it is called fail-fast iteration. **Java docs of fail-fast iterators says** “Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw {@code ConcurrentModificationException} on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.” It means there is no guaranty that fail-fast behavior of iterators will immediate throw an exception if medication will happened in collection. There might be chance it does not throw exception or throw exception bit later. Because modifications count (modCount) is not volatile variable, on which basis iterator check modification. There might be chance modCount give stale value of modCount and would not able to detect modification. **Fail-safe iterator** Fail-safe iterator behave opposite to fail-fast iterator. It does not throw any exception if collection structure gets modified during iteration. Fail-safe iteration is supported in thread safe classes of java.util.concurrent package. Like CopyOnWriteArrayList, ConcurrentHashMap etc. **Java docs of fail-safe iterators says** “This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don't want to synchronize traversals, yet need to preclude interference among concurrent threads. The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw*ConcurrentModificationException*. The iterator will not reflect additions, removals, or changes to the list since the iterator was created. Element-changing operations on iterators themselves (*remove(), set(), and add()*) are not supported. These methods throw*UnsupportedOperationException*.” Means iterator of these classes iterates on snap-shot (clone) of collection, which does not get modified if any modification happens on collection. Therefore, fail-safe iterators never throw ConcurrentModificationException in Java.

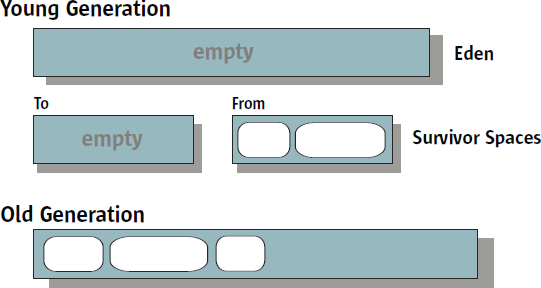
# Serial Garbage Collector in Java

**Serial Collector:** Garbage collection can be done with different types of algorithms. These algorithms have different kind to approaches and performance. Therefore algorithm selection depends on application types and its throughput requirements. On the basis of these algorithms, garbage collection designers have designed different types of garbage collectors. Serial collector is one of them. Serial collector is one of simplest garbage collector, Serial collector collects both young and old generation garbage serially (using a single CPU), in a stop-the-world fashion. Its means, application execution is halted while collection is taking place. Therefore throughput of serial collector is very less compare to other garbage collectors. If you are not aware about garbage collection fundamentals and object generations, then I will recommend you to first read my blog "[How memory management (Garbage Collection) works in Java and difference types of Garbage Collectors in Java?](http://www.somanyword.com/2014/01/how-memory-management-garbage-collector-works-in-java-and-difference-types-of-garbage-collectors-in-java/)"

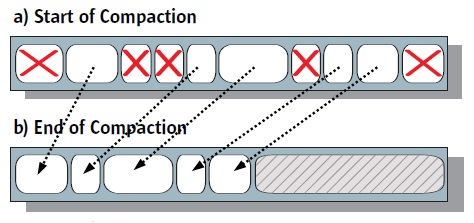
**Young Generation Collection Using the Serial Collector:** As you know most of new objects are allocated in Eden part of young generation. When minor collection happens, garbage collector identifies all live objects from Eden and From space. These live objects in Eden are copied to the initially empty survivor space (labeled To ), except for ones that are too large to fit comfortably in the To space. Such objects are directly copied to the old generation (Tenured). The live objects in the occupied survivor space (labeled From) that are still relatively young are also copied to the other survivor space (labeled To), while objects that are relatively old are copied to the old generation. It is important to understand that one survivor space from both will always be empty to hold relatively survived live objects from collections. If the To space becomes full, the live objects from Eden or From that have not been copied to To space will directly copied to old generation (Tenured), regardless of how many young generation collections they have survived. Any objects remaining in Eden or the From space after copied the live objects are considered as dead objects (garbage), and they do not need to be examined. These garbage objects are marked with an 'X' in the figure, though in fact the collector does not examine or mark these objects.



After a young generation collection is complete, both Eden and the formerly occupied survivor space are empty and only the formerly empty survivor space contains live objects. At this point, the survivor spaces swap roles. Now To space will swap it's label as From and vice-versa.



**Old Generation Collection Using the Serial Collector:** Serial collector collects old and permanent generations with Mark-Sweep-Compact collection algorithm. In the mark phase, the collector identifies which objects are still live and rest of object are garbage. The sweep phase “sweeps” over the generations and remove all garbage. In compact phase, the collector performs sliding compaction, means sliding the live objects towards the beginning of the old generation space (and similarly for the permanent generation), leaving any free space in a single contiguous chunk at the opposite end. The compaction allows any future allocations into the old or permanent generation to use the fast, bump-the-pointer technique.



**When to Use the Serial Collector:** The serial collector is choice of those applications which run on client-style machines and that do not have a requirement for low pause times. Because throughput of serial collector is low. On today’s hardware which are very fast, the serial collector can efficiently manage a lot of nontrivial applications with 64MB heaps and relatively short worst-case pauses of less than half a second for full collections. **Serial Collector Selection** In the Java 5 release, the serial collector is automatically chosen as the default garbage collector on machines that are not server-class machines. On other machines, the serial collector can be explicitly selected by using the  -XX:+UseSerialGC command line option.

# Parallel Garbage Collector in Java

**Parallel Collector:** Garbage collection with parallel collector is much faster that serial collector. As you know today almost all Java applications run on machines with a lot of physical memory and multiple CPUs. The parallel collector, also known as the throughput collector, was developed in order to take advantage of available CPUs rather than leaving most of CPU processes idle. Therefore throughput of parallel collector is much better than serial collector. If you are not aware about garbage collection fundamentals and object generations, then I will recommend you to first read my blog “[How memory management (Garbage Collection) works in Java and difference types of Garbage Collectors in Java?](http://www.somanyword.com/2014/01/how-memory-management-garbage-collector-works-in-java-and-difference-types-of-garbage-collectors-in-java/)”

**Young Generation Collection Using the Parallel Collector:** The parallel collector uses a parallel version of the young generation collection algorithm utilized by [serial collector](http://www.somanyword.com/2014/01/serial-garbage-collector-in-java/) as shown in figure. If you do not know collection process of serial collector, please first read my blog “[Serial Garbage Collector in Java](http://www.somanyword.com/2014/01/serial-garbage-collector-in-java/)”. It is still a Stop-The-World and copying collector, but performing the young generation collection in parallel, using many CPUs, decreases garbage collection overhead and hence increases application throughput. Figure illustrates the differences between the serial collector and the parallel collector for the young generation.

**Old Generation Collection Using the Parallel Collector:** Old generation garbage collection for the parallel collector is done using the same serial mark-sweep-compact collection algorithm as the [serial collector](http://www.somanyword.com/2014/01/serial-garbage-collector-in-java/).

**When to Use the Parallel Collector:** Those applications running on machines with one than one CPUs and do not have pause-time constraints can use parallel collector for garbage collection. Examples of applications for which the parallel collector is often appropriate, those that do batch processing, billing, payroll, scientific computing, and so on. You may want to consider choosing the parallel-compacting collector (described next) over the parallel collector, since the former performs parallel collections of all generations, not just the young generation.

**Parallel Collector Selection** In the Java 5 release, the parallel collector is automatically chosen as the default garbage collector on server-class machines. On other machines, the parallel collector can be explicitly selected by using the -XX:+UseParallelGC command line option.

# Parallel Compacting Garbage Collector in Java

**Parallel Compacting Collector:** Parallel compacting collector was introduced since Java 5. The difference between parallel compacting collector and the parallel collector is that parallel compacting collector uses a new algorithm (compacting) for old generation garbage collection, that is why this collector is also known as Parallel Old Collector, but it work similarly for young generation as parallel collector. The parallel compacting collector takes an overhead to compact the fragmented memory of old generation. But this process helps in fast memory allocation to new objects with bump-the-pointer technique. If you are not aware about garbage collection fundamentals and object generations, then I will recommend you to first read my blog “[How memory management (Garbage Collection) works in Java and difference types of Garbage Collectors in Java?](http://www.somanyword.com/2014/01/how-memory-management-garbage-collector-works-in-java-and-difference-types-of-garbage-collectors-in-java/)”

**Young Generation Collection Using the Parallel Compacting Collector:** Young generation garbage collection for the parallel compacting collector is done using the same algorithm as that for young generation collection using the parallel collector. If you want to know about parallel collector, read my blog “[Parallel Garbage Collector in Java](http://www.somanyword.com/2014/01/parallel-garbage-collector-in-java/)”

**Old Generation Collection Using the Parallel Compacting Collector:** The old and permanent generations collections using parallel compacting collector are work in stop-the-world, mostly parallel fashion with sliding compaction. The parallel compacting collector utilizes three phases. First, each generation is logically divided into fixed-sized regions. In the marking phase, the initial set of live objects directly reachable from the application code is divided among garbage collection threads, to mark these objects parallel, and then all live objects are marked in parallel. As an object is identified as live, the data for the region it is in is updated with information about the size and location of the object. The summary phase operates on regions, not objects. Due to partially compactions from previous collections, it is typical that some portion of the left side of each generation will be dense, containing mostly live objects. The amount of space that could be recovered from such dense regions is not worth the cost of compacting them. So the first thing the summary phase does is examine the density of the regions, starting with the leftmost one, until it reaches a point where the space that could be recovered from a region and those to the right of it is worth the cost of compacting those regions. The regions to the left of that point are referred to as the dense prefix, and no objects are moved in those regions. The regions to the right of that point will be compacted, eliminating all dead space. The summary phase calculates and stores the new location of the first byte of live data for each compacted region. Note: The summary phase is currently implemented as a serial phase; parallelization is possible but not as important to performance as parallelization of the marking and compaction phases. In compaction phase, the garbage collection threads use the summary data to identify regions that need to be filled, and the threads can independently copy data into the regions. This produces a heap that is densely packed on one end, with a single large empty block at the other end.

**When to Use the Parallel Compacting Collector:** As with the parallel collector, the parallel compacting collector is beneficial for applications that are run on machines with more than one CPU. In addition, the parallel operation of old generation collections reduces pause times and makes the parallel compacting collector more suitable than the parallel collector for applications that have pause time constraints. The parallel compacting collector might not be suitable for applications run on large shared machines, where no single application should monopolize several CPUs for extended periods of time. On such machines, consider either decreasing the number of threads used for garbage collection (via –XX:ParallelGCThreads=n command line option) or selecting a different collector.

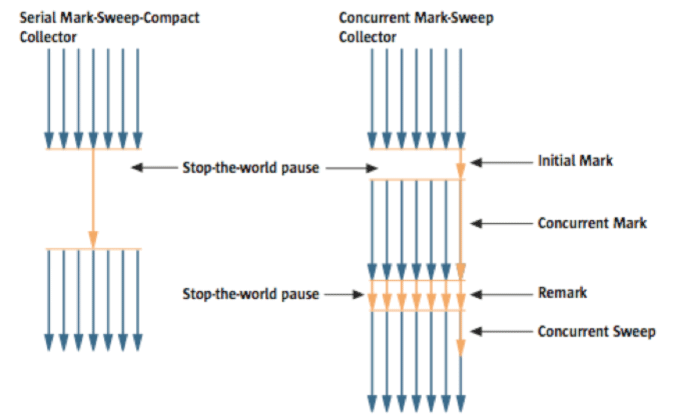
**Parallel Compacting Collector Selection:** If you want the parallel compacting collector to be used, you must select it by specifying the command line option -XX:+UseParallelOldGC.

# Concurrent Mark Sweep (CMS) Garbage Collector in Java

**Concurrent Mark-Sweep (CMS) Collector:** Concurrent-Mark-Sweep collector is most popular garbage collector of Java. CMS collector is popular for its better throughput and less pause time. Because for many applications, end-to-end throughput is not as important as fast response time. For example gaming applications need fast response time to make their gaming experience better, if any game hang for a second only, it lost its charm. As you know young generation collections do not typically cause long pauses, because of its small size and less amount of live objects survived. However, old generation collections is uncertain, can impose long pauses, especially when large heaps are involved. To address this issue, the Java HotSpot JVM includes a collector called the concurrent-mark-sweep (CMS) collector, also known as the low-latency collector. If you are not aware about garbage collection fundamentals and object generations, then I will recommend you to first read my blog “[How memory management (Garbage Collection) works in Java and difference types of Garbage Collectors in Java?](http://www.somanyword.com/2014/01/how-memory-management-garbage-collector-works-in-java-and-difference-types-of-garbage-collectors-in-java/)”

**Young Generation Collection Using the CMS Collector:** The CMS collector collects the young generation in the same manner as the [parallel collector](http://www.somanyword.com/2014/01/parallel-garbage-collector-in-java/). Because young generation collection do not typically take long pauses because of its small size. There is one more reason, as you know in young generation large amount of objects become garbage and get de-allocated at very first time of collection, which do not take much time and introduce very short pause.

**Old Generation Collection Using the CMS Collector:** Concurrent Mark Sweep collector play significant role in old generation garbage collection. CMS mainly works on four phases. Initial Mark, Concurrent Mark, Remark and Concurrent Sweep.



Most of the collection of the old generation using the CMS collector is done concurrently with the execution of the application. A collection cycle for the CMS collector starts with a short pause, called the initial mark, that identifies the initial set of live objects directly reachable from the application code. Then, during the concurrentmarking phase, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase. To handle this, the application stops again for a second pause, called remark, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency. At the end of the remark phase, all live objects in the heap are guaranteed to have been marked, so the subsequent concurrent sweep phase reclaims all the garbage that has been identified. Figure illustrates the differences between old generation collection using the serial mark-sweep-compact collector and the CMS collector. Since some tasks, such as revisiting objects during the remark phase, increase the amount of work the collector has to do, its overhead increases as well. This is a typical trade-off for most collectors that attempt to reduce pause times. The CMS collector is the only collector that is non-compacting. That is, after it frees the space that was occupied by dead objects, it does not move the live objects to one end of the old generation. This saves time, but since the free space is not contiguous, the collector can no longer use a simple pointer indicating the next free location into which the next object can be allocated. Instead, it now needs to employ free lists. That is, it creates some number of lists linking together unallocated regions of memory, and each time an object needs to be allocated, the appropriate list (based on the amount of memory needed) must be searched for a region large enough to hold the object As a result, allocations into the old generation are more expensive than they are with a simple bump-the-pointer technique. This also imposes extra overhead to young generation collections, as most allocations in the old generation occur when objects are promoted during young generation collections. Another disadvantage the CMS collector has is a requirement for larger heap sizes than the other collectors. Given that the application is allowed to run during the marking phase, it can continue to allocate memory, thereby potentially continuing to grow the old generation. Additionally, although the collector guarantees to identify all live objects during a marking phase, some objects may become garbage during that phase and they will not be reclaimed until the next old generation collection. Such objects are referred to as floating garbage. Finally, fragmentation may occur due to lack of compaction. To deal with fragmentation, the CMS collector tracks popular object sizes, estimates future demand, and may split or join free blocks to meet demand. Unlike the other collectors, the CMS collector does not start an old generation collection when the old generation becomes full. Instead, it attempts to start a collection early enough so that it can complete before that happens. Otherwise, the CMS collector reverts to the more time-consuming stop-the-world mark-sweep-compact algorithm used by the parallel and serial collectors. To avoid this, the CMS collector starts at a time based on statistics regarding previous collection times and how quickly the old generation becomes occupied. The CMS collector will also start a collection if the occupancy of the old generation exceeds something called the initiating occupancy. The value of the initiating occupancy is set by the command line option –XX:CMSInitiatingOccupancyFraction=n, where n is a percentage of the old generation size. The default is 68. In summary, compared to the parallel collector, the CMS collector decreases old generation pauses sometimes dramatically at the expense of slightly longer young generation pauses, some reduction in throughput, and extra heap size requirements.

**Incremental Mode:** The CMS collector can be used in a mode in which the concurrent phases are done incrementally. This mode is meant to lessen the impact of long concurrent phases by periodically stopping the concurrent phase to yield back processing to the application. The work done by the collector is divided into small chunks of time that are scheduled between young generation collections. This feature is useful when applications that need the low pause times provided by the concurrent collector are run on machines with small numbers of processors (e.g., 1 or 2). For more information on usage of this mode, search for “Tuning Garbage Collection with the 5.0 Java Virtual Machine”.

**When to Use the CMS Collector:** Use the CMS collector if your application needs shorter garbage collection pauses and can afford to share processor resources with the garbage collector when the application is running. (Due to its concurrency, the CMS collector takes CPU cycles away from the application during a collection cycle.) Typically, applications that have a relatively large set of long-lived data (a large old generation), and that run on machines with two or more processors, tend to benefit from the use of this collector. An example would be web servers. The CMS collector should be considered for any application with a low pause time requirement. It may also give good results for interactive applications with old generations of a modest size on a single processor.

**Concurrent Mark Sweep Collector Selection:** If you want the CMS collector to be used, you must explicitly select it by specifying the command line option -XX:+UseConcMarkSweepGC. If you want it to be run in incremental mode, also enable that mode via the –XX:+CMSIncrementalMode option.

# What is the difference between Error and Exception?

This is one of the most frequently asked questions in Java interview. Error and Exception, both are very important features for any programming language. That is why, interviewer want to know how much you know about them and how would you handle them. Both Error and Exception are sub-classes of Throwable class in Java. All sub-classes of Error class are known as errors and all sub-classes of Exception class are known as exceptions. Let us discuss something more about them.

**Error:** An error is an irrecoverable condition occurring at runtime. For example, NoClassDefFoundError, OutOfMemoryError etc. A program throw NoClassDefFoundError error when JVM doesn't find any class at runtime, which was there at the time of compilation. While OutOfMemoryError error occur when heap is almost full and does not have efficient memory to allocate new objects or  hold existing objects. In both cases, these situations are irrecoverable. Whenever any error happens during program execution that means something really goes wrong and you need to understand error carefully to resolve it. A program can catch error in catch block, but cannot repair error at runtime. We can write any recoverable alternative for this. If an error occurs, program get halt and stop responding. 

**Exception:**An exception is the condition, which occur due to bad input or programmatically mistakes. For example, NullPointerException, FileNotFoundException etc. A program throw NullPointerException when any null reference is used in program, which is a programmatically mistake, while FileNotFoundException is thrown when program is trying to load or used any file which does not exist, which is bad input. An exception can be caught in catch block and can be repair at runtime by any recoverable alternative in catch block. If an exception occurs, program executes alternative procedures written in catch block. These are few differences between error and exception, which tell us how to deal these situations.

# What is difference between Comparator and Comparable interfaces in Java?

Both Comparator and Comparable interfaces are designed for objects comparison and to help in objects collection sorting. If any class has implemented Comparable interface, objects collection of that class can be sort with java.util.Collections utility. But both interfaces have differences in their purpose of implementations. Let us discuss some differences in both interfaces.

**1.** First basic difference of both interfaces is that both interfaces are part of different packages. Comparator is defined in java.util package while Comparable is defined in java.lang package.

**2.** A class implements Comparable interface for its default comparison behavior, which could be based on any existing attributes of that class . While a programmer can implement Comparator interface, if programmer intent to sort any class objects collection by its own choice of attributes, which can not be done by default comparison behavior implemented with Comparable . For example, Lets have an example of Employee class which have member attributes or properties like Employee Name, Age, Salary and Department. If Employee class wants default sorting on Employee name attribute then it must implement Comparable interface. In other hand if any other programmer later on want to sort Employee objects on the basis of Salary, then user can create a new class with implementation of Comparator interface. You can see coding example below for better understanding of this scenario.

**3.** Comparable interface has a method with public int compareTo(Object o); signature, which returns positive, zero or negative integer number in corresponding situation where current object (this object) is greater than, equal or less than the given object. In case of Comparable interface comparison happens with current object. While Comparator interface has a method with public int compare(Object obj1, Object obj2); signature, which also returns positive, zero or negative integer number in corresponding situation where first object is greater than, equal to or less than second object. In case of Comparator interface, comparison happens in two given objects, it never happens with current object.

**4.** It is important to understand the significance of both interfaces. If you are not the author of any class and you want to sort collection of that class by those properties which are not part of default comparison behavior of that class, in this case you can use Comparator interface implementation to solve your problem. Which helps you to write your desired comparison behaviour.

**5.** Comparable implemented classes can be sorted with Collections.sort(List list); utility. Where Comparator implementation is required to pass in sorting utility. For example, Collections.sort(List list, Comparator c);

# How to design Singleton Pattern using Java?

Singleton pattern is one most commonly used pattern in every types of software application developments. Therefore every interviewer would like to ask about this pattern first. This is really simple design pattern to create but you need to take care of some common things to design this pattern. Before going deep, let us concentrate on singleton pattern definition.

***"Singleton pattern provides the possibility to control the number of instances (mostly one) that are allowed to be create."***

As you can see definition of singleton pattern describe singleton as special kind to class which allow only limited number of objects creation. Most of time singleton allows only one instance for whole application as its name is suggesting "Singleton" means single object. Only instance is used throughout application. **What is the significance of singleton?** Now question arise that where to use this pattern and what is significance of singleton pattern? Singleton is used where you want to control limited number of objects creation. Sometime applications have few heavy objects of particular classes, Therefore application cannot afford to create new heavy objects every time to perform any action. For example we use application context, session factory and database factory, which do not allow us to create any instance. When we ask for instance if no instance is available then it will create an instance and return, if is available and already create then it will return same instance again to use.

**How to create singleton class?** Before creating singleton class you need to take care few points about singleton. Let discuss these points.

1. Singleton class does not allow to create any instance directly. Therefore all constructors in class must be declared private, so that no client can create its new instance directly.
2. There must be a public static method which returns an object of same class. For example,  public static SingletonClass getInstance(); This method responsible to create and return an object of singleton class. If instance is already present (instance is not null) then it will return same instance otherwise it will create an instance and return it to client and keep it to further use in application.
3. One most important thing to arrange your code this way, so that ere must not be any change of creating second object of class.

To understand this you can see implementation of singleton class below **Coding Example** Here I would like to show you an example of File Logger, which would be a singleton class and will provide only single object throughout application to log any error.

**package** com.hawk.java.patterns;

/\*\*

\* FileLogger is a singleton class and will provide

\* only single object throughout application to log any error.

\*

\* @author SoManyWord

\*

\*/

**public** **class** FileLogger {

**private** **static volatile** FileLogger logger;

/\*\*

\* Prevent client for using any constructor and create object directly.

\*/

**private** FileLogger() {

}

/\*\*

\* Create and return singleton object.

\* @return FileLogger

\*/

**public** **static** FileLogger getInstance() {

**if** (logger == **null**) {

**synchronized** (FileLogger.**class**) {

**if** ( logger == **null** ) {

logger = **new** FileLogger();

}

}

}

**return** logger;

}

/\*\*

\* Log message to the text file.

\* @param message String message.

\*/

**public** **synchronized** **void** log(String message) {

//Write to log file.

}

}

This is coding example which would help you to understand singleton class internal design.

# How to create and implement custom annotations in Java?

Annotations are the most important feature of Java. There are lots more Java based frameworks available, which are working on annotations. For example JUnit, Hibernate, Spring etc. These few are most famous frameworks designed with the help of annotations. Therefore knowledge of annotations is quite necessary if you are working on Java. “Annotation are tags or meta-data, which can be inserted in to source code. So that it could be process at runtime and can take decision according to that annotations.” Java compiler understands couple of annotations, which we would discuss later in this article. But most important thing is to create own annotations or custom annotations. Lets see an example, how to create custom annotations in Java. Annotation can be define with same way as we define interface but in case of annotation @interface keyword is used. Let us figure out some important points about custom annotation

1. @interface keyword is used to create annotation

2. Only public or default access modifier can be used for annotation.

3. Only public or abstract access modifier can be used for attributes of Annotation.

4. Only primitive types, String, Class, Enum, Annotation and one dimensional arrays of mentioned types are permitted as attributes in annotations.

5. Default value can be defined for attributes.

6. Four types of meta-annotation are used for custom annotation. Which are @Documented, @Inherited, @Target and @Retention.

These meta-annotations descriptions are mentioned below in this article.

**package** com.hawk.java;

**import** java.lang.annotation.Documented;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Inherited;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*

\* @author SoManyWord.com

\*/

@Documented

@Inherited

@Target (ElementType.METHOD)

@Retention (RetentionPolicy.RUNTIME)

**public** @**interface** MethodMetaData {

**public** String methodName();

**public** **int** arguments() **default** 0;

**public** Class<? **extends** Exception> expected() **default** java.lang.Exception.**class**;

**public** String Author();

}

Code shown above is coding example of custom annotations. This is how you can create your own annotation in Java. Now its time to discuss more about meta-annotations used in custom annotation creation.  **Meta-annotations of Annotations** There are four types of meta-annotations used to create custom annotations. According to javadoc these meta-annotations are describe as mentioned bellow:

**@Documented:**Indicates that annotations with a type are to be documented by javadoc and similar tools by default. This type should be used to annotate the declarations of types whose annotations affect the use of annotated elements by their clients. If a type declaration is annotated with Documented, its annotations become part of the public API of the annotated elements.

**@Inherited:**Indicates that an annotation type is automatically inherited. If an Inherited meta-annotation is present on an annotation type declaration, and the user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, then the class's superclass will automatically be queried for the annotation type. This process will be repeated until an annotation for this type is found, or the top of the class hierarchy (Object) is reached. If no superclass has an annotation for this type, then the query will indicate that the class in question has no such annotation. Note that this meta-annotation type has no effect if the annotated type is used to annotate anything other than a class. Note also that this meta-annotation only causes annotations to be inherited from super classes; annotations on implemented interfaces have no effect.

**@Target:**Indicates the kinds of program element to which an annotation type is applicable. If a Target meta-annotation is not present on an annotation type declaration, the declared type may be used on any program element. If such a meta-annotation is present, the compiler will enforce the specified usage restriction. For example, this meta-annotation indicates that the declared type is itself a meta-annotation type. It can only be used on annotation type declarations rather than class, method or field. ElementType type enum is used to assign its values as mentioned in above example. For example @Target (value = ElementType.METHOD). ElementType enum has many more values available, use them accordingly.

**@Retention:**Indicates how long annotations with the annotated type are to be retained. If no Retention annotation is present on an annotation type declaration, the retention policy defaults to RetentionPolicy.CLASS. For Example: RetentionPolicy.RUNTIME Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.

**In-Built Annotations in Java** Java has some in-built annotations, which are known for Java compiler. There are three types of in-built annotations in Java. Let us discuss something more about these in-built annotations. According to javadoc these in-built annotations are describe as mentioned bellow:

**@Override:** Indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message.

**@Deprecated:** A program element annotated @Deprecated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists. Compilers warn when a deprecated program element is used or overridden in non-deprecated code.

**@SuppressWarnings:** Indicates that the named compiler warnings should be suppressed in the annotated element (and in all program elements contained in the annotated element). Note that the set of warnings suppressed in a given element is a superset of the warnings suppressed in all containing elements. For example, if you annotate a class to suppress one warning and annotate a method to suppress another, both warnings will be suppressed in the method. As a matter of style, programmers should always use this annotation on the most deeply nested element where it is effective. If you want to suppress a warning in a particular method, you should annotate that method rather than its class.

**Coding Example of Annotation Implementation in Java**This is time to create and use annotation in real life. Now we will see that how can we create an annotation? How to use annotation? How to use annotation at runtime and make decision on basis of annotation? I hope most of you have used unit test framework JUnit, which use annotations to create test cases and execute them. If you didn’t use it, not a big reason to worry. Let us take an example, suppose we have a class called Maths, which have two methods sum(int a, int b) and divide(int a, int b) which support mathematics add and divide operations. Method divide(int a, int b) throws a custom exception BadParametersException, if any number is divided by zero, it is a invalid operation. In such case if method throws BadParametersException exception, means test case must be pass, because it is a valid case to throw an exception in such scenario. Therefore, I have created a @Test annotation with Expected attribute, which accept any kind to exception class, if method throws same exception, means test case is pass. This is the same way how this scenario is handled in JUnit. Let us try to resolve this case in our way.

**Test.java (Annotation)**

**package** com.hawk.java;

**import** java.lang.annotation.Documented;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Inherited;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*

\* @author SoManyWord.com

\*/

@Documented

@Inherited

@Target (value = ElementType.METHOD)

@Retention (RetentionPolicy.RUNTIME)

**public** @**interface** Test {

/\*\*

\* This field except <code>Class</code> type of any custom <code>Exception</code>.

\* This annotation attribute is used whether method throw assigned custom <code>Exception</code>

\* or not. If method throws custom <code>Exception</code>, it means test case is pass.

\* @return

\*/

**public** Class<? **extends** Exception> Expected() **default** java.lang.Exception.**class**;

}

**BadParametersException.java**

**package** com.hawk.java;

/\*\*

\* The class <code>BadParametersException</code> is sub-class of <code>Exception</code> class.

\* This class is used to throw exception if method arguments are invalid to process.

\* @author SoManyWord.com

\*/

**public** **class** BadParametersException **extends** Exception {

**private** String message = **null**;

**public** BadParametersException() {

**super**();

}

**public** BadParametersException(String message) {

**super**(message);

**this**.message = message;

}

**public** BadParametersException(Throwable throwable) {

**super**(throwable);

}

@Override

**public** String toString() {

**return** **this**.message;

}

@Override

**public** String getMessage() {

**return** **this**.message;

}

}

**Maths.java**

**package** com.hawk.java;

/\*\*

\* General math class, which support few mathematical operations.

\* @author SoManyWord.com

\*/

**public** **class** Maths {

**public** **int** sum(**int** a, **int** b) {

**return** a + b;

}

**public** **float** divide(**int** a, **int** b) **throws** BadParametersException{

**if** (b == 0) {

**throw** **new** BadParametersException("Invalid patameter exception: No number can be divide by 0.");

}

**float** c = a/b;

**return** c;

}

}

**UnitTests.java**

**package** com.hawk.java;

/\*\*

\* All test cases are written here with @Test annotation.

\* @author SoManyWord.com

\*/

**public** **class** UnitTests {

@Test(Expected = BadParametersException.**class**)

**public** **void** testDivide() **throws** BadParametersException {

Maths math = **new** Maths();

math.divide(5, 0);

}

@Test

**public** **void** testSum() {

Maths math = **new** Maths();

**int** sum = math.sum(10, 5);

**if** (sum == 15)

System.out.println("Sum Tast Case Pass, Output Value Is: " + sum);

}

}

**TestRunner.java**

**package** com.hawk.java;

**import** java.lang.reflect.InvocationTargetException;

**import** java.lang.reflect.Method;

/\*\*

\* This class is test suite runner. This class runs all test cases

\* provided in given test class. In main method test class is mentioned.

\* This class particularly check custom exception thrown by method. If thrown exception

\* match with expected exception mentioned on attributes of @Test annotation, that means

\* test case is pass. This same scenario is used in JUnit as well.

\* @author SoManyWord.com

\*/

**public** **class** TestRunner {

**public** **void** runUnitTests(String className) {

**try** {

Class<?> testClass = Class.forName(className);

Object object = testClass.newInstance();

Method[] methods = testClass.getMethods();

**for** (Method method : methods) {

**if** (method.isAnnotationPresent(com.hawk.java.Test.**class**)) {

Test annotation = method.getAnnotation(com.hawk.java.Test.**class**);

Class<? **extends** Exception> expectedClass = annotation.Expected();

**if** (expectedClass != **null**) {

**try** {

method.invoke(object);

}**catch** ( InvocationTargetException ex ) {

**if** ( ex.getTargetException().getClass() == expectedClass ) {

System.out.println("Test Case Pass With Custom Exception (BadParametersException): " + ex.getTargetException().getLocalizedMessage());

} **else** {

System.out.println("Test Case Fail With An Exception: " + ex.getMessage());

}

} **catch** ( Exception ex ) {

System.out.println("Test Case Fail With An Exception: " + ex.getMessage());

}

}

}

}

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (InstantiationException e) {

e.printStackTrace();

} **catch** (IllegalAccessException e) {

e.printStackTrace();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

}

}

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

TestRunner testRunner = **new** TestRunner();

testRunner.runUnitTests("com.hawk.java.UnitTests");

}

}

**output**

Test Case Pass With Custom Exception (BadParametersException): Invalid patameter exception: No number can be divide by 0.

Sum Tast Case Pass, Output Value Is: 15

# How to handle exceptions in thread environment in Java?

Exceptions handling of single-threaded and multi-threaded programs are different from each other. When single-threaded program terminates due to uncaught exception, it stop running and print exception stack trace on console. In other hand, when multi-threaded program encounter with uncaught exception, the stack trace may be printed, but no one may be watch that console. Because when that thread fail, application may appear continuously working. Therefore, there might be change that its failure may go unnoticed. Exception is an unavoidable conditions, which can be occur in program due to lots of reasons. In threading application, we write our task in run() method of Thread class or Runnable interface. As you know run() method is an overridden method, that is why we can not throw any exception from run() method. Therefore all exceptions must be handle in side that run() method. Most of time we used methods of other classes as task in our run() method. In that case we would only aware about checked exceptions which are declared with signature of any methods. But there might be chance that any method can throw RuntimeException. Any poor written code can throw NullPointerException from inside the method. These type of tasks should be call with in try and catch block that catch unchecked exceptions or with in try and finally block to ensure that if the thread exits abnormally the program is informed of this and can take corrective action. There are two way to handle exceptions in Java threads. One is already mentioned above, with try, catch and finally block. Here is an example of same case. In this coding example you can see how to handle exception in thread and how to write recovery procedure for that.

**public** **class** ThreadManager **extends** Thread{

@Override

**public** **void** run() {

Throwable thrown = **null**;

**try** {

**while** (!isInterrupted())

// Run tasks until there is no task in Work Queue.

runTask(getTaskFromWorkQueue());

} **catch** (Throwable ex) {

thrown = ex;

} **finally** {

// threadExited() method is recovery procedure

//if any exception occurs.

threadExited(**this**, thrown);

}

}

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

ThreadManager threadManager = **new** ThreadManager();

threadManager.start();

}

}

Above mentioned approach is proactive approach to the problem of unchecked exceptions. Where as Thread class it self provide UncaughtExceptionHandler facility. This handler let us detect when a thread dies due to an uncaught exception. Thread class has exposed and interface Thread.UncaughtExceptionHandler. We can implement this interface and set this implementation to any thread  Thread.setUncaughtExceptionHandler( UncaughtExceptionHandler eh ), so that thread can call that implementation, if any uncaught exception raise. If we are not able to set exception handler to a particular  thread (For example, we are not able to create thread in thread pools), then you can set a default handler to thread with static method of thread Thread.setDefaultUncaughtExceptionHandler( UncaughtExceptionHandler eh ) . Let me show you an coding example for both, to set an exception handler on particular thread and default exception handler using thread pool.**ExceptionThrowerRunnable.java**

**package** com.hawk.java.thread;

/\*\*

\* Demonstrates exceptions occurring inside a thread.

\*

\* @author SoManyWord.com

\*/

**public** **class** ExceptionThrowerRunnable **implements** Runnable {

**public** **void** run() {

// This task will always throw an IllegalStateException

**throw** **new** IllegalStateException("Running this is never legal.");

}

}

**LoggingThreadExceptionHandler.java**

**package** com.hawk.java.thread;

**import** java.util.logging.Level;

**import** java.util.logging.Logger;

/\*\*

\* Handle exceptions uncaught in threads by logging them.

\* @author SoManyWord.com

\*/

**public** **class** LoggingThreadExceptionHandler **implements** Thread.UncaughtExceptionHandler {

**private** Logger logger = Logger.getLogger("MYLOGGER");

**public** **void** uncaughtException(Thread t, Throwable e) {

logger.log(Level.ALL, "Error { "+ e +" } in Thread { "+ t +" } was uncaught.");

e.printStackTrace();

}

}

**ThreadRunner.java** This class has set uncaught exception handler on particular thread.

**package** com.hawk.java.thread;

/\*\*

\* Execute the threading application to demonstrate exception handling.

\* @author SoManyWord.com

\*/

**public** **class** ThreadRunner {

/\*\*

\* Start the application.

\*

\*/

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

Thread thread = **new** Thread(**new** ExceptionThrowerRunnable());

thread.setUncaughtExceptionHandler(**new** LoggingThreadExceptionHandler());

thread.start();

}

}

**PoolRunner.java** If threads are running inside thread-pools then it is quite difficult to handle exception of each thread running inside thread-pool. Therefore, java.lang.Thread has provided a static method setDefaultUncaughtExceptionHandler() which takes java.lang.Thread.UncaughtExceptionHandler as an argument. By setting a default handler with this static method, you can set a default handler for all threads. This class sets a default uncaught exception handler on threads, which can catch any exception thrown inside thread pool.

**package** com.hawk.java.thread;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

/\*\*

\* Execute the tasks using thread pools.

\* @author SoManyWord.com

\*

\*/

**public** **class** PoolRunner {

/\*\*

\* Execute the {@link ExceptionThrowerRunnable} a couple of times in a pool.

\*/

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

// Set the default uncaught exception handler,

// If you are not able to set handler to a particular thread.

Thread.setDefaultUncaughtExceptionHandler(**new** LoggingThreadExceptionHandler());

ExecutorService pool = Executors.newFixedThreadPool(2);

Runnable r = **new** ExceptionThrowerRunnable();

**for**(**int** i = 0; i < 10; i++) {

pool.execute(r);

}

}

}

Now you know how to handle uncaught exceptions in thread environment. Either you could set handler on particular thread or you could set default handler as well. This default handler will be call if no other handler is set on thread. Hope this blog will help you to write robust code in multi-threading environment.

# Common problems of concurrency (Multi-Threading) in Java

Writing correct program is hard and writing correct concurrent program is harder. There are lots of things that can go wrong in concurrent program than a sequential program. Concurrency (Threading) is very important feature of Java. If you are interested to write concurrent program in Java, then there are lots of things what you have to take care about. Therefore, I would like to cover some common problems, which you could might be encounter with in concurrency. These are some points which are important to understand about concurrent programming. It is necessary to know about reasons and solutions for these problems. I have taken help from really a good book (Java Concurrency In Practice) for concurrent programming in Java to write this blog. Let us discuss these problems one by one.

**1. Atomicity:**In concurrent environment two or many threads can run concurrently, if all threads are trying to update some shared resource concurrently, there might be chance that threads trying to read shared resource get stale data and produce wrong outputs. To resolve this issue, it is necessary to make such critical code atomic. Atomicity is to create any operation atomic, in such way that operation can be performed in a single attempt. If a critical section of code is getting execute by only one thread at a time and no other thread can enter in this section until that thread complete its execution. That means this critical section is atomic. You can make any critical section of code atomic by synchronization.

**2. Visibility:** Visibility of shared object could be different for different threads. This is one of the problems of multi-threading environment. Suppose there are two threads, one is writing any shared variable and other is reading from shared variable. When reading and writing will be occur in different threads, there is no guarantee that reader thread will see the value written by writer thread. This problem of multi-threading is known as visibility issue of shared resource. This problem can be resolved with proper synchronization of shared resource.

**3. Reordering:** Order of execution of code can be different as it is written in class. Because JVM rearrange this code according to its convenience, that is known as reordering. Today’s machines do run on multiple processors. To utilize these processors JVM split code to run with different processors, therefore order of code could be change during execution. There might be change that any critical section of code can generate wrong result after reordering of code. If you do not want to change the order of execution of code written then you will have to synchronize your critical code.

**4. Race Conditions:** A race condition occurs when the correctness of computation depends on the relative time or interleaving of multiple threads by the runtime. In other words, when getting the right answers relies on lucky timing. The most common type of race condition is check-then-act, where a potentially stale observation is used to make a decision on what to do next. You can resolve this problem by make your critical section atomic using proper synchronization.

**5. Livelock:** Livelock is a form of liveness failure in which a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail. Livelock often occurs in transaction a messaging applications, where the messaging infrastructure rolls back a transaction if a message cannot be processed successfully, and puts it back at the head of the queue.

**6. Starvation:** Starvation occur when a thread is denied and not getting access to resource to progress its task. The most common starved resource is CPU cycles. Starvation in Java application can be caused by inappropriate use of thread priorities. For example two thread of same priority is trying to access a common resource and each thread allowing other thread to have its access first. In such situation no thread could get access of resource. That means both threads are suffering from starvation.

**7. Deadlock:** Deadlock occur when two of many thread are waiting for two or more resources, where each thread need to get access on all resources to progress and those resources are acquired by different threads and waiting for other resources to be release, which will not be possible ever. For example, two threads A and B need both resources X and Y to perform their tasks. If thread A acquires resource X and thread B acquires Y and now both threads are waiting for resources to be release by other thread. Which is not possible and this is called deadlock. You need to take care of the way of your synchronization if this is going to be a cause of deadlock.

**Conclusion:** These are few common problems which are most frequently occurs in concurrency in Java. You need to take care about proper synchronization of your code in multi-threaded environment otherwise it could be a cause of big issues. If you want to know more about these issues deeply, you could read my upcoming blogs on same problems with deep detail on issues and their appropriate solutions to make threading robust in Java.

# Why 64 Bit operations are not thread safe in Java?

When you do work in multi-threaded environment in Java. There are lots of things you need to keep in mind to write your code correct and safe. You expect that your code will give you correct output in concurrent execution environment. But there are lots of things which could might be goes wrong in multi-threaded applications. If you want to know more about these things you can read my blog “[Common problems of concurrency (Multi-Threading) in Java](http://www.somanyword.com/2014/03/common-problems-of-concurrency-multi-threading-in-java/)”. But here we are specifically talking about 64 Bit operations in multi-threaded environment. It is important to know about 64 Bit data types how they work internally, which will help us to write correct and robust code. Whenever a thread read any variable without synchronization, it may read some stale value from variable, but at least it will read some value what any other thread write on variable, whether it is fresh or stale. This safety guarantee is known as out-of-thin-air safety. Out-of-thin-air safety applies to all variables except from 64 Bit variables ( double, long ). All 64 Bit variables which are not declared volatile are unsafe to use in multiple threads. Any operation on non-atomic and non-volatile 64 Bit variable is break down in two separate 32 Bit operations by JVM. Therefore if read and write occurs in two different threads, it is possible that read of non-volatile long or double return the higher 32 bits of one value and the lower 32 bits of another. Thus, even if you don't care about stale values, it is still not safe to use shared mutable long and double variables in multi-threaded programs unless they are declared volatile or guarded by a lock, because in this case it may give you totally wrong value which does not belong to program. If you are using 64 Bit variables in your multi-threaded environment please make sure your variable must be volatile or synchronized. In short, it is necessary to make 64 Bit variable atomic, so no other thread can interfere, if any thread is writing on 64 Bit variables. Hope this will help you to write robust code for concurrent environment in Java.

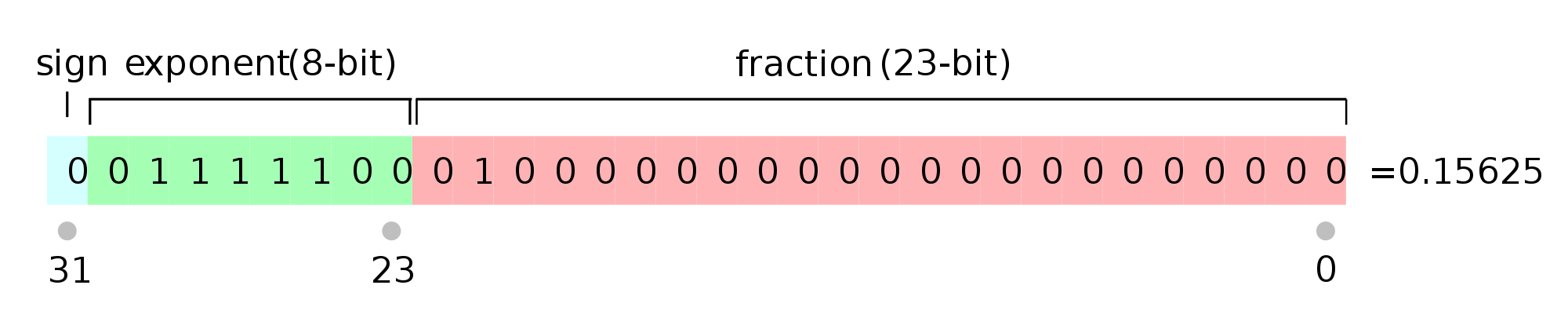
# What is difference between sleep and wait methods in Java?

This is most frequently asked questions in Java telephonic interviews. Because this is very basis question about java programming. Both methods sleep(long millis) and wait() looks alike, means both method halt program execution. But there are huge differences in both methods. Let us discuss them.

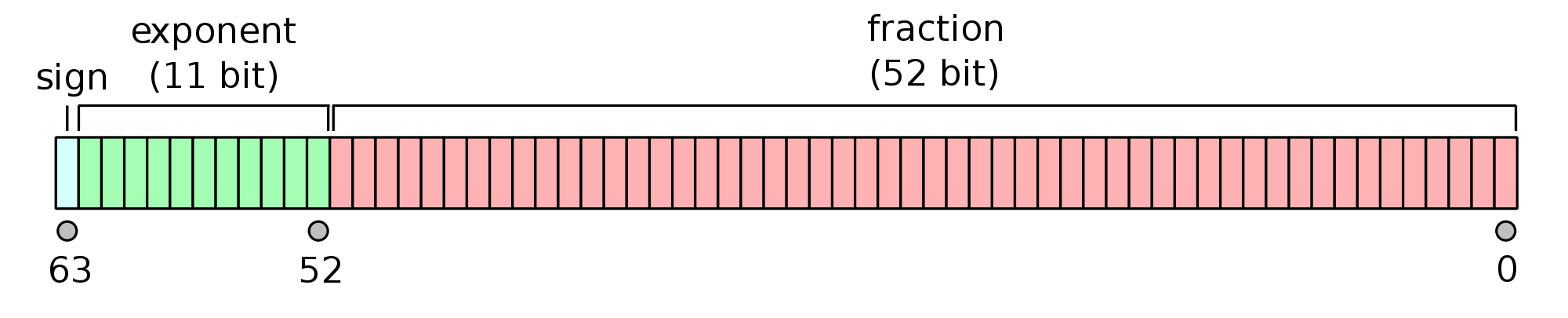
1. sleep(long millis) is method of java.lang.Thread class, while wait() is method of java.lang.Object class.
2. sleep(long millis) is static method, while wait() is non-static method.
3. Most significant difference is sleep(long millis) method halt the execution of current thread till given milliseconds. but current thread does not leave monitor on synchronized resource and does not allow other threads to execute on synchronized resource. While wait() method halt the execution of current thread till another thread notify or wake up this thread with notify() and notifyAll() methods of java.lang.Object class. wait() method leave the monitor on synchronized resource and allow other threads to take lock on synchronized resource.
4. sleep(long millis) method basically used to halt execution for specific time while wait() and nofity() methods are used for inter threads communications.
5. sleep method has two overloaded variants sleep(long millis) and sleep(long millis, int nanos), while wait method has three overloaded variants wait(), wait(long timeout) and wait(long timeout ,int nanos).

# Why float and double are not correct data types for monetary and financial calculation?

When we do write code to hold decimal point value in any variable, first data types come in our mind are float and double. Which is obvious because float and double data types are designed to hold floating points (Fractional Numbers). But only most experienced developers know that float and double are not correct data types to hold any monetary (currency) or financial value. This issue is not only exists in Java programming language. But issue is present in every language which use native floating point to hold decimal values. Therefore it is important to choose appropriate data types to hold such values and perform operations on them. Otherwise your minor mistake can lead a erroneous result. Now question arise, why float and double are not correct data types to hold currency and financial values? Answer is float and double data types are designed to hold binary floating values for scientific calculations. If we elaborate it more, that means float and double are represent binary values with base 2 multiplier. While monetary (currency) or financial values are decimal values with base 10 multiplier. Which is major difference with both types of decimal values. Let us understand little more about float data type and how this data type works and hold binary values internally. As you know, **IEEE 754** representation of float is 32 Bit data type. It dedicate 1 Bit to store ***sign***of value ( -ve or +ve ), next 8 Bits to store ***exponent***for the base of 2 and rest all 23 Bit for ( ***Mantissa***) multiplier of evaluated base.



Similarly, double is 64 Bit data types. It dedicate 1 Bit to store ***sign*** of value (-ve or +ve), next 11 Bits to store ***exponent*** for the base of 2 and rest all 52 Bits for (***Mantissa***) multiplier of evaluated base.



Formula representation for binary floating point number is:

**Floating Point Number = Sign \* 2Exponent \* Mantissa**

Therefore if we have to represent a number 10.25 in floating point it will be represent this number internally as

**10.25 = + \* 2-4 \* 164**

While, monetary (currency) and financial decimal values are evaluated on base 10. Therefore, if we have to store decimal financial values, we need represent like this:

**Floating Point Number = Sign \* 10Exponent \* Mantissa**

To represent number 10.25 in decimal floating point, its representation will be like this.

**10.25 = + \* 10-2 \* 1025**

This is the main issue, that is why, we cannot correctly represent monetary decimal values in float and double data types. But there is an alternative way through which you can achieve this in Java with java.math.BigDecimal class. Similarly, in C# there is decimal data type. Lets see one example which could explain this problem better:

**package** com.hawk.java;

**import** java.math.BigDecimal;

/\*\*

\* This class shows, how decimal values can behave weird

\* with different types of data types and how to resolve

\* this issue in Java.

\*

\* @author SoManyWord.com

\*/

**public** **class** FloatingPoint {

**private** **double** number1 = 10.45;

**private** **double** number2 = 9.30;

**private** BigDecimal decNum1 = **new** BigDecimal(10.45);

**private** BigDecimal decNum2 = **new** BigDecimal(9.30);

**private** BigDecimal decimalNum1 = **new** BigDecimal("10.45");

**private** BigDecimal decimalNum2 = **new** BigDecimal("9.30");

**public** **void** subtractDoubleValues() {

System.out.println("Subtraction With Double Data Types : " + (number1 - number2));

}

**public** **void** subtractBigDecimalValuesWithDoubleConstructor() {

System.out.println("Subtraction With Double Constructor of Big Decimal Class : " + (decNum1.subtract(decNum2)));

}

**public** **void** subtractBigDecimalValuesWithStringConstructor() {

System.out.println("Subtraction With String Constructor of Big Decimal Class : " + (decimalNum1.subtract(decimalNum2)));

}

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

FloatingPoint floatingPoint = **new** FloatingPoint();

floatingPoint.subtractDoubleValues();

floatingPoint.subtractBigDecimalValuesWithDoubleConstructor();

floatingPoint.subtractBigDecimalValuesWithStringConstructor();

}

}

**Outputs**

Subtraction With Double Data Types : 1.1499999999999986

Subtraction With Double Constructor of Big Decimal Class : 1.149999999999998578914528479799628257751464843750

Subtraction With String Constructor of Big Decimal Class : 1.15

This example shows you how double data type and BigDecimal class behave differently. You assume that subtraction will be (**10.45 – 9.30 = 1.15**). But in case of double is output is **1.1499999999999986**.  Initially this value difference is looking minor error, but later on when you will do further operations on this value this minor error will become big and could spoil your all calculation. Therefore, if you want to avoid this situations, there are some points which can help you to avoid this issue.

1. If possible use long or int data types instead of double and float. For example, $10.55 can be convert in to 1055 cents. Which give you more precise calculation.
2. If you are using Java as programming language to perform monetary calculations, use BigDecimal class instead of float and double. In case of C# language you can use decimal data type.
3. Use String constructor of BigDecimal class instead of double constructor of BigDecimal class.

These are few points which is necessary to know to write correct and robust code.

# What is UnSupportedClassVersionError in Java?

Few day back one of my friend call me and asked about one error in his Java program. He told me, his program is throwing UnSupportedClassVersionError. At that time he was not aware about this error, because this error does not occur much frequently as other exception and error like ClassNotFoundException and NoClassDefFoundError. But this is one of important error, you must know about. Therefore, let us discuss UnSupportedClassVersionError more deeply. If you want to know[what is difference between Error and Exception](http://www.somanyword.com/2014/01/what-is-the-difference-between-error-and-exception/), please read my blog [here](http://www.somanyword.com/2014/01/what-is-the-difference-between-error-and-exception/). java.lang.UnSupportedClassVersionError is subclass of java.lang.ClassFormatError and this is the subclass of java.lang.LinkageError and which is subclass of java.lang.Error class. Class hierarchy of UnSupportedClassVersionError is illustrated below:

**JavaDocs** According to javadoc of java.lang.LinkageError class *“Subclasses of LinkageError indicate that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.“* According to javadoc of java.lang.ClassFormatError class *“ ClassFormatError thrown when the Java Virtual Machine attempts to read a class file and determines that the file is malformed or otherwise cannot be interpreted as a class file.”* According to javadoc of java.lang.UnSupportedClassVersionError class *“ UnsupportedClassVersionError thrown when an attempt is made to load a class with a format version that is not supported by the JVM.”* **Example**

Exception in thread "main" java.lang.UnsupportedClassVersionError:

GenerateInvoice (Unsupported major.minor version 49.0)

at java.lang.ClassLoader.defineClass0(Native Method)

at java.lang.ClassLoader.defineClass(ClassLoader.java:539)

at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:123)

at java.net.URLClassLoader.defineClass(URLClassLoader.java:251)

at java.net.URLClassLoader.access$100(URLClassLoader.java:55)

at java.net.URLClassLoader$1.run(URLClassLoader.java:194)

at java.security.AccessController.doPrivileged(Native Method)

at java.net.URLClassLoader.findClass(URLClassLoader.java:187)

at java.lang.ClassLoader.loadClass(ClassLoader.java:289)

at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:274)

at java.lang.ClassLoader.loadClass(ClassLoader.java:235)

at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:302)

**Reason** What is the reason of UnSupportedClassVersionError? As you can see this error is somewhere related to class version as it's name suggesting us. This error occurs, when JVM is not capable to understand the format of class file and ClassLoader reject class to load in JVM. Now question arise why ClassLoader reject any Java class to load in JVM? This is because, if class compilation version of JDK and version of JVM where you want to load is different then is error occurs. Means, if any class is compiled with any upper version of JDK, i.e JDK1.6 and you are trying to load that particular class in lower version JVM i.e JDK1.5, then ClassLoader will throw UnSupportedClassVersionError because JVM is not able to understand upper version format. But vice-versa is not true here, because class compiled with lower version of JDK can be loaded in upper versions of JVM, because upper version JVM gives lower version support. **Resolution**There are few resolution for UnSupportedClassVersionError, you have to select , what suits you in your scenario. 1. If you have source code you class, then you can compile that class in your JDK version accordingly. 2. One easiest way to install same version of JDK, which was used to compile particular JAR or Class or you can install latest version of JDK. 3. If you think JDK version is correct then check your JAVA\_HOME and PATH environment variables. **How to find the java compiler target version from a java class file?** Every Java class has to version associated with it one in *major version* and other is *minor version*. For example 45.3, where 45 is *major version* and 3 is *minor version*. Java major and minor version list is given below according to associated JDK versions.

1. JDK 1.0 → major version 45 and minor version 3
2. JDK 1.1 → major version 45 and minor version 3
3. JDK 1.2 → major version 46 and minor version 0
4. JDK 1.3 → major version 47 and minor version 0
5. JDK 1.4 → major version 48 and minor version 0
6. J2SE 5.0 → major version 49 and minor version 0
7. J2SE 6.0 → major version 50 and minor version 0
8. J2SE 7.0 → major version 51 and minor version 0
9. J2SE 8.0 → major version 52 and minor version 0

If you want to know compiler target version of Java class. There is one command according to Operating System (UNIX/ Linux/ Windows). On UNIX/ Linux: ***javap -verbose MyClass*** or ***javap -verbose MyClass | grep "major"*** On Windows: ***javap -verbose MyClass*** or ***javap -verbose MyClass | findstr "major"***This is the command, which can help you to find compiler target version of JDK. So these are only few things what you have to remember to tackle this error in future. I hope you would like this article and it will help you somewhere in future.

# What is difference between ClassNotFoundException and NoClassDefFountError in Java?

This is one of the most popular interview questions of Java. If you are a Java professional, everybody will expect from you that you would be aware about ClassNotFoundException and NoClassDefFoundError. Because, both are very important in Java application development and have significant differences. Lets discuss differences in both Exception and Error.

**1.** As you can easily guess that **ClassNotFoundException** is 'Excetion', while **NoClassDefFoundError** is 'Error'. If you want to know [difference between Error and Exception](http://www.somanyword.com/2014/01/what-is-the-difference-between-error-and-exception/), you can read my blog [here](http://www.somanyword.com/2014/01/what-is-the-difference-between-error-and-exception/). **2. ClassNotFoundException** is thrown, when application tries to load any class by its String name and that class does not found. This exception thrown when below mentioned methods of given classes get called by String name of a class:

* The forName(String className) method of class Class.
* The forName(String name, boolean initialize, ClassLoader loader) method of class Class.
* The loadClass(String className) method of class ClassLoader.
* The loadClass(String className, boolean resolve) method of class ClassLoader.
* The findClass(String className) method of class ClassLoader.
* The findSystemClass(String className) method of class ClassLoader.

**NoClassDefFoundError** is thrown, when ClassLoader instance tries to load definition of class and no definition of class found. This error happens if class definition existed when the currently executing class was compiled with that definition, but the definition is no longer available. This error thrown when below mentioned methods of ClassLoader class get called:

* The defineClass(String name, byte[] b, int off, int len, ProtectionDomain protectionDomain) method of class ClassLoader.
* The defineClass(String name, ByteBuffer b, ProtectionDomain protectionDomain) method of class ClassLoader.

1. **ClassNotFoundException** is an Exception in Java, which could be happened due to wrong input at runtime. For example at runtime when user trying to load any class by reflective call and that class name does not exist. That means, class name is invalid (wrong input), but this case is recoverable, you can write and alternative mechanism for this. **NoClassDefFoundError** is an Error in Java, which could happened, when any source is compiled successfully but at runtime any required class file would not found. This may be happen sometime during distribution of JARs. For example, at the time of compilation, all classes are exists and source get compiled successfully, but at time of distribution any class file got missed for JAR. In this case, when JVM try to load and create the definition of missing class, NoClassDefFoundError will get raise. This is really a serious problem, because there is no recoverable procedures for errors. JVM will not run application properly if distributed classes are not available. Therefore it is necessary to repackage that distributed JAR and try again.

**Conclusion** Therefore, ClassNotFoundException and NoClassDefFoundError are very important to know, because at the time of Java application development, you may encounter with these problems, which are very common problems. Knowing all the facts about them will reduce your time to resolve these types of issues and you can write robust code by tackling these issue at very first level of development.

# Classloading Mechanism in Java

Classes loading on Virtual Machine is a hidden process of any programming language. Most of the developers do not bother about it. Because this is the responsibility of frameworks to load their respective classes to run Virtual Machine and application on it. But believe me, this is really very important to know about classloaders and classloading mechanism. If you are writing higher end applications or frameworks, which has wide scope of work to do, there might be chance, you need to write your own custom classloaders for your applications. Lots of Java based frameworks have implemented their custom classloaders, which solve their concerned problems. Here we specifically talking about Java classloaders, Lets we discuss about classloading mechanism in Java Virtual Machine. **Class Loading on JVM**Classloading is a process to load required classes on Virtual Machine. Only after loading required classes your application can run on JVM. There are two types of classloading.

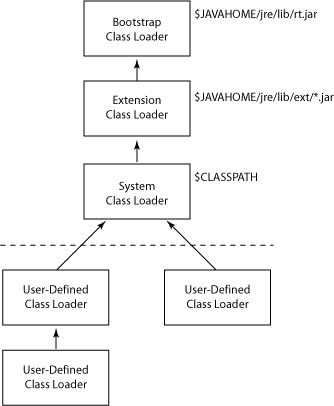
1. **Eager Classloading** is process of loading all classes comprising the source of application before run the application on VM. Application can take some time to run on VM, because classloaders will first load all classes to VM and after that they will run application on VM.
2. **Lazy Classloading** is process of loading classes, at first active use of class rather that loading them on start-up of application on VM.

The first active use of class occurs, when one of the following event occurs:

* An instance of that class is created.
* An instance of one of its subclass is initialized.
* One of its static fields is initialized.

These two are the types of classloading on VM. But its totally depends of JVM vender that what types of classloading they are providing with their JVM. Note that, JVM provided by IBM supports both types of classloading but JVM provided by Oracle (Sun) is only support Lazy classloading. **java.lang.ClassLoader class** Before describe more about classloading mechanism in JVM, I would like to tell you somethings about java.lang.ClassLoader class, which will  help you to under classloading mechanism. According to JavaDoc, the class ClassLoader is an abstract class. If you want to create your own custom class loader for your application or framework then you can extend your classloader class from java.lang.ClassLoader class. There are methods available in ClassLoader class to load and define classes from file system. ClassLoader class load any class by its binary name and return java.lang.Class class object. Every java.lang.Class object contain reference of it ClassLoader. java.lang.Class objects for array classes are not created by classloaders, but are created automatically as required by the Java runtime. The ClassLoader for an array class, as returned by Class.getClassLoader() is the same as the classloader for its element type; if the element type is a primitive type, then the array class has no ClassLoader. The ClassLoader class uses a delegation model to search for classes and resources. Each instance of classloader has an associated parent classloader. When requested to find a class or resource, a ClassLoader instance will delegate the search for the class or resource to its parent class loader before attempting to find the class or resource itself. The Virtual Machine's built-in class loader, called the "***bootstrap classloader***", does not itself have a parent but may serve as the parent of a ClassLoader instance. Class loaders that support concurrent loading of classes are known as *parallel capable classloaders* and are required to register themselves at their class initialization time by invoking the ClassLoader.registerAsParallelCapable() method. Note that the ClassLoader class is registered as parallel capable by default. However, its subclasses still need to register themselves if they are parallel capable. In environments in which the delegation model is not strictly hierarchical, class loaders need to be parallel capable, otherwise class loading can lead to deadlocks because the loader lock is held for the duration of the class loading process (see loadClass methods). Normally, the Java Virtual Machine loads classes from the local file system in a platform-dependent manner. For example, on UNIX systems, the virtual machine loads classes from the directory defined by the CLASSPATH environment variable. However, some classes may not originate from a file; they may originate from other sources, such as the network, or they could be constructed by an application. The method defineClass() converts an array of bytes into an instance of class java.lang.Class. Instances of this newly defined class can be created using Class.newInstance().

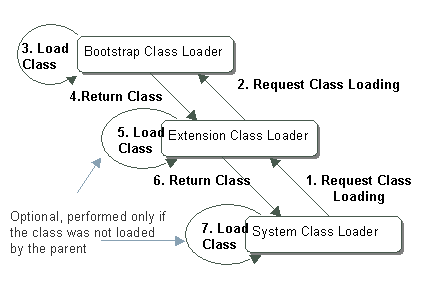
**ClassLoaders Hierarchy:** Java has it own classloaders hierarchy to load classes on JVM. Java has three in-built classloaders, which are shown bellow in classloaders hierarchy diagram.



As you can see in above diagram, these three core classloaders communicate with each other through there hierarchy protocols, which I would mention later-on in this article. Now lets understand about particular classloader individually, their hierarchy and responsibilities:

* **Bootstrap classloader :** Bootstrap classloader is core native classloader of Java. This is the topmost  classloader of classloaders hierarchy, it wouldn't be wrong, if would say, this is the parent of every classloader in Java, like java.lang.Object class is parent class of every class in Java. Because this is native classloader, therefore it's implementation could be vary with different JVMs. Native classloaders are mostly implemented in C language. Bootstrap classloader is responsible to load all available classes in **rt.jar** file, which is located at ***$JAVAHOME/jre/lib/rt.jar***
* **Extension classloader:**Extension classloader come done next to Bootstrap classloader, in other word, this is child classloader of Bootstrap classloader. It is implemented by sun.misc.Launcher$ExtClassLoader class. This classloader is responsible to load all classes available in **ext** folder, which is located at  ***$JAVAHOME/jre/lib/ext/\*.jar***or any other directory specified by ***java.ext.dirs*** system property.
* **System classloader:**System classloader is also known as *Application classloader.*It is implemented by sun.misc.Launcher$AppClassLoader class, which come done next to Extension classloader. This is child classloader of Extension classloader. The main responsibility of this classloader is to load all available classes in ***$CLASSPATH*** or ***java.class.path*** location.
* **User Defined ClassLoaders:**User defined or Custom classloader are those classloaders, which are defined by developers itself for their applications or frameworks. If you want to create your own classloader then you have to extend your classloader class by java.lang.ClassLoader class. These custom classloaders will come done under System classloader in hierarchy. Their class loading responsibility would be defined by the developers.

**Classloading Procedures:** This is very important and interesting to understand, how does JVM load any class internally. Till now you almost clear about ClassLoaders, their hierarchy and their individual responsibilities. Now its time to understand communication between classloaders, their overall class loading mechanism and workflow. Now have smile, lets go little deep and understand it better. Whenever JVM instantiate any class, then JVM will load this class, this is known as *Lazy classloading*. JVM uses a delegation model to load classes.  Basic idea of delegation model is that classloader has a parent classloader, when classloader load a class, it first delegate the search for class to its parent classloader, before attempting to find the class itself.

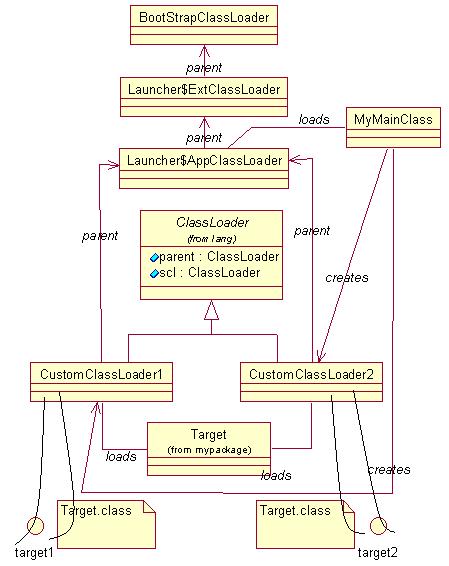


Classloading is a recursive process, When a classloader get request to load a class, it will initiate class loading. First of all it pass the request to it parent classloader, if that parent class loader has any of its parent classloader then it will pass request to its parent classloader and so on. Such way topmost classloader in hierarchy will first search that class in its directory and its *cache* ( Every ClassLoader maintain its own cache ), whether this class is already been loaded by this classloader, if yes then classloader will return class from cache, otherwise classloader will find that class in its directory. If class exist there in directory, it will store that class in its *cache* and return that class to its child classloader and so on. In other hand, if parent classloader did not find that class in its directory and cache, it will throw ClassNotFoundException. Now its child classloader catch this exception and try to find that class in its directory location. The same procedure will go on until that class would not found by any classloader in that hierarchy. If still that class is not available in overall hierarchy of classloader, eventually classloader, who initiate class loading will throw ClassNotFoundException and stop the execution. There are few observations about classloading, which are important in this classloading mechanism:

* Parent classloader has always opportunities to load class first.
* Each classloader maintain their own cache and check any class first in its cache, if it is not available in cache then check it in its directory and try to load it.
* It is ensure that one classlaoder can only see classes loaded by itself or by its parent classloaders, but  not by its children and siblings classloaders.

**ClassLoader Call Flow** In previous section of this article you have learnt about classloading mechanism. Now you are aware about how does classloading take place internally in JVM. In this section, I would like to explain, how custom classloader are implemented by extending java.lang.ClassLoader class and what is the ClassLoader’s methods call flow to load and define a class in JVM. There are three important methods of ClassLoader class, which play significant role to load and define a class in JVM. These methods of ClassLoader class are given bellow:

1. **public** **Class**<?> loadClass(**String** name) **throws** ClassNotFoundException;
2. **protected** **Class**<?> findClass(**String** name) **throws** ClassNotFoundException;
3. **protected** **final** **Class**<?> defineClass(**String** name, **byte**[] b, **int** off, **int** len) **throws** ClassFormatError;



Whenever a ClassLoader is created, it is assigned a parent ClassLoader. As per parent delegation model, when client invoke loadClass(String name) method of ClassLoader class, it will call the loadClass method of parent ClassLoader, where is classloading algorithms is implemented. Before JDK 1.2, loadClass method was abstract, every ClassLoader was required to provide its own classloading algorithm. But later on it is observed that class loading is a critical, complicated and important for JVM, therefore since JDK 1.2 JVM introduced parent delegation model and provide default implementation for loadClass method. After that loadClass method invoke findClass(String name) method of ClassLoader class. Every Custom or User Defined ClassLoader should override findClass(String name) method to specify custom location, where classes need to be search before loading. The findClass method invoke search for given class in specified location, if class found, it read that class and return *bytecodes* of the class. This *bytecodes* of class is passed as byte array to the defineClass(String name, byte[] b, int off, int len) method, defineClass method returns java.lang.Class class, which define that given class. This java.lang.Class instance can be used to instantiate that class in Java application. This is the way, how a class get loading and defined in JVM.

**Conclusion** This article describes, how important a classloading mechanism is to understand, if you a Java developer. This articles explain entire important aspects regarding classloading and ClassLoaders. After reading this article you can create your own ClassLoader, which suites to your applications. Hope you would like this article. This article encourages you to go more deep in concepts.

# What is assertion or assert in Java?

Assertion or assert is a less known feature of Java. Most of developers don't use assert in their daily routine coding, that is why most of developers don't much bother about it. But frankly speeking, this is a useful feature of Java and every developer need to know about it. Most of you, who have used JUnit framework for unit testing, have used assertion to test any test case. JUnit is the best example of Java assertion. Let’s discuss little more about assertion in Java. Since JDK 1.4, Java has introduced assert keyword to support assertion. Assertion is basically used for validation checks in program, if validation fails during assertion check, it throw AssertionError and program got terminated, otherwise it allow program execution as usual.

**What Is Assertion?** Assertion facility introduced in Java since JDK 1.4. Java introduced assertion by assert keyword. An assertion is a boolean expression that a developer specifically proclaims to be true at runtime. There are two forms of assertion in Java

***assert Expression1;***

*or*

***assert Expression1 : Expression2;***

***Expression1*** must be a boolean expression, which is evaluated at runtime, if it is false, an AssertionError is thrown. ***Expression2***value is typically a String, but could also be anything that expression returns. This expression must not return void. It is basically used to customize message for AssertionError.

Note: AssertionError is an error, which is subclass of java.lang.Error class and not recoverable. It is used for logging purpose only.

**Where To Use Assertion?** An assertion is a statement, which assumed to be true during the execution of program. It returns a boolean result. If result is true, code executes normally. Otherwise it throw AssertionError and terminate program execution. Assertion is basically validate the result. It is good to use assertion with one of the following:

***1. Precondition***is a assert condition which must be true, when method is invoked. If an argument is invalid for any method, its validation before using it in method is known as precondition check.  Precondition assertion check must be use in non-public methods. For public methods you must throw IllegalArgumentExcetption.  Because, public methods can be invoked by any client code, if client is invoking any method with illegal value then client must be informed by exception that passed value as argument is illegal for this method. If we use assertion instead of exception here, client would not be aware about his/ her mistake. Therefore precondition assertion must be use for non-public methods. For example.

**private** **void** myMethod(**int** arg) {

**assert** arg>0 && arg <=1000 : “Bad arg: ”+arg;

…

}

***2. Internal Invariant*** is an assumption of program behavior. If somewhere in program any abnormal condition occurs that you can check with assertion. For example.

**if** (x%3 == 0) {

…

} **else** **if** (x%3 == 1) {

…

} **else** {

**assert** x%3 == 2 : x;

// x modulo 3 should be only 0, 1 or 2. Everything

// would work fine until you get negative x’s.

// \*A simple assertion help to discover unsupported results.

}

//--------------------------------------

**switch** (key) {

**case** CST\_VALUE\_A :

…

**break**;

**case** CST\_VALUE\_B :

…

**break**;

…

**default**:

**assert** **false** : key;

// \*A simple assertion help to discover unsupported values.

}

***3. Postcondition*** is a assert condition which, must be true, when method is about to finish its work and ready with final resultant. If final result ready to return, its validation before return by method is known as postcondition check.  Postcondition assertion check can be use in public or  non-public methods.

**public** **double** square(**double** val) {

.. // compute square

**assert** result\*result == val : val;

}

***4. Class Invariant***is type of internal invariant, which must be true with every instance of class. For example implementation of heap. Heap must be balanced with every instance is created inside it.

**private** **boolean** balanced {

…

}

**public** **void** addValue(**double** key, Object obj) {

… // implementation all method should check if heap is balanced prior to its return.

**assert** balanced();

}

***5. Control Flow***is a assertion check that would check a location that will never be executed.

**if** (…) {

…

**return**;

}

**assert** **false**; // should never be reached

**Where To Avoid Assertion** There are few situations, where it is recommended to avoid assertion. Following are the situations:

* Argument checking for public method by assertion is not recommended. Reason behind this is same, what is mentioned in *precondition,s*description.
* In your application where you required correct operation, you must avoid assertion there. Expression evaluation during assert code can be harmful.
* Evaluating the expression should not affect any state that is visible after the evaluation is complete. There is one exception in this rule is that assertion can modify the state that is used only form within other assertions.

**Differences Between Exception and Assertion**

|  |  |
| --- | --- |
| **Exception** | **Assertion** |
| Exception tells the programmer that something gone wrong in the program. | Assertion document or log the program, when it fail and tell that program has bug. |
| Exception are created to inform and deal with problems which might occur in program. | Assertion are create to validate accuracy of program. |
| Exception are used to test input or output as well as whether the program is legal or not | Assertion are used to document and find bug in program. Assertion are also used for debugging purpose. |

**Important Points about Assertion**

* Assertion is introduced since JDK 1.4.
* In order to compile class with assertions using javac compiler, you must use the -source 1.4 command-line option as in this example:

javac **-source 1.4** MyClass.java

This flag is necessary so as not to cause source compatibility problems.

* Assertion can be enable and disable at runtime by using switches *-enableassertions* or *-ea* and *-disableassertions* or *-da*.
* Running 1.4 binaries needs a 1.4 runtime engine. Assertion are default disabled. To activate them, use the following syntaxes:

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| java -ea | Enable assertions in all classes excepted system classes |
| java –ea:<className> | Enable assertions for a specific class |
| java –ea:<packageName>… | Enable assertions for a package and any sub-packages |
| java –ea:… | Enable assertions in the unnamed package in the current working directory |
| java -da | Disable assertions (default) |
| java –da:<className> | Disable assertions for a specific class |
| java –da:<packageName>… | Disable assertions for a package and any sub-packages |
| java –da:… | Disable assertions in the unnamed package in the current working directory |
| java -esa | Enable assertions in system classes |
| java -dsa | Disable assertions in system classes |

**Conclusion** That's all about Java assertion. After reading this article, you understand Java assertions, where to use assertions, how to use assertion and where should avoid using assertions. I hope you will use assertions in your programs to make them more robust.

# What is Bridge Method and Type Erasure in Java Generics?

Bridge method and type erasure in Java generics are advance concepts. These concepts are important for experienced Java professionals. These are basically a low level typics, I mean Java Bytecode compilation level topics. Therefore, It is necessary that you must have good understanding of Java generics and it's implementation before going more deep in these concepts. Bridge method and type erasure are really interesting topics in Java. After reading this article, you would be amazed that how generics classes work internally. Apart from that, these topics are also important for all experienced Java professionals, who wants to have good understanding behind Java Generics. Because, most of interviewers would expect from you that you would have understanding of JVM internal execution and Java Bytecode compilation. If you are really interested to know, let us talk about bridge method and type erasure.

But before going on these topics let us understand what is Java generic and how does generic works behind? When we do create generics classes or interfaces, they do not declare any concert types, while they do not aware of, what kind of objects they will contain at real time. But we tell and specify to generic type,  some specific concert objects types at the time of compilation. These specific concert objects types declaration occurs, when we actually extends, implements and use these generics classes or interfaces. But now question arise, if generics classes or interfaces do not know about concert objects on which they have to perform operations, then how do they manages their internal execution without knowing their concert types? Answer of this question is by type erasure. Lets take a generics coding example to illustrate this.

**public** **interface** Foo<T> {

**public** **void** setName(T name);

**public** T getName();

}

**public** **class** Bar **implements** Foo<String> {

**private** String name;

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getName() {

**return** **this**.name;

}

}

In this example you can see that  Foo<T> interface is generic type and its implementation Bar class can use any kind of concert object in its implementation, here we have used String type object for Bar class implementation. Now you can see that generic type setName and getName methods can accept and return String types objects. But how any generic types can accept any specific concert objects. This problem can be resolve by using type erasure. **Type Erasure In Java** Type erasure is a process to map a generic parameter type or method to its unique bytecode representation. The compiler generates single unique bytecode representation for generic type or method by eliding type parameters and arguments. All instances of generics types get mapped with unique bytecode representation. This unique bytecode representation is created by type erasure. To create an unique bytecode representation, compiler elide generic type to java.lang.Object type (List<T>  → List<Object>), which can accept any kind of objects. If generic type is extended by one of specific type then it is elide to that specific type (List<T extends Animal> → List<Animal>). **Bridge Method In Java** Bridge method is a auto-generated method or unique representation generated by compiler after type erasure to any generic type method. These bridge methods are unique bytecode representations which are used to perform generic operations. The compiler insert a bridge method in sub class of generic super class by type erasure. So that sub class can work as expected. Lets see bridge methods example.

**public** **interface** Foo {

// Bride Method Declaration By Type Erasure

**public** **void** setName(Object name);

// Bride Method Declaration By Type Erasure

**public** Object getName();

}

**public** **class** Bar **implements** Foo {

**private** String name;

**public** **void** setName(String name) {

**this**.name = name;

}

// Bride Method Declaration By Type Erasure

**public** **void** setName(Object name) {

**this**.setName((String) name);

}

**public** String getName() {

**return** **this**.name;

}

// Bride Method Declaration By Type Erasure

**public** Object getName() {

**return** **this**.getName();

}

}

After type erasure conversion by compiler setName(String name) is not implementation of Foo interface. Because Foo interface required setName(Object name) implementation, which is introduced by compiler as bridge method. Therefore bridge method provides accurate implementation. But these bridge methods delegate original methods internally in bridge methods. Bridge methods can not access by client directly, because Java do not expose bridge methods for user to use. But still there is a way to break this rule. Because bridge methods can access by Java reflection. Through refection you can even invoke private methods of class, same you can invoke bridge methods of class through reflection as well. This is how generic type classes or interface transform from generic type to any specific type. In this conversion bridge method and type erasure play an important role. Without bridge methods it would not be possible to map concert type objects with generics types. I hope you would like this post and this post will help you to understand mechanism behind generics.

# What is cloning and how many types of cloning exist in Java?

There are lots of ways to create objects in Java. You can create an object by new keyword with constructor, by Java reflection, by Java deserialization to a serialized object and by Java cloning. But here we are specially taking about Java cloning. Java cloning is process to create an exact copy of an object or clone of an object. Java cloning creates copy of an object without invoking any construct. There are two types of cloning in Java *Shallow Copy* and *Deep Copy* cloning. I will discuss these topics later-on in this article. Java cloning is an controversial topic in Java. Lots of people do not think that cloning is a best implementation in Java. But we are not talking about that controversy here, instead of that we are discussing on present cloning feature in Java and its mechanism.

**How To Create A Class Clone-able?** There are few requirements to create a class clone-able. First of all your class must implement java.lang.Cloneable interface, which is a [marker interface](http://www.somanyword.com/2013/11/user-defined-marker-interface-in-java/) in Java. Therefore, java.lang.Cloneable interface does not declare any abstract method, this is an empty interface, which tell the JVM that, this class supports cloning mechanism. If class doesn't implement java.lang.Cloneable interface then it will throw java.lang.CloneNotSupportedException. This is a checked exception in Java, which is mandatory to handle during cloning. Second and important requirement is to override clone() method of java.lang.Object class. This method is declared as protected and native in java.lang.Object class. Therefore clone() method create clone of object by calling super.clone() method of its parent class and eventually it will call the clone() method of java.lang.Object class. As I told you earlier that cloning process does not invoke constructor of class to create copy of object. That is why cloning use native approach to initialize new object and equal object in memory.

**Important Aspects About Java Cloning** As you know Java cloning is a process to create a exact copy of an object. But there are some important aspects and contract between original object and cloned object. Which is really significant to understand before writing cloning mechanism in Java. Lets talk about them.

1. To support cloning in Java, class must implement java.lang.Cloneable interface.
2. After implementing java.lang.Cloneable interface, class must override clone() method of java.lang.Object class.
3. Cloning should not use constructor to create copy of object. You can use native approach to initialize new cloned object.
4. Cloned object must be a new object in memory. clone() method should not return same instance (this) on which clone() method is invoked. Therefore x == x.clone() must be false.
5. x.getClass() == x.clone().getClass() will be true, But its not an absolute requirement.
6. Cloned object must have same internal state as it's original object. Both original and cloned object's  data members must have similar state or value. Therefore,  x.clone().equals(x) must be true.
7. Checked exception java.lang.CloneNotSupportedException must be handled at the time of cloning.

**Java Cloning Implementation Example**

**public** **class** Foo **implements** Cloneable{

**private** **int** number = 0;

**public** **void** setNumber(**int** number) {

**this**.number = number;

}

**public** **int** getNumber() {

**return** number;

}

@Override

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

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

}

}

**Types Of Cloning In Java** Now you know, what is cloning and all significant aspects about cloning in Java. But types of cloning is also very important aspect in Java. It is most important, if you want to implement clone-able class by yourself. There are two types of cloning in Java and both have significant differences.

1. Shallow Copy
2. Deep Copy

**Shallow Copy:** Shallow copy is a new copy (clone) of original object, but all internal variables of new cloned object are same as original object. Java by default create shallow copy during object cloning. As you have seen above  in *Foo* class example, *Foo* class's override clone() method call super.clone() native method of java.lang.Object class. That's mean native clone() method of java.lang.Object class assign similar internal variable reference of original object to cloned object's variable. In that case, we have two different objects but internally they holding same references as their data members. Means variables of original object and cloned object both referring same objects on heap. This technique is correct if your class holding only primitives or [immutable](http://www.somanyword.com/2014/05/how-to-make-any-class-immutable-in-java/) objects, but this technique goes wrong, if your class hold any mutable object. Because any change made on mutable reference of original object will be reflect on  cloned object or vice-versa. Therefore, shallow copy technique somewhere breaking cloning purpose. In such way original object and cloned object are not totally independent, somewhere both are dependent on each other. You can see below diagram, how original and shallow copy objects share internal references.

Let’s see shallow copy example and its disadvantage.

**public** **class** Foo **implements** Cloneable{

// Primitive Reference

**private** **int** number = 0;

// Immutable Reference

**private** String name;

// Mutable Reference

**private** Bar bar;

**public** **void** setNumber(**int** number) {

**this**.number = number;

}

**public** **int** getNumber() {

**return** number;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** **void** setBar(Bar bar) {

**this**.bar = bar;

}

**public** Bar getBar() {

**return** bar;

}

@Override

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

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

}

}

ShallowCopyCloneTest.java class shows that, how should we create a clone object from cloneable class. This class will also put some light on problem exist in shallow copy technique.

**public** **class** ShallowCopyCloneTest {

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

**try** {

Bar bar = **new** Bar();

bar.setValue("Hi");

Foo object = **new** Foo();

object.setNumber(20);

object.setName("Hawk");

object.setBar(bar);

Foo clone = (Foo) object.clone();

System.out.println("Foo Orignial# Number:" + object.getNumber() + " Name:" + object.getName() +" Bar-Value:" + object.getBar().getValue());

System.out.println("Foo Cloned# Number:" + clone.getNumber() + " Name:" + clone.getName() +" Bar-Value:" + clone.getBar().getValue());

object.setNumber(40);

object.setName("Eagle");

object.getBar().setValue("Hello");

object.setBar(bar);

System.out.println("Foo Original Object Changed....");

System.out.println("Foo Orignial After Change# Number:" + object.getNumber() + " Name:" + object.getName() +" Bar-Value:" + object.getBar().getValue());

System.out.println("Foo Cloned Without Change# Number:" + clone.getNumber() + " Name:" + clone.getName() +" Bar-Value:" + clone.getBar().getValue());

} **catch** (CloneNotSupportedException e) {

e.printStackTrace();

}

}

}

Output after running shallowCopyCloneTest.java class

Foo Orignial# Number:20 Name:Hawk Bar-Value:Hi

Foo Cloned# Number:20 Name:Hawk Bar-Value:Hi

Foo Original Object Changed....

Foo Orignial After Change# Number:40 Name:Eagle Bar-Value:Hello

Foo Cloned Without Change# Number:20 Name:Hawk Bar-Value:Hello

If you would see the code and output carefully, you can understand that cloned object looks like exact copy of original object before any modification done on original object. But after making modification on original object 'Number' and 'Name' data member of cloned object doesn't get change, because 'Number' is primitive data type and 'Name' is String type, which is [immutable](http://www.somanyword.com/2014/05/how-to-make-any-class-immutable-in-java/) in Java, that mean they are safe to share between objects. But 'Bar-Value, of cloned object get changed itself with modification on  'Bar' reference of original object, this is because 'Bar' reference is mutable, which is not safe to share between objects. This is how shallow copy break rule of cloning that both original and cloned object should be conceptually equal and independent. Modification on one object should not reflect on other object. **Deep Copy** Deep copy cloning is a solution for references sharing problem of shallow copy cloning. Deep copy cloning is very important and useful cloning technique, which takes guaranty that cloned copy will be an equal object as original object and both objects will be totally independent, that means no internal references will be shared (Modification on original object will not affect cloned object any more or vice-versa). In deep copy cloning all mutable variables further get manual deep cloned before returning cloned object to client. This is the responsibility of class author that if your class is cloneable, then you will use deep copy cloning techniques, so that your class can return deep copy as cloned object. But somewhere we have to compromise with some basic cloning rules, like "We should not use constructor during cloning". But if we have to create deep copy cloning, somewhere we have to use constructor to create deep copy object, that means we have to create new references for cloned object.Here, I would like to use an best example, which is illustrated by **Jashua Bloch** in his book **"Effective Java"** . This is an amazing book for Java concepts. Everyone must read this book once.  In this simple example he has shown that how java.util.HashTable class has implemented deep copy cloning inside this class. In java.util.HashTable class Entry[] array is an mutable reference, which is deep cloned by deepCopy() method. Go through this example, it will give you a good idea about, how should we create deep copy for a cloned object.

**public** **class** HashTable **implements** Cloneable {

**private** Entry[] buckets = ...;

**private** **static** **class** Entry {

**final** Object key;

Object value;

Entry next;

Entry(Object key, Object value, Entry next) {

**this**.key = key;

**this**.value = value;

**this**.next = next;

}

// Recursively copy the linked list headed by this Entry

Entry deepCopy() {

**return** **new** Entry(key, value,

next == **null** ? **null** : next.deepCopy());

}

}

@Override

**public** HashTable clone() {

**try** {

HashTable result = (HashTable) **super**.clone();

result.buckets = **new** Entry[buckets.length];

**for** (**int** i = 0; i < buckets.length; i++)

**if** (buckets[i] != **null**)

result.buckets[i] = buckets[i].deepCopy();

**return** result;

} **catch** (CloneNotSupportedException e) {

**throw** **new** AssertionError();

}

}

}

**Conclusion** That is all about Java cloning. In this article you have understand basic requirement for cloning, all important rules to implement cloning. You have read about differences between shallow copy and deep copy cloning. Now you know what types of problems, you may encounter with in case of shallow copy cloning. But most important thing is that, Now you have a good idea about deep copy cloning and you know what approach you should take to write your own deep copy cloning approach. I hope this article will help you somewhere to write correct and robust code.

# How to make any class immutable in Java?

Immutability is most important topic in respect of coding practice. This is one of the frequently asked topic in Core Java interview. Every interviewer expect that you must have good understanding about immutable classes, advantages of immutable classes and very important that how to create any class immutable in Java. These are few important aspects, which is necessary to know about immutable objects. In this article, I would divide it into three parts, so that you can have better understanding about immutability level by level. Let us talk about immutability concept deeply.

**1. What Is Immutable Class?** As you know general meaning of "immutability" in English is "A quality, not to being changeable". Same a class called immutable, only if it's internal state never change once it is initialized. When a immutable class get initialize, it create its internal state ( initialize it's data members ) at the time of initialization by passing arguments through constructor. Once an immutable object get initialized, nobody can changes its internal data members. In Java there are lots of classes, which are immutable in nature, like java.lang.String, java.lang.Integer, java.lang.Double etc.

**2. What Are The Advantages Of Immutability?** This is really a nice question that why should we create immutable class? or what are the advantages of immutable class? The main reason to make any class immutable is it's internal state safety. Immutable classes are *thread-safe*. Immutable objects can be passed to untrusted code. Biggest worry in multi-threading environment is that any mutable object shared between threads can spoil its state unpredictably. But immutable objects are totally safe to share between threads. Second advantage is that its simple to make any class immutable by following some set of simple rules. There are predefined rules which are necessary to follow to protect object state. We will cover all these rules in next section of this article. Third advantage is that we don't need to implement cloning on immutable class. We can share same copy anywhere. Because, immutable objects don't change it's state anyhow. If you want to learn about cloning, you can read my article, [What is cloning and types of cloning in Java?](http://www.somanyword.com/2014/05/what-is-cloning-and-how-many-types-of-cloning-exist-in-java/)

**3. How To Make A Class Immutable?** As of now, you know what is immutability. But how can we make any class immutable in Java? There are some set of rules, which we need to follow at the time of creating any immutable class. In this section we will go through all important rules, what we need to care about and will understand how these rule can make any class immutable. Lets see how we can make a class immutable. Before that, I would like to show you one simple class example (Immutable Class), which gives you an idea, how an immutable class look like and why we say it immutable.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **final** **class** Immutable {

// int is primitive data type.

**private** **final** **int** value;

// String is immutable class;

**private** **final** String text;

// List is mutable type.

**private** **final** List<String> list;

**public** Immutable(**int** value, String text, List<String> list) {

**this**.value = value;

**this**.text = text;

**this**.list = copyOfList(list);

}

**public** **int** getValue() {

**return** value;

}

**public** String getText() {

**return** text;

}

**public** List<String> getList() {

**return** copyOfList(list);

}

**private** List<String> copyOfList(List<String> list) {

List<String> newList = **new** ArrayList<String>();

**for** ( String element : list )

newList.add(element);

**return** newList;

}

}

Please understand this above example carefully, because in next step, I will use this example to explain all important rules regarding immutability. Lets talk about the immutability rules in detailed manner:

1. All data members inside class must be private.  Because, private data members cannot access from outside of the class. Therefore, nobody can access and change the state of internal data members.
2. All data members inside class should be final. It is a good coding practice, but this is not hard and fast rule. Because immutable class don't allow to access any data member of immutable class outside. Therefore, It doesn't meter that whether data members of class are final or not.
3. All data member must be set through constructor. No setter method will be provided for any data member. Because, if class will provide setter methods then anybody can change immutable objects state.
4. Class must be declared as final. So that no other class can extend this class and spoil its immutability. Now, there could be one question in your mind, how can we break immutability by extending non final class? Lets me show you an example:

**public** **class** Immutable {

**private** **final** Object value;

**public** Immutable(Object value) {

**this**.value = value;

}

**public** Object getValue() {

**return** value;

}

}

**public** **class** Mutable **extends** Immutable {

**private** Object realValue;

**public** Mutable(Object value) {

**super**(value);

realValue = value;

}

**public** Object getValue() {

**return** realValue;

}

**public** **void** setValue(Object newValue) {

realValue = newValue;

}

}

Here you can see how extended class Mutable can pass its own Object in it's superclass Immutable constructor. Now Mutable class can easily break immutability of Immutable class by changing its own Object. That is why it is significant to make class final.

1. All mutable objects coming in through constructor should not use directly, instead of that, create new object copy of that object, which must be equal to passed object. As you can see in my topmost coding example (Immutable Class), inside constructor I have used this.list = copyOfList(list);  instead of using this.list = list; Here I have create an copy of list object rather than passing it directly.
2. All mutable objects going outside through getter methods must return new copy of mutable object instead of returning same object outside. New returned copy object must be equal to the original object. As you can see in my topmost coding example (Immutable Class), I have used return copyOfList(list); inside getList() method instead of using return list; Because I don't want to share my internal reference to outside the world.
3. But primitive and immutable reference can be used directly and can be shared outside the world without any hesitation. Because primitive and immutable reference cannot change anyhow.

**Conclusion** That is all about immutability concept. By reading this article, you know about immutability, advantage of immutability and all important rules to make class immutable. This concept don't belong to any specific language. You can follow these rules to make your class immutable in any language. I hope this article will encourage you to use immutability in your coding practices.

# UML relationships like Association, Aggregation, Composition, Abstraction, Generalization, Specialization, Realization and Dependency

In this article, we are specifically taking about few Object Oriented Programming relationships like association, aggregation, composition, abstraction, generalization, specialization, realization and dependency. These relationships are very common for any object oriented programming. Despite of that, lots of developers feels that these terms are little weird and unknown for them. In this article, I would try to explain each and every term with suitable examples. I hope after reading this article few of you would be amazed that these all relationships are so familiar to you. But you do not know them so closely. But after reading this you can understand these relationships very well. These relationships are very important and useful in UML to demonstrate relationships between classes and objects. So let us explore these topics one by one.

**Association:** In Object Oriented Programming, each object shared some kind of relationships with other objects. Therefore, relationship between objects is known as “Association”. This relationship could be any type like one-to-one, one-to-many or many-to-many. For example, Student and Subject relationship. Relationship between Student and Subject is many-to-many, because one Student can read many Subjects and one subject can be read by many Students. Therefore, Students and Subjects are associated with each other.

**Aggregation:** Aggregation is special type of association, in which one object contain other object, which is directional relationship. This relationship is also known as “has a” relationship. For example, every Employee has a Department. Therefore, there is an aggregation between Employee object and Department object.

**Composition:** Composition is special case of aggregation. In this relationship, one object contains other object. This is also a “has a” relationship. But there is one restriction with this relationship, that is, contained object can’t exist without container object. For example, Person has a Address, but Address does not have any importance and can’t exist without Person. Therefore, a Person composes an Address.

**Abstraction:** Abstraction is one of the important concepts of Object Oriented Programming. Abstraction is to hiding implementation information from user. Instead of exposing implementation, its declare methods or functions and their general detailed information to help implementation. Later on user can provide concreteness by giving correct implementation based on details information provided by abstract layer. Examples of abstraction are Abstract classes and Interfaces.

**Generalization:** Generalization is a relation between Superclass and Subclass. This relation is “is a” relationship. Generalization is process of moving common code from Subclasses to Superclass. This process will make that common code to general code. So that, this general code can be reuse by any of its Subclass. Generalization is type of inheritance.

**Specialization:** Specialization is a process of creating a new subclass of existing superclass. In specialization, subclass extends existing general implementation to specific implementation by overriding existing implemented methods. This is also “is a” relationship. Specialization is type of inheritance.

**Realization:** Realization is a relationship between interface and class. In realization, new class implements all unimplemented abstract methods and provide concrete implementation. This process provides real business logic implementation to any abstract class or interface. This is also “is a” relationship.

**Dependency:** Dependency is relationship between two or more classes, in which, if any modification done on one class can affect other classes. That means one class depends on second clas . In such way one class has dependency on second class.

**Symbols Used By UML To Show Relationships Between Classes Or Objects**

# What is Reentrant Synchronization or Locking in Java?

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

**Intrinsic Or Monitor Lock** Java provides built-in locking mechanism to make any operation atomic by synchronization in concurrent environment. In Java programming language ‘synchronized' keyword is used for taking lock or monitor on object. There are other locking utility also available in Java but we will concentrate on basic locking mechanism (synchronized) only. synchronized keyword can be use with method signature or with particular code block to make that synchronize. These built-in locks known as intrinsic or monitor lock.

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

1. Creating object level lock with method signature.

**public** **synchronized** **void** objectLevelLock() {

// Code to make synchronized or atomic

}

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

**public** **void** objectLevelLock () {

**synchronized**(lockObject) {

// Code to make synchronized or atomic

}

}

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

**public** **static** **synchronized** **void** classLevelLock () {

// Static members and Code to make synchronized or atomic

}

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

**public** **void** classLevelLock () {

**synchronized**(ClassName.**class**) {

// Static members and Code to make synchronized or atomic

}

}

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

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

**public** **class** Reentrancy {

**public** **synchronized** **void** inner() {

// Code to make synchronized or atomic

}

**public** **synchronized** **void** outer() {

// Code to make synchronized or atomic

inner();

}

}

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

**How Reentrant Lock Works In Java?** Reentrancy is implemented in Java by associating a counter with each thread. Initially counter initialized with value zero and considered as unlocked. When thread acquires lock on an object, counter get incremented by one. Again, if thread acquires lock on another synchronized method or block, counter again get incremented by one; counter will become two and so on. Same reverse process is followed, when thread release lock by leaving synchronized method or block. When thread releases lock from synchronized method or block; counter get decremented by one and so on. Once again, when counter reached to zero; object gets unlocked. Now other threads are free to acquire lock on that object. This is the approach by which Java manage reentrancy.

**Conclusion** That’s all how Java intrinsic and reentrant lock works. After reading this article, you are aware about, what are intrinsic and reentrant locks and how these locks works. I hope this article will help you to understand about Java locking mechanism.