**Lazy Initialization**

In [computer programming](http://en.wikipedia.org/wiki/Computer_programming), **lazy initialization** is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

This is typically accomplished by maintaining a flag indicating whether the process has taken place. Each time the desired object is summoned, the flag is tested. If it is ready, it is returned. If not, it is initialized on the spot.

In a [software design pattern](http://en.wikipedia.org/wiki/Software_design_pattern) view, lazy initialization is often used together with a [factory method pattern](http://en.wikipedia.org/wiki/Factory_method_pattern). This combines three ideas:

* using a factory method to get instances of a class ([factory method pattern](http://en.wikipedia.org/wiki/Factory_method_pattern))
* storing the instances in a map, so you get the same instance the next time you ask for an instance with same parameter (compare with a [singleton pattern](http://en.wikipedia.org/wiki/Singleton_pattern))
* using lazy initialization to instantiate the object the first time it is requested (lazy initialization pattern).

Example

import java.util.\*;

public class Fruit

{

private static final Map<String,Fruit> types = new HashMap<String,Fruit>();

private final String type;

// using a private constructor to force use of the factory method.

private Fruit(String type) {

this.type = type;

}

/\*\*

\* Lazy Factory method, gets the Fruit instance associated with a

\* certain type. Instantiates new ones as needed.

\* @param type Any string that describes a fruit type, e.g. "apple"

\* @return The Fruit instance associated with that type.

\*/

public static synchronized Fruit getFruit(String type) {

if(!types.containsKey(type))

types.put(type, new Fruit(type)); // Lazy initialization

return types.get(type);

}

}

|  |
| --- |
| **Cloning** |
|  |
| **Q1) What are different types of cloning in Java?**  Ans) Java supports two type of cloning: - Deep and shallow cloning. By default shallow copy is used in Java. Object class has a method clone() which does shallow cloning. |
| **Q2) What is Shallow copy?**  Ans) In **shallow copy** the object is copied without its contained objects. Shallow clone only copies the top level structure of the object not the lower levels. It is an exact bit copy of all the attributes.  Original Figure 1: Original java object obj  The shallow copy is done for obj and new object obj1 is created but contained objects of obj are not copied.  Shallow Copy Figure 2: Shallow copy object obj1  It can be seen that no new objects are created for obj1 and it is referring to the same old contained objects. If either of the containedObj contain any other object no new reference is created |
| **Q3) What is deep copy and how it can be acheived?**  Ans) In **deep copy** the object is copied along with the objects it refers to. Deep clone copies all the levels of the object from top to the bottom recursively.  Original Figure 3 : Original Object obj  When a deep copy of the object is done new references are created.  Deep Copy Figure 4: obj2 is deep copy of obj1  One solution is to simply implement your own custom method (e.g., deepCopy()) that returns a deep copy of an instance of one of your classes. This may be the best solution if you need a complex mixture of deep and shallow copies for different fields, but has a few significant drawbacks:   * You must be able to modify the class (i.e., have the source code) or implement a subclass. If you have a third-party class for which you do not have the source and which is marked final, you are out of luck. * You must be able to access all of the fields of the classâ€™s superclasses. If significant parts of the objectâ€™s state are contained in private fields of a superclass, you will not be able to access them. * You must have a way to make copies of instances of all of the other kinds of objects that the object references. This is particularly problematic if the exact classes of referenced objects cannot be known until runtime. * Custom deep copy methods are tedious to implement, easy to get wrong, and difficult to maintain. The method must be revisited any time a change is made to the class or to any of its superclasses.   Other common solution to the deep copy problem is to use **Java Object Serialization** (JOS). The idea is simple: Write the object to an array using JOSâ€™s **ObjectOutputStream** and then use **ObjectInputStream** to reconsistute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph.   * It will only work when the object being copied, as well as all of the other objects references directly or indirectly by the object, are serializable. (In other words, they must implement java.io.Serializable.) Fortunately it is often sufficient to simply declare that a given class implements java.io.Serializable and let Javaâ€™s default serialization mechanisms do their thing. Java Object Serialization is slow, and using it to make a deep copy requires both serializing and deserializing.   There are ways to speed it up (e.g., by pre-computing serial version ids and defining custom readObject() and writeObject() methods), but this will usually be the primary bottleneck. The byte array stream implementations included in the java.io package are designed to be general enough to perform reasonable well for data of different sizes and to be safe to use in a multi-threaded environment. These characteristics, however, slow down ByteArrayOutputStream and (to a lesser extent) ByteArrayInputStream . |
| **Q4) What is difference between deep and shallow cloning?**  Ans) The differences are as follows:   * Consider the class:   public class MyData{ String id; Map myData; } The shallow copying of this object will have new id object and values as “” but will point to the myData of the original object. So a change in myData by either original or cloned object will be reflected in other also. But in deep copying there will be new id object and also new myData object and independent of original object but with same values.   * Shallow copying is default cloning in Java which can be achieved using clone() method of Object class. For deep copying some extra logic need to be provided. |
| **Q5) What are the characteristics of a shallow clone?**  Ans) If we do a = clone(b) 1) Then b.equals(a) 2) No method of a can modify the value of b. |
| **Q6) What are the disadvantages of deep cloning?**  Ans) Disadvantages of using Serialization to achieve deep cloning –   * Serialization is more expensive than using object.clone(). * Not all objects are serializable. * Serialization is not simple to implement for deep cloned object.. |

**Use of hashCode() and equals()**

Object class provides two methods hashcode() and equals() to represent the identity of an object. It is a common convention that if one method is overridden then other should also be implemented.

Before explaining why, let see what the contract these two methods hold. As per the Java API documentation:

1. Whenever it is invoked on the same object more than once during an execution of a Java application, the hashcode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
2. If two objects are equal according to the equals(object) method, then calling the hashCode() method on each of the two objects must produce the same integer result.
3. It is NOT required that if two objects are unequal according to the equals(Java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Now, consider an example where the key used to store the in Hashmap is an Integer. Consider that Integer class doesn't implement hashcode() method. The code would look like:

map.put(new Integer(5),"Value1");  
String value = (String) map.get(new Integer(5));  
System.out.println(value);  
//Output : Value is null

Null value will be displayed since the hashcode() method returns a different hash value for the Integer object created at line 2and JVM tries to search for the object at different location.

Now if the integer class has hashcode() method like:

public int hashCode(){  
return value;  
}

Everytime the new Integer object is created with same integer value passed; the Integer object will return the same hash value. Once the same hash value is returned, JVM will go to the same memory address every time and if in case there are more than one objects present for the same hash value it will use equals() method to identify the correct object.

Another step of caution that needs to be taken is that while implementing the hashcode() method the fields that are present in the hashcode() should not be the one which could change the state of object.

Consider the example:

**public** **class** FourWheeler **implements** Vehicle {

**private** String name;

**private** **int** purchaseValue;

**private** **int** noOfTyres;

**public** FourWheeler(){}

**public** FourWheeler(String name, **int** purchaseValue) {

**this**.name = name;

**this**.purchaseValue = purchaseValue;

}

**public** **void** setPurchaseValue(**int** purchaseValue) {

**this**.purchaseValue = purchaseValue;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + ((name == **null**) ? 0 : name.hashCode());

result = prime \* result + purchaseValue;

**return** result;

}

}

FourWheeler fourWObj = new FourWheeler(â€œSantroâ€?,â€?333333);  
map.put(fourWObj,â€?Hyundai);  
fourWObj.setPurchaseValue(â€œ555555)  
System.out.println(map.get(fourWObj));  
//Output: null

We can see that inspite of passing the same object the value returned is null. This is because the hashcode() returned on evaluation will be different since the purchaseValue is set to 555555â€™ from 333333â€™.

Though the above implementation is correct but it fails because for generating hashcode a changeable property (in this case price) is selected. To make above implementation correct we can either exclued the property or include some other property like noOfTyres and keep the logic of implementation same.

Hence we can conclude that the hashcode() should contain fields that doesn't change the state of object.

One compatible, but not all that useful, way to define hashCode() is like this:

public int hashcode(){  
return 0;  
}  
This approach will yield bad performance for the HashMap. The conclusion which can be made is that the hashcode() should(not must) return the same value if the objects are equal. If the objects are not equal then it must return different value.

**Overriding the equals() method**

Consider the example:

**public** **class** StringHelper {

**private** String inputString;

**public** StringHelper(String string) {

inputString=string;

}

@Override

**public** **int** hashCode() {

**return** inputString.length();

}

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

StringHelper helperObj = **new** StringHelper("string");

StringHelper helperObj1 = **new** StringHelper("string");

**if**(helperObj.hashCode() == helperObj1.hashCode()){

System.*out*.println("HashCode are equal");

}

**if**(helperObj.equals(helperObj1)){

System.*out*.println("Objects are equal");

}**else**{

System.*out*.println("Objects are not equal");

}

}

**public** String getInputString() {

**return** inputString;

}

// Output:  
HashCode are equal  
Objects are not equal

We can see that even though the StringHelper object contains the same value the equals method has returned false but the hashcode method has return true value.

To prevent this inconsistency, we should make sure that we override both methods such that the contract between both methods doesn't fail.

**Steps that need to be taken into consideration while implementing equals method.**

1. **Use the == operator to check if the argument is a reference to this object.** If so, return true. This is just a performance optimization, but one that is worth doing if the comparison is potentially expensive.

2. **Use the instanceof operator to check if the argument has the correct type**.

If not, return false. Typically, the correct type is the class in which the method occurs. Occasionally, it is some interface implemented by this class. Use an interface if the class implements an interface that refines the equals contract to permit comparisons across classes that implement the interface. Collection interfaces such as Set, List, Map, and Map.Entry have this property.

3. **Cast the argument to the correct type. Because this cast was preceded by an instanceof test, it is guaranteed to succeed.**

4. **For each significant field in the class, checks if that field of the argument matches the corresponding field of this object**.If all these tests succeed, return true; otherwise, return false

5. **When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?**

The correct implementation if equals method for the StringHelper class could be:

@Override

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

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

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

**return** **false**;

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

**if** (inputString == **null**) {

**if** (other.inputString != **null**)

**return** **false**;

} **else** **if** (!inputString.equals(other.inputString))

**return** **false**;

**return** **true**;

  }

**How many ways to create an object in java??? plz give an example.**

There are four different ways (I really don t know is there a fifth way to do this) to create objects in java:

**1. Using new keyword**  
This is the most common way to create an object in java. I read somewhere that almost 99 of objects are created in this way.

MyObject object new MyObject();

**2. Using Class.forName()**  
If we know the name of the class & if it has a public default constructor we can create an object in this way.

MyObject object (MyObject) Class.forName( subin.rnd.MyObject ).newInstance();

**3. Using clone()**The *clone()* can be used to create a copy of an existing object.

MyObject anotherObject new MyObject();  
MyObject object anotherObject.**clone()**;

**4. Using object deserialization**  
Object deserialization is nothing but creating an object from its serialized form.

ObjectInputStream inStream new ObjectInputStream(anInputStream );  
MyObject object (MyObject) inStream.readObject();

**Example:**

package objectCreation;  
  
import java.io.ByteArrayInputStream;  
import java.io.ByteArrayOutputStream;  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.io.Serializable;  
  
public class TestObjectCreation implements Cloneable, Serializable{  
  
/\*\*  
\* @param args  
\*/  
  
public void method1(){  
System.out.println("inside method1");  
}  
  
  
public static void main(String[] args) throws ClassNotFoundException InstantiationException IllegalAccessException CloneNotSupportedException IOException {  
  
//way no 1  
Class classObj =Class.forName("objectCreation.TestObjectCreation");  
TestObjectCreation obj1 =(TestObjectCreation)classObj.newInstance();  
obj1.method1();  
  
//way no 2  
TestObjectCreation obj2 =new TestObjectCreation();  
obj2.method1();  
  
//using clone  
TestObjectCreation obj3 =(TestObjectCreation)obj1.clone();  
obj3.method1();  
  
//from deserialization  
ByteArrayOutputStream baos= new ByteArrayOutputStream();  
ObjectOutputStream oos =new ObjectOutputStream(baos);  
oos.writeObject(obj1);  
ByteArrayInputStream bais =new ByteArrayInputStream(baos.toByteArray());  
ObjectInputStream ois =new ObjectInputStream(bais);  
TestObjectCreation obj4 =(TestObjectCreation)ois.readObject();  
obj4.method1();  
  
  
}  
  
}  
  
OUTPUT:  
inside method1  
inside method1  
inside method1  
inside method1

**What are the differences between HashMap and Hashtable?**

**Answer**   
Both provide key-value access to data. The Hashtable is one of the original collection classes in Java. HashMap is part of the new Collections Framework, added with Java 2, v1.2.

The key difference between the two is that access to the Hashtable is synchronized on the table while access to the HashMap isn't. You can add it, but it isn't there by default.

Another difference is that iterator in the HashMap is fail-safe while the enumerator for the Hashtable isn't. If you change the map while iterating, you'll know.

And, a third difference is that HashMap permits null values in it, while Hashtable doesn't.

For new code, I would tend to always use HashMap.

1)Synchronized means only one thread can modify a hash table at one point of time.Basically, it means that any thread before performing an update on a hashtable will have to acquire a lock on the object while others will wait for lock to be released.

2)Fail-safe is relevant from the context of iterators.If an iterator has been created on a collection object and some other thread tries to modify the collection object "structurally",a concurrent modification exception will be thrown.It is possible for other threads though to invoke "set" method since it doesnt modify the collection "structurally".However, if prior to calling "set", the collection has been modified structurally, "IllegalArgumentException" will be thrown.

**How to Upload a File and How to Store it?**

A quick and dirty Java Sockets API program:

// imports java.net.Socket, java.io.\*

void uploadFile(String hostAddr, int port, String fileName) throws   
FileNotFoundException, IOException {   
byte[] buf = new byte[1000];   
int len = -1;

// connect to host at hostAddr, port   
Socket sock = new Socket(hostAddr, port);

FileInputStream in = new FileInputStream(fileName);   
OutputStream out = sock.getOutputStream();

while ( ( len = in.read(buf) ) != -1 ) {   
out.write(buf, 0, len);   
out.flush(); // manually flush output stream for sockets   
}   
out.close();   
sock.close();   
in.close();   
}

**Help on HashCode**

1. I came to know that for every object created there will be an hashcode assigned. is that anything concerned with   
memory address?

Yeah, a hashcode is assigned for every object created.  Identity and value of an Object are 2 important properties among others. Value may be the contents/state of the Object, Identity is that property of the Object using which you can distinguish   
it from every other Object. Hashcode of an Object does serve as Identity most of the times. And yes the algorithm used by the Object class to get the hascode is said to be based on the memory address of the Object.

Hashcode is used to find objects in data structures such as Hashtable etc(There are also other identity based uses for it). The designers of the language anticipated that collections like Hashtable will be used a lot by programmers and provided a easy way to compute hashcode.

There's lot more to it, try google'ing on hashcode() and equals().

2, Also when ever im printing an object of a class itz string representation is like this classname@23e45g   
My question is ,what that number signifies?

When you pass an Object to System.out.println what it prints is the return value of the toString method of that object. The above string representation which you have pasted is the default implementation of the toString() method in the Object class.

What you see is simple, its nothing but a concatenation of three things:   
1)the name of the class   
2) @   
3)hascode() of the object in Hexadecimal

The name of the class can be obtained by using the getName() method on the Object's Class object. You can obtain the class object for any object by invoking the getCLass() method.

The implementation is simple:

public String toString() {   
 return getClass().getName() + "@" + Integer.toHexString(hashCode());   
    }

**Use of a Static Block in a Class**

**Can anyone tell me that what is the use of a static block in a class?**   
**e.g.**   
   
**public class myclass{**   
   
**static{**   
   
**//some statements here**   
   
**}**   
   
**//some variables declared here**   
**//some functions defined here**   
**}**   
   
**This question was asked me in an interview. so if anyone knows the answer please give me the answer.**

Static loop is executed, when the class is loaded.   
So without creating the object we can execute this loop.   
eg:   
When we're loading a DB driver using Class.forName("");   
The driver is registered to DriverManager class,   
Using the registerDriver()method of DM.   
This process done in the static loop of driver class

Got confused !!! Leave that and try this simple example   
public class Stat   
{   
static int i=0;   
static{   
i=10;   
System.out.println(i);   
}   
public static void main(String []ss)   
{   
}   
}

Hopes you got cleared

**Jim**

If you need to do computation in order to initialize your static variables,you can declare a static block which gets executed exactly once,when the class is first loaded.

The following example shows a class that has a static method,some static variables,ana a static initialization block.   
    
class UseStatic {   
   static int a=3;   
   static int b;   
static void meth(int x) {   
   System.out.println("x = " + x);   
   System.out.println("a = " + a);   
   System.out.println("b = " + b);   
}   
    
static {   
  System.out.println("Static block initialized.");   
  b = a \* 4;   
}   
public static void main(String args[]) {   
  meth(42);   
 }   
    
}   
    
As soon as the UseStatic class is loaded, all of the static statements are run.First, a is set to 3, then the static block executes(printing a message),and finally, b is initialized to a\* 4 or 12.Then main( ) is called,which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b,as well as to the local varible x.   
    
Here is the output of the program:   
    
Static block initialized.   
x = 42   
a = 3   
b = 12

**Difference Between ArrayList and Vector**

**What is difference between ArrayList and vector?**

Ans: )

1) Synchronization - ArrayList is not thread-safe whereas Vector is thread-safe. In Vector class each method like add(), get(int i) is surrounded with a synchronized block and thus making Vector class thread-safe.

2) Data growth - Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.    
    
**How can Arraylist be synchronized without using Vector?**

Ans) Arraylist can be synchronized using:

• Collection.synchronizedList(List list)

Other collections can be synchronized:

• Collection.synchronizedMap(Map map)

• Collection.synchronizedCollection(Collection c)    
    
**If an Employee class is present and its objects are added in an arrayList. Now I want the list to be sorted on the basis of the employeeID of Employee class. What are the steps?**

Ans) 1) Implement Comparable interface for the Employee class and override the compareTo(Object obj) method in which compare the employeeID

2) Now call Collections.sort() method and pass list as an argument.

Now consider that Employee class is a jar file.

1) Since Comparable interface cannot be implemented, create Comparator and override the compare(Object obj, Object obj1) method .

2) Call Collections.sort() on the list and pass comparator as an argument.   
    
**What is difference between HashMap and HashTable?**

Ans) Both collections implements Map. Both collections store value as key-value pairs. The key differences between the two are

1. Access to the Hashtable is synchronized on the table while access to the HashMap isn't. You can add it, but it isn't there by default.

2. Another difference is that iterator in the HashMap is fail-safe while the enumerator for the Hashtable isn't. If you change the map while iterating, you'll know. • Fail-safe - “if the Hashtable is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException”

3. HashMap permits null values and only one null key, while Hashtable doesn't allow key or value as null.   
    
**What is difference between Arrays and ArrayList ?**

Ans) Arrays are created of fix size whereas ArrayList is of not fix size. It means that once array is declared as :

int [] intArray= new int[6];    
intArray[7]   // will give ArraysOutOfBoundException.    
Also the size of array cannot be incremented or decremented. But with arrayList the size is variable.   
Once the array is created elements cannot be added or deleted from it. But with ArrayList the elements can be added and deleted at runtime.    
List list = new ArrayList();   
list.add(1);   
list.add(3);   
list.remove(0) // will remove the element from the 1st location.

ArrayList is one dimensional but array can be multidimensional.    
            int[][][] intArray= new int[3][2][1];   // 3 dimensional array        
    
**When to use ArrayList or LinkedList ?**

Ans)  Adding new elements is pretty fast for either type of list. For the ArrayList, doing  random lookup using "get" is fast, but for LinkedList, it's slow. It's slow because there's no efficient way to index into the middle of a linked list. When removing elements, using ArrayList is slow. This is because all remaining elements in the underlying array of Object instances must be shifted down for each remove operation. But here LinkedList is fast, because deletion can be done simply by changing a couple of links. So an ArrayList works best for cases where you're doing random access on the list, and a LinkedList works better if you're doing a lot of editing in the middle of the list.

Source : Read More - from java.sun    
    
**Consider a scenario. If an ArrayList has to be iterate to read data only, what are the possible ways and which is the fastest?**

Ans) It can be done in two ways, using for loop or using iterator of ArrayList. The first option is faster than using iterator. Because value stored in arraylist is indexed access. So while accessing the value is accessed directly from the index.    
    
**Now another question with respect to above question is if accessing through iterator is slow then why do we need it and when to use it.**

Ans) For loop does not allow the updation in the array(add or remove operation) inside the loop whereas Iterator does. Also Iterator can be used where there is no clue what type of collections will be used because all collections have iterator.    
    
**Which design pattern Iterator follows?**

Ans) It follows Iterator design pattern. Iterator Pattern is a type of behavioral pattern. The Iterator pattern is one, which allows you to navigate through a collection of data using a common interface without knowing about the underlying implementation. Iterator should be implemented as an interface. This allows the user to implement it anyway its easier for him/her to return data. The benefits of Iterator are about their strength to provide a common interface for iterating through collections without bothering about underlying implementation.

Example of Iteration design pattern - Enumeration The class java.util.Enumeration is an example of the Iterator pattern. It represents and abstract means of iterating over a collection of elements in some sequential order without the client having to know the representation of the collection being iterated over. It can be used to provide a uniform interface for traversing collections of all kinds.    
    
**Why is it preferred to declare: List<String> list = new ArrayList<String>(); instead of ArrayList<String> = new ArrayList<String>();**

Ans) It is preferred because:

If later on code needs to be changed from ArrayList to Vector then only at the declaration place we can do that.    
The most important one – If a function is declared such that it takes list. E.g void showDetails(List list);   
When the parameter is declared as List to the function it can be called by passing any subclass of List like ArrayList,Vector,LinkedList making the function more flexible    
    
**How to sort list in reverse order?**

Ans) To sort the elements of the List in the reverse natural order of the strings, get a reverse Comparator from the Collections class with reverseOrder(). Then, pass the reverse Comparator to the sort() method.

List list = new ArrayList();

Comparator comp = Collections.reverseOrder();

Collections.sort(list, comp)

**How to sort list of strings - case insensitive?**

Ans) using Collections.sort(list, String.CASE\_INSENSITIVE\_ORDER);   
    
**Can a null element added to a set ?**

Ans) A null element can be added only if the set contains one element because when a second element is added then as per set defination a check is made to check duplicate value and comparison with null element will throw NullPointerException.

|  |  |
| --- | --- |
| **Q:** | **What is the difference between an Interface and an Abstract class?** |
| **A:** | An abstract class can have instance methods that implement a default behavior. An Interface can only declare constants and instance methods, but cannot implement default behavior and all methods are implicitly abstract. An interface has all public members and no implementation. An abstract class is a class which may have the usual flavors of class members (private, protected, etc.), but has some abstract methods. |

|  |  |
| --- | --- |
| **Q:** | **What is static in java?** |
| **A:** | Static means one per class, not one for each object no matter how many instance of a class might exist. This means that you can use them without creating an instance of a class. Static methods are implicitly final, because overriding is done based on the type of the object, and static methods are attached to a class, not an object. A static method in a superclass can be shadowed by another static method in a subclass, as long as the original method was not declared final. However, you can't override a static method with a nonstatic method. In other words, you can't change a static method into an instance method in a subclass. |

# Increase heap size in Java to prevent java.lang.OutOfMemoryError

If Java runs out of memory, the following error occurs:

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

This can have two reasons:

* Your Java application has a memory leak. There are tools like [YourKit Java Profiler](http://www.yourkit.com/) that help you to identify such leaks.
* Your Java application really needs a lot of memory (more than 128 MB by default!). In this case the Java heap size can be increased using the following runtime parameters:

java -Xms<initial heap size> -Xmx<maximum heap size>

Defaults are:

java -Xms32m -Xmx128m

You can set this either in the Java Control Panel or on the command line, depending on the environment you run your application

For example, you can set minimum heap to 64 MB and maximum heap 256 MB for a Java program HelloWorld.

java -Xms64m -Xmx256m HelloWorld

## Getting / Reading default heap size

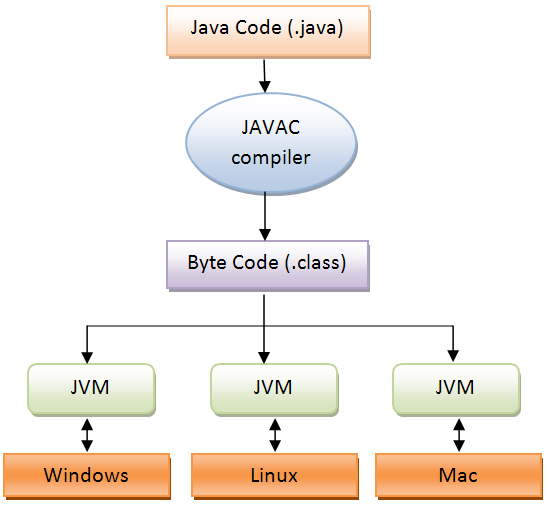
It is possible to read the default JVM heap size programmatically by using totalMemory() method of Runtime class. Use following code to read JVM heap size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | public class GetHeapSize { | | | |
|  | public static void main(String[]args){ | | | | |
|  | |  |
|  | | //Get the jvm heap size. | |
|  | long heapSize = Runtime.getRuntime().totalMemory(); | | | | | | |
|  |  | | | | | |

|  |  |
| --- | --- |
|  | //Print the jvm heap size. |
|  | System.out.println("Heap Size = " + heapSize); | |

|  |  |  |
| --- | --- | --- |
|  | } | |
|  | } |

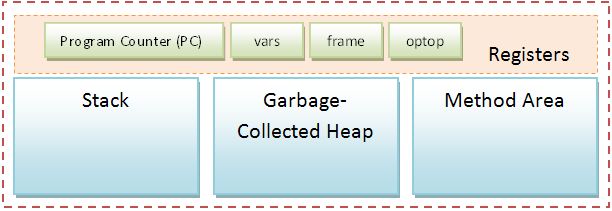
## Java Virtual Machine, An inside story!!



## Java Virtual Machine

Java Virtual Machine like its real counter part, **executes** the program and generate output. To execute any code, JVM utilizes different components.

JVM is divided into several components like the stack, the garbage-collected heap, the registers and the method area. Let us see diagram representation of JVM.



### The Stack

Stack in Java virtual machine stores various method arguements as well as the local variables of any method. Stack also keep track of each an every method invocation. This is called **Stack Frame**. There are three registers thats help in stack manipulation. They are vars, frame, optop. This registers points to different parts of current Stack.

There are three sections in Java stack frame:

#### Local Variables

The local variables section contains all the local variables being used by the current method invocation. It is pointed to by the **vars** register.

#### Execution Environment

The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the **frame** register.

#### Operand Stack

The operand stack is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed, and results of bytecode instructions are found. The top of the operand stack is pointed to by the **optop** register.

### Method Area

This is the area where bytecodes reside. The program counter points to some byte in the method area. It always keep tracks of the current instruction which is being executed (interpreted). After execution of an instruction, the JVM sets the PC to next instruction. Method area is shared among all the threads of a process. Hence if more then one threads are accessing any specific method or any instructions, synchorization is needed. Synchronization in JVM is acheived through Monitors.

### Garbage-collected Heap

The Garbage-collected Heap is where the objects in Java programs are stored. Whenever we allocate an object using new operator, the heap comes into picture and memory is allocated from there. Unlike C++, Java does not have free operator to free any previously allocated memory. Java does this automatically using Garbage collection mechanism. Till Java 6.0, **mark and sweep** algorithm is used as a garbage collection logic. Remember that the local object reference resides on Stack but the actual object resides in Heap only. Also, arrays in Java are objects, hence they also resides in Garbage-collected Heap.

**Parsing XML String using DOM**

The first step in parsing an XML String using DOM is to get the **org.w3c.dom.Document** object from our String object. Here I am going to parse the following XML which is stored in a String variable xmlString

**<data>**

**<address>**

**<name>**Tom**</name>**

**<city>**Bangalore**</city>**

**</address>**

**<address>**

**<name>**Chris**</name>**

**<city>**New Jersey**</city>**

**</address>**

**</data>**

First lets create a Document object using xmlString object.

javax.xml.parsers.DocumentBuilderFactory factory =

javax.xml.parsers.DocumentBuilderFactory.newInstance();

javax.xml.parsers.DocumentBuilder db = factory.newDocumentBuilder();

org.xml.sax.InputSource inStream = **new** org.xml.sax.InputSource();

inStream.setCharacterStream(**new** java.io.StringReader(xmlString));

org.xml.sax.Document doc = db.parse(inStream);

Once we got the Document object we need to get the NodeList from our Document object.

org.w3c.dom.NodeList nodeList = doc.getElementsByTagName("address");

This would return all the address elements available in our XML. Next we need to loop through all the nodes in our NodeList and get the nodes present.

**for**(**int** index=0; index < nodeList.getLength(); index++) {

org.w3c.dom.Node node = nodeList.item(index);

}

Once we got the Node we need to cast it to an org.w3c.dom.Element object if the Node is of type org.w3c.dom.Element

**if** (node.getNodeType() == org.w3c.dom.Node.ELEMENT\_NODE) {

org.w3c.dom.Element element = (org.w3c.dom.Element) node;

}

Now we need to find out all the names that is coming under the current address.

org.w3c.dom.NodeList nameNode = element.getElementsByTagName("name");

Now we need to do all the above exercise again to get the final element from where we can get the value.

**for**(**int** iIndex=0; iIndex< nameNode.getLength(); iIndex++) {

**if** (nameNode.item(iIndex).getNodeType() == org.w3c.dom.Node.ELEMENT\_NODE) {

org.w3c.dom.Element nameElement = (org.w3c.dom.Element) nameNode.item(iIndex);

System.out.println("Name = " +nameElement.getFirstChild().getNodeValue().trim());

}

}

So the complete code is

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder db = factory.newDocumentBuilder();

InputSource inStream = **new** InputSource();

inStream.setCharacterStream(**new** StringReader(xmlString));

Document doc = db.parse(inStream);

NodeList nodeList = doc.getElementsByTagName("address");

**for**(**int** index=0; index < nodeList.getLength(); index++)

{

Node node = nodeList.item(index);

**if** (node.getNodeType() == Node.ELEMENT\_NODE)

{

Element element = (Element) node;

NodeList nameNode = element.getElementsByTagName("name");

**for**(**int** iIndex=0; iIndex< nameNode.getLength(); iIndex++)

{

**if** (nameNode.item(iIndex).getNodeType() ==Node.ELEMENT\_NODE)

{

Element nameElement = (Element) nameNode.item(iIndex);

System.out.println("Name = " +nameElement.getFirstChild().getNodeValue().trim());

}

}

}

}

**Example:**

import java.io.\*;

import java.util.\*;

import java.io.StringReader;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.CharacterData;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.Node;

import org.w3c.dom.NodeList;

import org.xml.sax.InputSource;

import java.net.\*;

import org.xml.sax.SAXException;

public class NasdaqXMLParser

{

public static void main(String a[])

{

try

{

System.setProperty("http.proxyHost","10.112.62.73");

System.setProperty("http.proxyPort","8080");

URL nasdaq = new URL("http://www.nasdaq.com/aspxcontent/nasdaqRSS.aspx?data=quotes&symbol=infy&symbol=ABB");

BufferedReader in = new BufferedReader(

new InputStreamReader(

nasdaq.openStream()));

String inputLine;

String data = new String();

while((inputLine = in.readLine()) != null)

{

data = data.concat(""+ inputLine);

}

System.out.println(data);

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder db = factory.newDocumentBuilder();

InputSource is = new InputSource();

is.setCharacterStream(new StringReader(data));

Document doc = db.parse(is);

System.out.println(doc);

System.out.println("hello everyone");

NodeList nodeList = doc.getElementsByTagName("table");

for(int i=0; i<nodeList.getLength();i++)

{

System.out.println("inside for loop");

Node node = nodeList.item(i);

System.out.println(node.getNodeValue().trim());

Element element = (Element)node;

NodeList rowNode = element.getElementsByTagName("tr");

for(int index=0;index<rowNode.getLength();index++) {

Node node1 = rowNode.item(index);

Element rowElement = (Element)node1;

NodeList columnNode = rowElement.getElementsByTagName("td");

for(int iindex=0;iindex<columnNode.getLength();iindex++)

{

Element colElement = (Element)columnNode.item(iindex);

System.out.println(colElement.getFirstChild().getNodeValue().trim());

}

}

//System.out.println(node.getNodeValue().trim());

}

}catch(Exception e)

{

e.printStackTrace();

}

}

}