String StringBuffer

Immutable Mutable

String s=new String("karthik"); StringBuffer sb=new StringBuffer("karthik")

s.concat("reddy"); sb.append("reddy");

System.out.println(s); System.out.println(sb);

O/P:karthik O/P:karthikreddy

--->once we created a String object ---->once we created a StringBuffer object

we can't perform any changes in the existing we can perform any changes in the existing

object.If we are trying to perform any object.It is nothing but mutablity of

changes with those changes a new object of a StrongBuffer object

will be created.It is nothing but Immutability

of a String object

Use String--->If you require immutabilty

Use StringBuffer---->If you require mutable + threadsafety

Use StringBuilder--->If you require mutable + with out threadsafety

String s=new String("karthik");

--->here 2 objects will be created one is heap and the other is in stringconstantpool(scp) and s is always pointing to heap object

String s="karthik";

--->In this case only one object will be created in scp and s is always pointing to that object only

While I understand that this is not a major differentiating factor, I noticed today that StringBuffer(and StringBuilder) provides some interesting methods that String doesn't.

* reverse()
* setCharAt(

The differences are

1. Only in **String** class **+** operator is overloaded. We can concat two String object using **+**operator, but in the case of **StringBuffer** we can't.
2. **String** class is overriding toString(), equals(), hashCode() of **Object** class, but **StringBuffer**only overrides toString().
3. String s1 = new String("abc");
4. String s2 = new String("abc");
5. System.out.println(s1.equals(s2)); // output true
6. StringBuffer sb1 = new StringBuffer("abc");
7. StringBuffer sb2 = new StringBuffer("abc");

System.out.println(sb1.equals(sb2)); // output false

1. **String** class is both **Serializable** as well as **Comparable**, but **StringBuffer** is only **Serializable**.
2. Set<StringBuffer> set = new TreeSet<StringBuffer>();
3. set.add(sb1);
4. set.add(sb2);

System.out.println(set); // gives ClassCastException because there is no Comparison mechanism

1. We can create a String object with and without **new** operator, but StringBuffer object can only be created using **new** operator.
2. String is immutable but StringBuffer is mutable.
3. StringBuffer is synchronized, whereas String ain't.
4. StringBuffer is having an in-built **reverse()** method, but String dosen't have it.

**String Literal Vs String Object**

Both expression gives you String object, but there is subtle difference between them. When you use **new String( "Hello World!!" );** , it explicitly creates a new and referentially distinct instance of a String object. It is an individual instance of the java.lang.String class. **String s="Hello World!!";** may reuse an instance from the string constant pool if one is available (String Pool is a **pool of Strings** stored in Java heap memory ).

In this example both string literals refer the same object:



In the following code, 2 different objects are created and they have different references:



In terms of good coding practice: do not use == to check for String equality, use **.equals()** instead.

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

String s1 = "abcde";

String s2 = new String("abcde");

String s3 = "abcde";

All are valid, but have a slight difference. s1 will refer to an interned String object. This means, that the character sequence "abcde" will be stored at a central place, and whenever the same literal "abcde" is used again, the JVM will not create a new String object but use the reference of the cached String.

s2 is guranteed to be a new String object, so in this case we have:

s1 == s2 // is false

s1 == s3 // is true

s1.equals(s2) // is true

|  |  |
| --- | --- |
| up vote7down vote | String is a class in Java different from other programming languages. So as for every class the object declaration and initialization is  String st1 = new String();  or  String st2 = new String("Hello");  String st3 = new String("Hello");  Here, st1, st2 and st3 are different objects.  That is:  st1 == st2 // false  st1 == st3 // false  st2 == st3 // false  Because st1, st2, st3 are referencing 3 different objects, and == checks for the equality in memory location, hence the result.  But:  st1.equals(st2) // false  st2.equals(st3) // true  Here .equals() method checks for the content, and the content of st1 = "", st2 = "hello" and st3 = "hello". Hence the result.  And in the case of the String declaration  String st = "hello";  Here, intern() method of String class is called, and checks if "hello" is in intern pool, and if not, it is added to intern pool, and if "hello" exist in intern pool, then st will point to the memory of the existing "hello".  So in case of:  String st3 = "hello";  String st4 = "hello";  Here:  st3 == st4 // true  Because st3 and st4 pointing to same memory address.  Also:  st3.equals(st4); // true as usual |

The following are some comparisons:

String s1 = "Hello";

String s2 = "Hello";

String s3 = new String("Hello");

System.out.println(s1 == s2); //true

System.out.println(s1.equals(s2)); //true

System.out.println(s1 == s3); //false

System.out.println(s1.equals(s3)); //true

s3 = s3.intern();

System.out.println(s1 == s3); //true

System.out.println(s1.equals(s3)); //true

When intern() is called the reference is changed.

Reverse a String

String string="whatever";

String reverse = new StringBuffer(string).reverse().toString();

System.out.println(reverse);

public static String reverseIt(String source) {

int i, len = source.length();

StringBuilder dest = new StringBuilder(len);

for (i = (len - 1); i >= 0; i--){

dest.append(source.charAt(i));

}

return dest.toString();

}

**Reverse string by WORDS:**

public static void reverseStringByWords(String string) {

StringBuilder stringBuilder = new StringBuilder();

String[] words = string.split(" ");

for (int j = words.length-1; j >= 0; j--) {

stringBuilder.append(words[j]).append(' ');

}

System.out.println("Reverse words: " + stringBuilder);

}

Replace all occurances of a string in list

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** Main {

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

ArrayList<String> arrayList = **new** ArrayList<String>();

arrayList.add("A");

arrayList.add("B");

arrayList.add("A");

arrayList.add("C");

arrayList.add("D");

System.out.println(arrayList);

Collections.replaceAll(arrayList, "A", "Replace All");

System.out.println(arrayList);

}

}

/\*

[A, B, A, C, D]

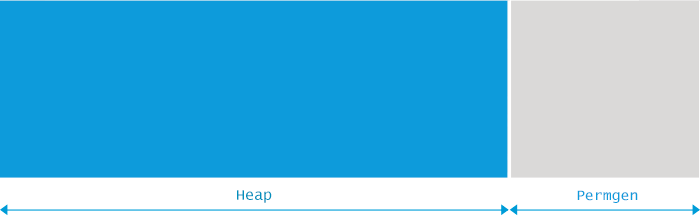
[Replace All, B, Replace All, C, D]

\*/

Out Of Memory Error :

# java.lang.OutOfMemoryError:**Java heap space**

Java applications are only allowed to use a limited amount of memory. This limit is specified during application startup. To make things more complex, Java memory is separated into two different regions. These regions are called Heap space and Permgen (for Permanent Generation):



The size of those regions is set during the Java Virtual Machine (JVM) launch and can be customized by specifying JVM parameters -Xmx and -XX:MaxPermSize. If you do not explicitly set the sizes, platform-specific defaults will be used.

The java.lang.OutOfMemoryError: Java heap space error will be triggered when the application **attempts to add more data into the heap space area, but there is not enough room for it**.

Note that there might be plenty of physical memory available, but the java.lang.OutOfMemoryError: Java heap space error is thrown whenever the JVM reaches the heap size limit.

## What is causing it?

There most common reason for the java.lang.OutOfMemoryError: Java heap spaceerror is simple – you try to fit an XXL application into an S-sized Java heap space. That is – the application just requires more Java heap space than available to it to operate normally. Other causes for this OutOfMemoryError message are more complex and are caused by a programming error:

* **Spikes in usage/data volume**. The application was designed to handle a certain amount of users or a certain amount of data. When the number of users or the volume of data suddenly spikes and crosses that expected threshold, the operation which functioned normally before the spike ceases to operate and triggers the java.lang.OutOfMemoryError: Java heap space error.
* **Memory leaks**. A particular type of programming error will lead your application to constantly consume more memory. Every time the leaking functionality of the application is used it leaves some objects behind into the Java heap space. Over time the leaked objects consume all of the available Java heap space and trigger the already familiar java.lang.OutOfMemoryError: Java heap space error.

## Give me an example

### Trivial example

The first example is truly simple – the following Java code tries to allocate an array of 2M integers. When you compile it and launch with 12MB of Java heap space (java -Xmx12m OOM), it fails with the java.lang.OutOfMemoryError: Java heap spacemessage. With 13MB Java heap space the program runs just fine.

class OOM {

static final int SIZE=2\*1024\*1024;

public static void main(String[] a) {

int[] i = new int[SIZE];

}

}

### Memory leak example

The second and a more realistic example is of a memory leak. In Java, when developers create and use new objects e.g. new Integer(5), they don’t have to allocate memory themselves – this is being taken care of by the Java Virtual Machine (JVM). During the life of the application the JVM periodically checks which objects in memory are still being used and which are not. Unused objects can be discarded and the memory reclaimed and reused again. This process is called [Garbage Collection](https://plumbr.io/handbook/what-is-garbage-collection). The corresponding module in JVM taking care of the collection is called the [Garbage Collector (GC)](https://plumbr.io/handbook/garbage-collection-algorithms).

Java’s automatic memory management relies on [GC](https://plumbr.io/java-garbage-collection-handbook) to periodically look for unused objects and remove them. Simplifying a bit we can say that a **memory leak in Java is a situation where some objects are no longer used by the application but**[**Garbage Collection**](https://plumbr.io/handbook/garbage-collection-in-jvm)**fails to recognize it**. As a result these unused objects remain in Java heap space indefinitely. This pileup will eventually trigger the java.lang.OutOfMemoryError: Java heap space error.

It is fairly easy to construct a Java program that satisfies the definition of a memory leak:

class KeylessEntry {

static class Key {

Integer id;

Key(Integer id) {

this.id = id;

}

@Override

public int hashCode() {

return id.hashCode();

}

}

public static void main(String[] args) {

Map m = new HashMap();

while (true)

for (int i = 0; i < 10000; i++)

if (!m.containsKey(new Key(i)))

m.put(new Key(i), "Number:" + i);

}

}

When you execute the above code above you might expect it to run forever without any problems, assuming that the naive caching solution only expands the underlying Map to 10,000 elements, as beyond that all the keys will already be present in the HashMap. However, in reality the elements will keep being added as the Key class does not contain a proper equals() implementation next to its hashCode().

As a result, over time, with the leaking code constantly used, the “cached” results end up consuming a lot of Java heap space. And when the leaked memory fills all of the available memory in the heap region and [Garbage Collection](https://plumbr.io/handbook/what-is-garbage-collection) is not able to clean it, the java.lang.OutOfMemoryError:Java heap space is thrown.

The solution would be easy – add the implementation for the equals() method similar to the one below and you will be good to go. But before you manage to find the cause, you will definitely have lose some precious brain cells.

@Override

public boolean equals(Object o) {

boolean response = false;

if (o instanceof Key) {

response = (((Key)o).id).equals(this.id);

}

return response;

}

You could specify per project how much heap space your project wants

Following is for **Eclipse Helios/Juno/Kepler**:

Right mouse click on

Run As - Run Configuration - Arguments - Vm Arguments,

then add this

-Xmx2048m

Increasing the heap size is not a "fix" it is a "plaster", 100% temporary. It will crash again in somewhere else. To avoid these issues, write high performance code.

1. Use local variables wherever possible.
2. Make sure you select the correct object (EX: Selection between String, StringBuffer and StringBuilder)
3. Use a good code system for your program(EX: Using static variables VS non static variables)
4. Other stuff which could work on your code.
5. Try to move with multy THREADING

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Follow below steps:   1. Open catalina.sh from tomcat/bin. 2. Chnage JAVA\_OPTS to 3. JAVA\_OPTS="-Djava.awt.headless=true -Dfile.encoding=UTF-8 -server -Xms1536m 4. -Xmx1536m -XX:NewSize=256m -XX:MaxNewSize=256m -XX:PermSize=256m   -XX:MaxPermSize=256m -XX:+DisableExplicitGC"   1. Restart your tomcat   In Java, there are two types of exceptions – checked and unchecked exception. Here’s the summary :   1. **Checked** – Extends java.lang.Exception, for recoverable condition, try-catch the exception explicitly, compile error. 2. **Unchecked** – Extends java.lang.RuntimeException, for unrecoverable condition, like programming errors, no need try-catch, runtime error.  1. Custom Checked Exception **Note** Some popular checked exception : IOException, FileNotFoundException  1.1 If the client is able to recover from the exception, make it a checked exception. To create a custom checked exception, extends java.lang.Exception  NameNotFoundException.java  package com.mkyong.examples.exception;  public class NameNotFoundException extends Exception {  public NameNotFoundException(String message) {  super(message);  }  }  1.2 For checked exception, you need to try and catch the exception.  CustomerService.java  package com.mkyong.examples;  import com.mkyong.examples.exception.NameNotFoundException;  public class CustomerService {  public Customer findByName(String name) throws NameNotFoundException {  if ("".equals(name)) {  throw new NameNotFoundException("Name is empty!");  }  return new Customer(name);  }  public static void main(String[] args) {  CustomerService obj = new CustomerService();  try {  Customer cus = obj.findByName("");  } catch (NameNotFoundException e) {  e.printStackTrace();  }  }  }  Output  com.mkyong.examples.exception.NameNotFoundException: Name is empty!  at com.mkyong.examples.CustomerService.findByName(CustomerService.java:10)  at com.mkyong.examples.CustomerService.main(CustomerService.java:39)  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  at java.lang.reflect.Method.invoke(Method.java:498)  at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144) 2. Custom Unchecked Exception **Note** Some popular unchecked exception : NullPointerException, IndexOutOfBoundsException, IllegalArgumentException  2.1 If the client cannot do anything to recover from the exception, make it an unchecked exception. To create a custom unchecked exception, extends java.lang.RuntimeException  ListTooLargeException.java  package com.mkyong.examples.exception;  public class ListTooLargeException extends RuntimeException{  public ListTooLargeException(String message) {  super(message);  }  }  2.3 For unchecked exception, try and catch the exception is optional.  CustomerService.java  package com.mkyong.examples;  import com.mkyong.examples.exception.ListTooLargeException;  import java.util.ArrayList;  import java.util.Collections;  import java.util.List;  public class CustomerService {  public void analyze(List<String> data) {  if (data.size() > 50) {  //runtime exception  throw new ListTooLargeException("List can't exceed 50 items!");  }  //...  }  public static void main(String[] args) {  CustomerService obj = new CustomerService();  //create 100 size  List<String> data = new ArrayList<>(Collections.nCopies(100, "mkyong"));  obj.analyze(data);  }  }  Output  Exception in thread "main" com.mkyong.examples.exception.ListTooLargeException: List can't exceed 50 items!  at com.mkyong.examples.CustomerService.analyze(CustomerService.java:25)  at com.mkyong.examples.CustomerService.main(CustomerService.java:38)  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  at java.lang.reflect.Method.invoke(Method.java:498)  at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)  Can we create CustmException by extending Thowable class ?  Throwable is a class for all the bad situations, which can arise: Errors & Exceptions.  Error is something, you **can't handle at all**: OutOfMemoryError, VirtualMachineError, etc.  Exception is for **exceptional cases**.  Exceptions come in 2 flavours:   1. RuntimeExceptions.   These ones, you are not aware of: NullPointerException, ClassCastException, etc.   1. Checked exceptions.   These are the exceptions, which your code is aware of and should be explicitely catched (... throws MyException): IOExceptions, etc.  If you want the users of your code, to explicitely handle some exceptional situations, it would be good to just extend Exception, not the RuntimeException. There's no need to extend Throwable  Throwable is the super class of Error & Exception.  Like Exception, Error too, can be thrown & handled.  But it is not advisable, according to the following doc:  You are not required to catch Error objects or Error subtypes. You can also throw an Error yourself (although other than AssertionError you probably won't ever want to), and you can catch one, but again, you probably won't. What, for example, would you actually do if you got an OutOfMemoryError?  Keeping this concept in mind, I would suggest to extend Throwable if you want to throw and/or catch Exception & Error both. Extend Exception if you want to throw and/or catch Exceptiononly.  **java.util.ConcurrentModificationException**  java.util.ConcurrentModificationException is a very common exception when working with java collection classes. Java Collection classes are fail-fast, which means if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw **ConcurrentModificationException**. Concurrent modification exception can come in case of multithreaded as well as single threaded java programming environment.  Let’s see the concurrent modification exception scenario with an example.  import java.util.\*;  public class IteratorExample {  public static void main(String args[]){  List<String> myList = new ArrayList<String>();    myList.add("1");  myList.add("2");  myList.add("3");  myList.add("4");  myList.add("5");    Iterator<String> it = myList.iterator();  while(it.hasNext()){  String value = it.next();  System.out.println("List Value:"+value);  if(value.equals("3")) myList.remove(value);  }    Map<String,String> myMap = new HashMap<String,String>();  myMap.put("1", "1");  myMap.put("2", "2");  myMap.put("3", "3");  Iterator<String> it1 = myMap.keySet().iterator();  while(it1.hasNext()){  String key = it1.next();  System.out.println("Map Value:"+myMap.get(key));  if(key.equals("2")){  myMap.put("1","4");  //myMap.put("4", "4");  }  }  }  }  Above program will throw java.util.ConcurrentModificationException when executed, as shown in below console logs.  List Value:1  List Value:2  List Value:3  Exception in thread "main" java.util.ConcurrentModificationException  at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)  at java.util.AbstractList$Itr.next(AbstractList.java:343)  at com.journaldev.java.IteratorExample.main(IteratorExample.java:27)  From the output stack trace, its clear that the concurrent modification exception is coming when we call iterator next() function. If you are wondering how Iterator checks for the modification, its implementation is present in AbstractList class where an int variable modCount is defined. modCount provides the number of times list size has been changed. modCount value is used in every next() call to check for any modifications in a function checkForComodification().  Now comment the list part and run the program again. You will see that there is no ConcurrentModificationException being thrown now.  Output will be:  Map Value:3  Map Value:2  Map Value:4  Since we are updating the existing key value in the myMap, its size has not been changed and we are not getting **ConcurrentModificationException**. Note that the output may differ in your system because HashMap keyset is not ordered like list. If you will uncomment the statement where I am adding a new key-value in the HashMap, it will cause ConcurrentModificationException. To Avoid ConcurrentModificationException in multi-threaded environment  1. You can convert the list to an array and then iterate on the array. This approach works well for small or medium size list but if the list is large then it will affect the performance a lot. 2. You can lock the list while iterating by putting it in a synchronized block. This approach is not recommended because it will cease the benefits of multithreading. 3. If you are using JDK1.5 or higher then you can use **ConcurrentHashMap** and **CopyOnWriteArrayList**classes. This is the recommended approach to avoid concurrent modification exception.  To Avoid ConcurrentModificationException in single-threaded environment You can use the iterator remove() function to remove the object from underlying collection object. But in this case you can remove the same object and not any other object from the list.  Let us run an example using Concurrent Collection classes:  package com.journaldev.java;  import java.util.Iterator;  import java.util.List;  import java.util.Map;  import java.util.concurrent.ConcurrentHashMap;  import java.util.concurrent.CopyOnWriteArrayList;  public class ThreadSafeIteratorExample {  public static void main(String[] args) {  List<String> myList = new CopyOnWriteArrayList<String>();    myList.add("1");  myList.add("2");  myList.add("3");  myList.add("4");  myList.add("5");    Iterator<String> it = myList.iterator();  while(it.hasNext()){  String value = it.next();  System.out.println("List Value:"+value);  if(value.equals("3")){  myList.remove("4");  myList.add("6");  myList.add("7");  }  }  System.out.println("List Size:"+myList.size());    Map<String,String> myMap = new ConcurrentHashMap<String,String>();  myMap.put("1", "1");  myMap.put("2", "2");  myMap.put("3", "3");    Iterator<String> it1 = myMap.keySet().iterator();  while(it1.hasNext()){  String key = it1.next();  System.out.println("Map Value:"+myMap.get(key));  if(key.equals("1")){  myMap.remove("3");  myMap.put("4", "4");  myMap.put("5", "5");  }  }    System.out.println("Map Size:"+myMap.size());  }  }  Output of above program is shown below. You can see that there is no ConcurrentModificationException being thrown by the program.  List Value:1  List Value:2  List Value:3  List Value:4  List Value:5  List Size:6  Map Value:1  Map Value:null  Map Value:4  Map Value:2  Map Size:4  From the above example its clear that:   1. Concurrent Collection classes can be modified safely, they will not throw ConcurrentModificationException. 2. In case of CopyOnWriteArrayList, iterator doesn’t accommodate the changes in the list and works on the original list. 3. In case of ConcurrentHashMap, the behaviour is not always the same.   For condition:  if(key.equals("1")){  myMap.remove("3");}  Output is:  Map Value:1  Map Value:null  Map Value:4  Map Value:2  Map Size:4  It is taking the new object added with key “4” but not the next added object with key “5”.  Now if I change the condition to below.  if(key.equals("3")){  myMap.remove("2");}  Output is:  Map Value:1  Map Value:3  Map Value:null  Map Size:4  In this case its not considering the new added objects.  So if you are using ConcurrentHashMap then avoid adding new objects as it can be processed depending on the keyset. Note that the same program can print different values in your system because HashMap keyset is not ordered. Use for loop to avoid java.util.ConcurrentModificationException If you are working on single-threaded environment and want your code to take care of the extra added objects in the list then you can do so using for loop rather than iterator.  for(int i = 0; i<myList.size(); i++){  System.out.println(myList.get(i));  if(myList.get(i).equals("3")){  myList.remove(i);  i--;  myList.add("6");  }  }  Note that I am decreasing the counter because I am removing the same object, if you have to remove the next or further far object then you don't need to decrease the counter. Try it yourself.  Difference between ArrayList and Vector  ArrayList and Vector both implements List interface and maintains insertion order.  But there are many differences between ArrayList and Vector classes that are given below.   |  |  | | --- | --- | | **ArrayList** | **Vector** | | 1) ArrayList is **not synchronized**. | Vector is **synchronized**. | | 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. | | 3) ArrayList is **not a legacy**class, it is introduced in JDK 1.2. | Vector is a **legacy** class. | | 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. | | 5) ArrayList uses **Iterator**interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. | |

Java Transient Keyword

**Java transient** keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

### Example of Java Transient Keyword

In this example, we have created the two classes Student and PersistExample. The age data member of the Student class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for transient variable.

Let's create a class with transient variable.

1. **import** java.io.Serializable;
2. **public** **class** Student **implements** Serializable{
3. **int** id;
4. String name;
5. **transient** **int** age;//Now it will not be serialized
6. **public** Student(**int** id, String name,**int** age) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.age=age;
10. }
11. }

Now write the code to serialize the object.

1. **import** java.io.\*;
2. **class** PersistExample{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. Student s1 =**new** Student(211,"ravi",22);//creating object
5. //writing object into file
6. FileOutputStream f=**new** FileOutputStream("f.txt");
7. ObjectOutputStream out=**new** ObjectOutputStream(f);
8. out.writeObject(s1);
9. out.flush();
11. out.close();
12. f.close();
13. System.out.println("success");
14. }
15. }

Output:

success

Now write the code for deserialization.

1. **import** java.io.\*;
2. **class** DePersist{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt"));
5. Student s=(Student)in.readObject();
6. System.out.println(s.id+" "+s.name+" "+s.age);
7. in.close();
8. }
9. }

211 ravi 0

As you can see, printing age of the student returns 0 because value of age was not serialized.