jAVA ::

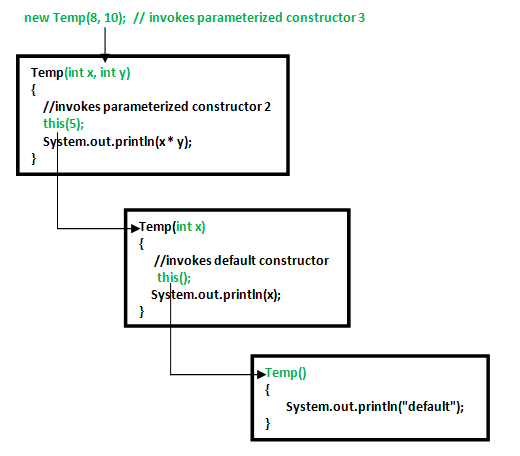
# Rules for writing Constructor:

* Constructor(s) of a class must has same name as the class name in which it resides.
* A constructor in Java can not be abstract, final, static and Synchronized.
* Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

# Does constructor return any value?

* There are no “return value” statements in constructor, but constructor returns current class instance. We can write ‘return’ inside a constructor.

Why do we need constructor chaining ?  
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.



# toString

You're not explicitly calling toString(), but implicitly you are:

See:

System.out.println(foo); // foo is a non primitive variable

[System](http://docs.oracle.com/javase/7/docs/api/java/lang/System.html) is a class, with a static field [out](http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#out), of type [PrintStream](http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html). So you're calling the [println(Object)](http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html" \l "println%28java.lang.Object%29) method of a PrintStream.

It is implemented like this:

public void println(Object x) {

String s = String.valueOf(x);

synchronized (this) {

print(s);

newLine();

}

}

As we see, it's calling the [String.valueOf(Object)](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \l "valueOf%28java.lang.Object%29) method.  
This is implemented as follows:

public static String valueOf(Object obj) {

return (obj == null) ? "null" : obj.toString();

}

And here you see, that [toString()](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "toString%28%29) is called.

# finalize() method In Java:

finalize() method is a protected and non-static method of **java.lang.Object** class. This method will be available in all objects you create in java. This method is used to perform some final operations or clean up operations on an object before it is removed from the memory.  you can override the finalize() method to keep those operations you want to perform before an object is destroyed. Here is the general form of finalize() method.

|  |  |
| --- | --- |
| 1  2  3  4 | protected void finalize() throws Throwable  {      //Keep some resource closing operations here  } |

# Garbage Collection In Java :

Whenever you run a java program, JVM creates three threads. 1) main thread   2) Thread Scheduler   3) Garbage Collector Thread. In these three threads, main thread is a user thread and remaining two are daemon threads which run in background.

The task of main thread is to execute the main() method. The task of thread scheduler is to schedule the threads. The task of garbage collector thread is to sweep out abandoned objects from the heap memory. Abandoned objects or dead objects are those objects which does not have live references. Garbage collector thread before sweeping out an abandoned object, it calls finalize() method of that object. After finalize() method is executed, object is destroyed from the memory. That means clean up operations which you have kept in the finalize() method are executed before an object is destroyed from the memory.

Garbage collector thread does not come to heap memory whenever an object becomes abandoned. It comes once in a while to the heap memory and at that time if it sees any abandoned objects, it sweeps out those objects after calling finalize() method on them. Garbage collector thread calls finalize() method only once for one object.

Let’s discuss some interesting points about garbage collection and finalize() method.

**Some Interesting Points About Garbage Collection And finalize() method In Java :**

***1)*** In some scenarios, finalize() method is not at all called by the garbage collector thread. For example, When I executed the below program in my system, finalize() method of Class A is not at all executed.

|  |
| --- |
| class A  {      int i = 50;        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method");      }  }    public class Test  {     public static void main(String[] args)     {        //Creating two instances of class A          A a1 = new A();          A a2 = new A();          //Assigning a2 to a1          a1 = a2;          //Now both a1 and a2 will be pointing to same object          //An object earlier referred by a1 will become abandoned          System.out.println("done");     }  } |

***2)*** You can make finalize() method to be executed forcefully using either **Runtime.getRuntime().runFinalization()** OR **Runtime.runFinalizersOnExit(true)**. But, both the methods have disadvantages. Runtime.getRuntime().runFinalization() makes the just best effort to execute finalize() method. It is not gauranteed that it will execute finalize() method. Runtime.runFinalizersOnExit(true) is deprecated in JDK because some times it runs finalize() method on live objects also.

|  |
| --- |
| class A  {      int i = 50;        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method");      }  }    public class Test  {     public static void main(String[] args)     {        //Creating two instances of class A          A a1 = new A();          A a2 = new A();          //Assigning a2 to a1          a1 = a2;          //Making finalize() method to execute forcefully          Runtime.getRuntime().runFinalization();          System.out.println("done");     }  } |

***3)*** you can call garbage collector explicitly using **System.gc()** or **RunTime.getRunTime().gc()**. Again it is just a request to garbage collector not a command. It is up to garbage collector to honour this request.

|  |
| --- |
| class A  {      int i;        public A(int i)      {          this.i = i;      }        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method, i = "+i);      }  }    public class Test  {     public static void main(String[] args)     {         //Creating two instances of class A           A a1 = new A(10);           A a2 = new A(20);           //Assigning a2 to a1           a1 = a2;           //Now both a1 and a2 will be pointing same object           //An object earlier referred by a1 will become abandoned               //Calling garbage collector thread explicitly           System.gc();              //OR call Runtime.getRuntime().gc();           System.out.println("done");     }  } |

***4)*** finalize() methods are not chained like constructors.i.e there is no calling statement to super class finalize() method inside the finalize() method of sub class. You need to explicitly call super class finalize() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | protected void finalize() throws Throwable  {      System.out.println("From Finalize Method");        //Calling super class finalize() method explicitly        super.finalize();  } |

***5)*** Exceptions occurred in finalize() method are not propagated. They are ignored by the garbage collector.

***6)*** You can call finalize() method explicitly on an object before it is abandoned. When you call, only operations kept in finalize() method are performed on an object. Object will not be destroyed from the memory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | class A  {      int i;        public A(int i)      {          this.i = i;      }        @Override      protected void finalize() throws Throwable      {          System.out.println("From Finalize Method, i = "+i);            //Calling super class finalize() method explicitly            super.finalize();      }  }    public class Test  {     public static void main(String[] args)     {         //Creating two instances of class A           A a1 = new A(10);           A a2 = new A(20);           //Calling finalize() method of a1 before it is abandoned         try         {             a1.finalize();         }         catch (Throwable e)         {             e.printStackTrace();         }           //Assigning a2 to a1           a1 = a2;           //Now both a1 and a2 will be pointing same object           //An object earlier referred by a1 will become abandoned           System.out.println("done");     }  } |

***7)*** finalize() method on an abandoned object is called only once by the garbage collector thread. GC ignores finalize() method called on an object by the developer.