[**How clone method works in Java?**](http://javarevisited.blogspot.in/2013/09/how-clone-method-works-in-java.html)

The clone() is a tricky method from java.lang.Object class, which is used to create a copy of an Object in Java.

What is the clone of an object in Java?

An object which is returned by the clone() method is known as a clone of original instance. A clone object should follow basic characteristics e.g. a.clone() != a, which means original and clone are two separate object in Java heap, a.clone().getClass() == a.getClass() and clone.equals(a), which means clone is exact copy of original object. This characteristic is followed by a well behaved, correctly overridden clone() method in Java, but it's not enforced by the cloning mechanism. Which means, an object returned by clone() method may violate any of these rules.  
  
By following the convention of returning an object by calling super.clone(), when overriding clone() method, you can ensure that it follows first two characteristics. In order to follow the third characteristic, you must override equals method to enforce logical comparison, instead of physical comparison exists in java.lang.Object.  
  
For example, clone() method of Rectangle class in this method return object, which has these characteristics, but if you run the same program by commenting equals(), you will see that third invariant i.e. **clone.equals(a)** will return **false**.

How Clone method works in Java

java.lang.Object provides default implementation of clone() method in Java. It's declared as [protected](http://javarevisited.blogspot.sg/2012/10/difference-between-private-protected-public-package-access-java.html) and native in Object class, so implemented in native code. Since its convention to return clone() of an object by calling super.clone() method, any cloning process eventually reaches to java.lang.Object clone() method. This method, first checks if the corresponding object implements Cloneable interface, which is a marker interface. If that instance doesn't implement Cloneable then it throws CloneNotSupportedException in Java, a checked exception, which is always required to be handled while cloning an object. If an object passes this check, than java.lang.Object's clone() method creates a shallow copy of the object and returned it to the caller.  
  
  
Since Object class' clone() method creates copy by creating new instance, and then copying field-by-field, similar to assignment operator, it's fine for primitives and Immutable object, but not suited if your class contains some mutable data structure e.g. Collection classes like ArrayList or [arrays](http://javarevisited.blogspot.sg/2011/06/converting-array-to-arraylist-in-java.html). In that case, both original object and copy of the object will point to the same object in the heap. You can prevent this by using the technique known as deep cloning, on which each mutable field is cloned separately. In short, here is how clone method works in Java:

1) Any class calls clone() method on an instance, which implements Cloneable and overrides protected clone() method from Object class, to create a copy.

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

  logger.info(rec);

**try** {

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

         Rectangle copy = rec.clone();

         logger.info("Copy " + copy);

    } **catch** (CloneNotSupportedException ex) {

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

    }

2) Call to clone() method on Rectangle is delegated to super.clone(), which can be a custom [superclass](http://java67.blogspot.sg/2013/06/difference-between-this-and-super-keyword-java.html) or by default java.lang.Object

    @Override

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

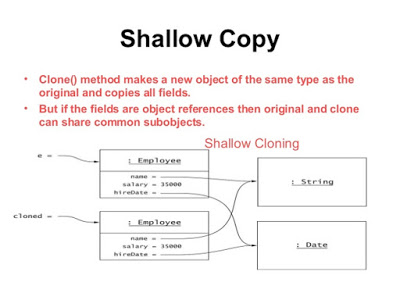
**return** (Rectangle) **super**.clone();

    }

3) Eventually, call reaches to java.lang.Object's clone() method, which verify if the corresponding instance implements Cloneable interface, if not then it throws CloneNotSupportedException, otherwise it creates a field-by-field copy of the instance of that class and returned to the caller.

So in order for clone() method to work properly, two things need to happen, a class should implement Cloneable interface and should*override clone() method of Object class*.  
  
By the way this was this was the simplest example of overriding clone method and how it works, things gets more complicated with real object, which contains mutable fields, arrays, collections, [Immutable object](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html), and primitives, which we will see in [second part](http://javarevisited.blogspot.sg/2015/01/java-clone-tutorial-part-2-overriding-with-mutable-field-example.html) of this **Java Cloning tutorial** series.

Here is how  a shallow copy of an object looks like:

[](http://1.bp.blogspot.com/-rojtWvPpiiQ/VgvbR1CshxI/AAAAAAAAD2I/rgHImvatmls/s1600/Shallo+Copy+in+Java+using+clone.jpg)

Java clone() method Example

In this article, we have not seen complexity of overriding clone method in Java, as our Rectangle class is very simple and only contains primitive fields, which means shallow cloning provided by Object's **clone()** method is enough. But, this example is important to understand the process of Object cloning in Java, and *how clone method works*. Here is complete code of this clone() method overriding example:

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

/\*\*

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

  \* Object works in Java.

  \*

  \* @author

 \*/

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

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

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

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

        logger.info(rec);

        Rectangle copy = **null**;

**try** {

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

            copy = rec.clone();

            logger.info("Copy " + copy);

        } **catch** (CloneNotSupportedException ex) {

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

        }

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

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

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

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

        //Updating fields in original object

        rec.setHeight(**100**);

        rec.setWidth(**45**);

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

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

    }

}

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

**private** **int** width;

**private** **int** height;

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

        width = w;

        height = h;

    }

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

**this**.height = height;

    }

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

**this**.width = width;

    }

**public** **int** **area**(){

**return** widthheight;

    }

    @Override

**public** String **toString**(){

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

    }

    @Override

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

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

    }

    @Override

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

**if** (obj == **null**) {

**return** **false**;

        }

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

**return** **false**;

        }

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

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

**return** **false**;

        }

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

**return** **false**;

        }

**return** **true**;

    }

    @Override

**public** **int** **hashCode**() {

**int** hash = **7**;

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

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

**return** hash;

    }

}

*Output:*

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

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

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

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

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

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

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

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

From the output, you can clearly see that cloned object has the same attribute as the original object in Java. Also changing the attribute of an original object is not affecting the state of copy object because they only contain primitive fields. If they had contained any mutable object, it would have affected both of them.  
  
You can also see that it follow standard properties of cloned object i.e.

* clone != original,
* clone.getClass() == original.getClass(), and
* clone.equals(original).

**Things to Remember - Clone method in Java**

1) The clone() method is used to create a copy of an object in Java. In order to use clone() method, class must implement java.lang.Cloneable [interface](http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html) and override protected clone() method from java.lang.Object.  
  
A call to clone() method will result in CloneNotSupportedException if that class doesn't implement Cloneable interface.

2) No constructor is called during cloning of Object in Java.

3) Default implementation of clone() method in Java provides "*shallow copy"* of object, because it creates copy of Object by creating new instance and then copying content by assignment, which means if your class contains a mutable field, then both original object and clone will refer to same internal object. This can be dangerous because any change made on that mutable field will reflect in both original and copy object. In order to avoid this, override clone() method to provide the [deep copy of an object](http://javarevisited.blogspot.sg/2014/03/how-to-clone-collection-in-java-deep-copy-vs-shallow.html).

4) By convention, clone of an instance should be obtained by calling super.clone() method, this will help to preserve invariant of object created by clone() method i.e. **clone != original** and **clone.getClass() == original.getClass()**. Though these are not absolute requirement as mentioned in Javadoc.

5) A shallow copy of an instance is fine, until it only contains primitives and Immutable objects, otherwise, you need to modify one or more mutable fields of object returned by super.clone(), before returning it to caller.

That's all on **How clone method works in Java**. Now we know, what is the clone and what is Cloneable interface, a couple of things about clone method and what does default implementation of clone method do in Java. This information is enough to move ahead and read [second part](http://javarevisited.blogspot.sg/2015/01/java-clone-tutorial-part-2-overriding-with-mutable-field-example.html) of this Java cloning tutorial, on which we will learn, *how to override clone() method in Java*, for classes composed with primitives, mutable and immutable objects in Java.

Deep Cloning

-----------------

Your object you wish to deep copy will need to implement serializable. If the class isn't final or can't be modified, extend the class and implement serializable.

Convert your class to a stream of bytes:

ByteArrayOutputStream bos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(bos);

oos.writeObject(object);

oos.flush();

oos.close();

bos.close();

byte[] byteData = bos.toByteArray();

Restore your class from a stream of bytes:

ByteArrayInputStream bais = new ByteArrayInputStream(byteData);

(Object) object = (Object) new ObjectInputStream(bais).readObject();

#Why clone is protected:

clone is protected because it is something that ought to be overridden so that it is specific to the current class. While it would be possible to create a public clone method that would clone any object at all, this would not be as good as a method written specifically for the class that needs it.