How does clone() method works in Java?

Exploring Java's clone() method to answer difference between deep cloning and shallow cloning

JAVINPAUL
APR 18, 2024 · PAID

3 1 Share ···

Hello folks, the `clone()` is a tricky method from `java.lang.Object` class, which is used to create a copy of an object in Java. The intention of the `clone()` method is simple, to provide a cloning mechanism, but somehow its implementation became tricky and has been widely criticized for a long time.

Anyway, we will not go to the classic debate of clones in Java, at least for now; instead, we will try to learn *how the clone method works in Java*.

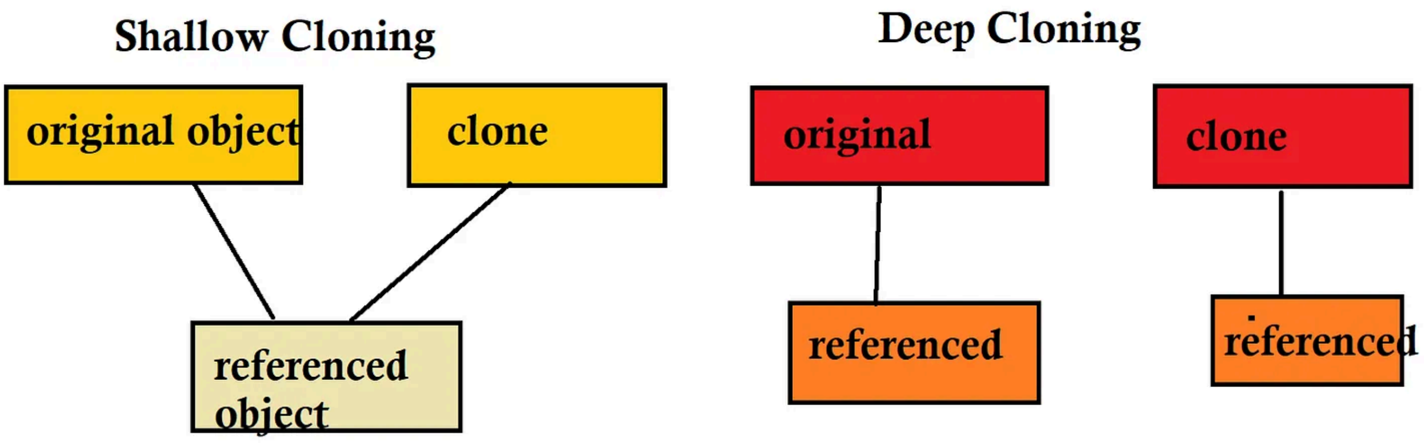
To be fair, understating the cloning mechanism in Java is not easy and even experienced Java programmers fail to **explain how cloning of mutable objects works**, or the difference between deep and shallow copy in Java.

In this three-part article, we will first see the working of the clone method in Java, and in the second part we will learn **how to override the clone method in Java**, and finally, we will discuss the *deep copy vs shallow copy* mechanism.

The reason I chose to make this a three-part article is to keep the focus on one thing at a time.

Since **clone() itself is confusing enough**, it's best to understand concepts one by one. In this post, we will learn what is clone method is, what it does, and How the clone method works in Java.

By the way, `clone()` is one of the few fundamental methods defined by objects, others being equals, `hashcode(), toString()` along with wait and notify methods.



What is the clone of an object in Java?

An object which is returned by the clone() method is known as a clone of the 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.

This means an object returned by the clone() method may violate any of these rules.

By following the convention of returning an object by calling super.clone(), when overriding the `clone()` method, you can ensure that it follows the first two characteristics.

In order to follow the third characteristic, you must override the 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 does clone() method works in Java

`java.lang.Object` provides default implementation of `clone()` method in Java. It's declared as protected and native in the 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, then `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.

In that case, both the 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 the 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 or by default `java.lang.Object`

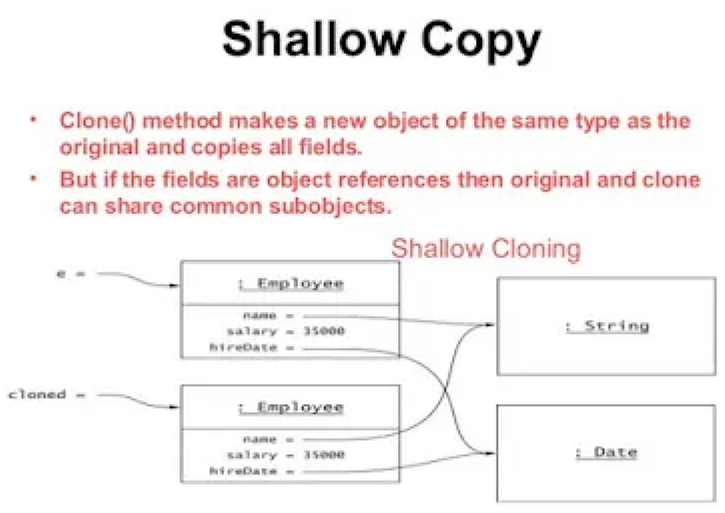
```
    @Override

    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.

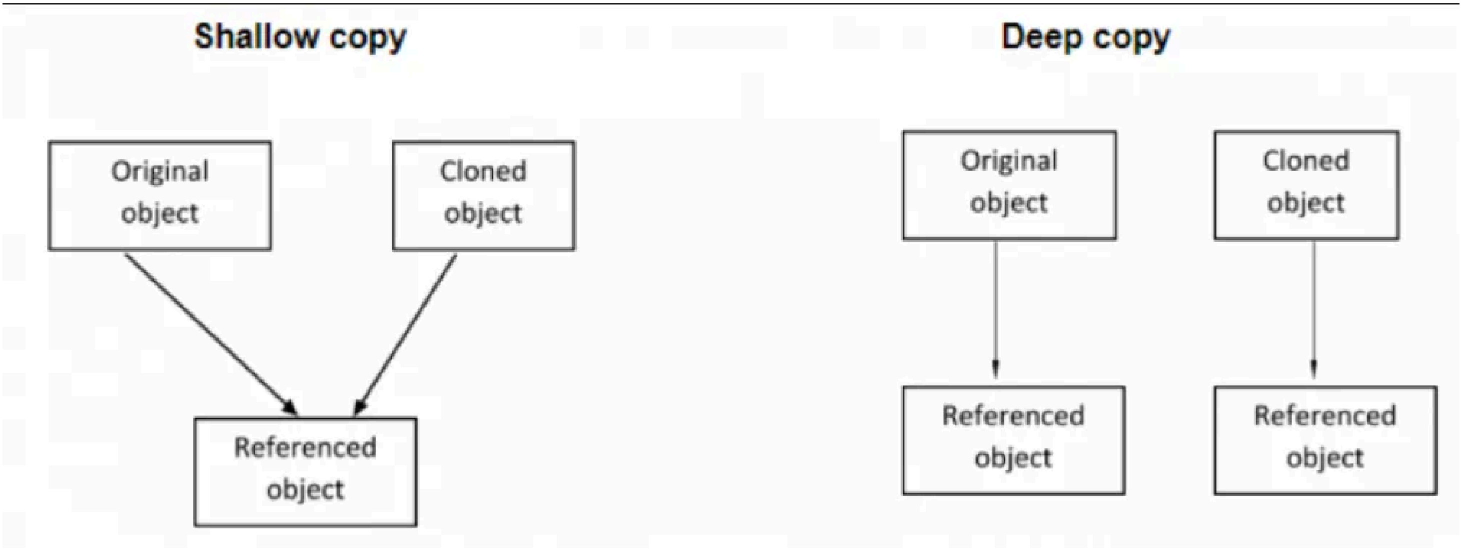
So in order for `clone()` method to work properly, two things need to happen, a class should implement Cloneable interface and should *override clone() method of Object class.*

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, and primitives, which we will see in second part of this **Java Cloning tutorial** series.

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


Shallow Copy
• Clone() method makes a new object of the same type as the original and copies all fields.
• But if the fields are object references then original and clone can share common subobjects.
Shallow Cloning
e =
: Employee
name =
salary = 35000
hireDate =
cloned =
: Employee
name =
salary = 35000
hireDate =
: String
: Date

And, here is a diagram which shows the difference between shallow and dep cloning:



You can see that in case of shallow copy, both original and clone object refer to same member object but in case of deep copy, both original and cloned object have separate copy of any referenced member object. Hence its called deep copy or deep cloning.

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 javinpaul

 */

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:

```
2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - Rectangle [width: 30, height: 60, area: 1800]

2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - Creating Copy of this object using Clone method

2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - Copy Rectangle [width: 30, height: 60, area: 1800]

2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - copy != rec : true

2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - copy.getClass() == rec.getClass() : true

2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - copy.equals(rec) : true

2024-04-17 23:46:58,882 0    [main] INFO  JavaCloneTest  - Original object :Rectangle [width: 45, height: 100, area: 4500]

2024-04-17 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 about Object.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 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 the object because it creates a copy of Object by creating a 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 the same internal object.

This can be dangerous because any change made on that mutable field will reflect in both the original and copy object. **In order to avoid this, override the clone() method to provide the deep copy of an object.**

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 the object returned by `super.clone()`, before returning it to the caller.

That's all on **How the clone method works in Java**. Now we know, what is the clone and what is Cloneable interface is, a couple of things about the clone method, and what does default implementation of the clone method does in Java.

This information is enough to move ahead and read the second part of this Java cloning tutorial, on which we will learn, *how to override the* `clone()` *method in Java*, for classes composed of primitives, mutable and immutable objects in Java.

Thank you all and all the best for your interviews

Please let me know how do you find this article ..

3 Likes · 1 Restack

Previous

Next

Comments

Write a comment...