How to Clone Collection in Java - Deep copy of ArrayList and HashSet

Programmer often mistook copy constructors provided by various collection classes, as a mean to

clone Collection e.g. `List`, `Set`, `ArrayList`,`HashSet` or any other implementation. What is worth remembering is that, copy constructor ofCollection in Java only provides shallow copy and not deep copy, which means objects stored in both original List and cloned List will be same and point to same memory location in Java heap. One thing, which adds into this misconception is shallow copy of Collections with Immutable Objects. Since Immutable can't be changed, It's Ok even if two collections are pointing to same object. This is exactly the case of String contained in pool, update on one will not affect other. Problem arise, when we use Copy constructor of `ArrayList` to create a clone of List of Employees, where `Employee` is not Immutable. In this case, if originalcollection modifies an employee, that change will also reflect into cloned collection. Similarly if an employee is modified in cloned collection, it will also appeared as modified in original collection. This is not desirable, in almost all cases, clone should be independent of original object. Solution to avoid this problem is **deep cloning ofcollection**, which means recursively cloning object, until you reached to primitive or Immutable. In this article, we will take a look at one approach of deep copying Collection classes e.g. `ArrayList` or `HashSet` in Java. By the way, If you know difference between shallow copy and deep copy, it would be very easy to understand how deep cloning ofcollection works.

Deep Cloning of Collection in Java

In following example, we have a Collection of `Employee`, a mutable object, with `name`and `designation` field. They are stored inside `HashSet`. We create another copy of thiscollection using `addAll()` method

of `java.util.Collection` interface. After that, we modified designation of each `Employee` object stored in original Collection. Ideally this change should not affect original Collection, because clone and original object should be independent of each other, but it does. Solution to fix this problem is deep cloning of elements stored in `Collection` class.

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
  * Java program to demonstrate copy constructor of Collection provides shallow
  * copy and techniques to deep clone Collection by iterating over them.
  * @author http://javarevisited.blogspot.com
  */
public class CollectionCloningTest {
    private static final Logger logger = LoggerFactory.getLogger(CollectionCloningclass);


    public static void main(String args[]) {

        // deep cloning Collection in Java
        Collection<Employee> org = new HashSet<>();
        org.add(new Employee("Joe", "Manager"));
        org.add(new Employee("Tim", "Developer"));
        org.add(new Employee("Frank", "Developer"));

        // creating copy of Collection using copy constructor
        Collection<Employee> copy = new HashSet<>(org);

        logger.debug("Original Collection {}", org);
        logger.debug("Copy of Collection  {}", copy );

        Iterator<Employee> itr = org.iterator();
        while(itr.hasNext()){
            itr.next().setDesignation("staff");
        }

        logger.debug("Original Collection after modification  {}", org);
        logger.debug("Copy of Collection without modification {}", copy );

        // deep Cloning List in Java

    }

}

class Employee {
    private String name;
    private String designation;
```

```
    public Employee(String name, String designation) {
        this.name = name;
        this.designation = designation;
    }

    public String getDesignation() {
        return designation;
    }

    public void setDesignation(String designation) {
        this.designation = designation;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("%s: %s", name, designation );
    }

}

Output :
- Original Collection [Joe: Manager, Frank: Developer, Tim: Developer]
- Copy of Collection  [Joe: Manager, Frank: Developer, Tim: Developer]
- Original Collection after modification  [Joe: staff, Frank: staff, Tim: staff]
- Copy of Collection without modification [Joe: staff, Frank: staff, Tim: staff]
```

You can see it clearly that modifying `Employee` object in original `Collection` (changed designation to `"staff"`) is also reflecting in cloned collection, because clone is shallow copy and pointing to same `Employee` object in heap. In order to fix this, we need to deep clone `Employee` object by iterating over Collection and before that, we need to override clone method for `Employee` object.

1) Let Employee implements `Cloneable` interface
2) Add following `clone()` method into `Employee` class

```
@Override
    protected Employee clone() {
        Employee clone = null;
        try{
            clone = (Employee) super.clone();
```

```
        }catch(CloneNotSupportedException e){
            throw new RuntimeException(e); // won't happen
        }

        return clone;

    }
```

3) Instead of using Copy constructor use following code, to deep copy Collection in Java

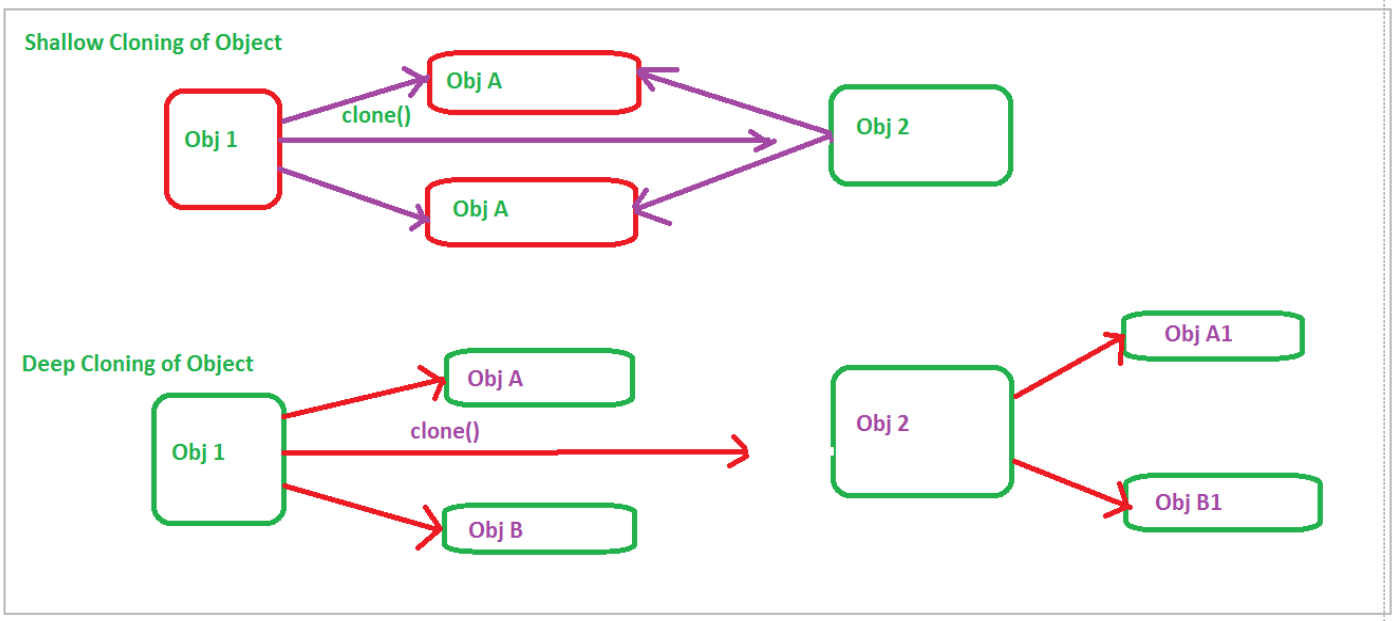
```
Collection<Employee> copy = new HashSet<Employee>(org.size());

Iterator<Employee> iterator = org.iterator();
while(iterator.hasNext()){
    copy.add(iterator.next().clone());
}
```

4) Running same code for modifying collection, will not result in different output:

- Original Collection after modification `[Joe: staff, Tim: staff, Frank: staff]`
- Copy of Collection without modification `[Frank: Developer, Joe: Manager, Tim: Developer]`

You can see that both clone and Collection are independent to each other and they are pointing to different objects.



That's all on **How to clone Collection in Java**. Now we know that copy constructor or various collection classes e.g. `addAll()` method of List or Set, only creates *shallow copy of Collection*, and both original and cloned Collection points to same objects. To avoid this, we can deep copy collection, my iterating over them and cloning each element. Though this requires that any object stored in Collection, must support deep cloning operation.