Clone Method and Clonable Interface

Harshit Maheshwari
Indian Institute of Technology, Kanpur
Computer Science and Engineering
harshitm@iitk.ac.in

1 Introduction

When we want classes to support copying functionality we want to ‘clone‘ the objects of the class. In Java all classes are extended from ‘Object‘ class. ‘Object‘ class implements cloneable interface. The ‘cloneable‘ interface of java is an empty interface which contains no members at all. It is used by a class to indicate that the class supports cloning. By ‘supports‘ it means that the class implements clone() method. However, the clone() method of the ‘Object‘ class does shallow copying. If we want to do deep copying we need to do that in the child class on the copy inherited from ‘Object‘ class clone method. Some of the basic properties a clone method should satisfy are:

- The cloned object does not have the same reference as that of original object .ie, o.clone() != o should be true
- The cloned object has the same values as that of the original object for all the fields of the class. ie, o.clone.equals(o) should be true.
- The cloned object and the original object should belong to the same class. ie, o.clone.getClass() == o.getClass() should be true.

2 Modifier

To define clone the clone method of ‘Object‘ class has to be overridden. The ‘clone‘ method implemented by ‘Object‘ class is *protected*.The child class can extend ‘clone‘ method using *public/protected* modifier. We cannot use *private/package protected* because it reduces the visibility of inherited method and gives compilation error.

```java
public class Animal implements Cloneable{
  Animal clone(){
    Animal a = new Animal();
    return a;
  }
}
```

Listing 1: Private/Package private gives error

The above code gives the following exception:
Cannot reduce the visibility of the inherited method from Object
at Animal.clone(Animal.java:6)

3 ‘clone‘ method is necessary

It is necessary to implement the clone method along with ‘cloneable‘ interface.

```java
public class Animal implements Cloneable{

}

public static void main(){String args[]){
  Animal a = new Animal();
  Animal aClone = a.clone();
}
```

Listing 2: ‘Clone method not implemented in child class‘

The code ‘Clone method not implemented in child class‘ gives the following exception
Unresolved Compilation Problem
The method clone() from the type Object is not visible

4 ‘CloneNotSupportedException‘ [3, 4]

When creating objects with ‘clone‘ method we need to extend the cloneable interface. Otherwise the compiler gives ‘CloneNotSupportedException‘. Note, that ‘CloneNotSupported‘ is a checked exception and therefore, we use keyword ‘throws‘ in clone method when we use the clone method of the parent class.

```java
public class Animal {
  public Animal clone(){
    return (Animal)super.clone();
  }
}
```

Listing 3: CloneNotSupportedException Example

Above code gives the following exception:
Unhandled exception type CloneNotSupportedException
at Animal.clone(Animal.java:5)

5 Shallow Cloning vs Deep Cloning [1, 2]

The ‘object‘ class clone method by default returns shallow copy of the object. In order to create deep copy we need to override the ‘clone‘ method of the parent class.

5.1 Shallow Cloning

As mentioned in Section 1 ‘clone‘ method of ‘Object‘ class returns a shallow copy.

```java
public class Dog implements Cloneable{
  public StringBuffer b;        //User defined class
  public Dog(StringBuffer s){
    this.b= s;
  }
  public boolean equals (Dog d){    //Overrides default equals
      method. Compares references of user defined types
    if(this.b == d.b){
      return true;
    }
    return false;
  }
  public Dog clone() throws CloneNotSupportedException{
    return (Dog) super.clone();
  }
}


public class Cat implements Cloneable{
  public int i;         //Primitive Data type
  public Cat(int i){
    this.i = i;
  }
  public boolean equals (Cat d){
    if(this.i == d.i){
      return true;
    }
    return false;
  }
  public Cat clone() throws CloneNotSupportedException{
    return (Cat) super.clone();
  }
}

public class Timepass  {
  public static void main(String[] args) throws
      CloneNotSupportedException {
    Dog d = new Dog(new StringBuffer("Tom"));
    Cat c = new Cat(1);
    Dog dClone = d.clone();
    Cat cClone = c.clone();

    c.i = 2;
    dClone.b.append("my");

    if(c == cClone){
      System.out.println("Cat references are same");
    }
    else{
      System.out.println("Cat references are different");
    }
    if(c.equals(cClone)){
      System.out.println("Cat members are same");
    }
    else{
      System.out.println("Cat members are different");
    }
    if(d == dClone){
      System.out.println("Dog references are same");
    }
    else{
      System.out.println("Dog references are different");
    }
    if(d.equals(dClone)){
      System.out.println("Dog members are same");
    }
    else{
      System.out.println("Dog members are different");
    }
    System.out.println("c.i= " + c.i + " vs " + "cClone.i = " +
       cClone.i);
    System.out.println(("d.i= " + d.b + " vs " + "dClone.i = " +
       dClone.b));
  }
}
```

Listing 4: Shallow Cloning

```
Cat references are different
Cat members are different
Dog references are different
Dog members are same
c.i= 2 vs cClone.i = 1
d.i= Tommy vs dClone.i = Tommy
```

Listing 5: Shallow Cloning Output

As we can see from code Shallow Cloning and the output Shallow Cloning Output if the members of child class consists of

- **Primitive/Immutable data types**
  New copies are created. The value of primitive type 'i' is copied in 'cClone'.
- **User Defined classes**
  Only references are copied. Only the reference of field 'StringBuffer b' is copied in clone 'dClone'.

The desired behaviour was to have different values in *d.b* and *dClone.b.*

5.2 Deep Cloning

For deep cloning, we must satisfy following:

- All the members of the class should implement Cloneable interface and should override Object's clone() method.
- If a member method does not follow the above rule, we must create a new instance of the member class and copy all the attributes to the new object(ensuring deep copying).

In order to implement the desired behaviour we need to modify the clone method of the child class as shown in Code 'Deep Cloning'

```java
public Dog clone() throws CloneNotSupportedException{
    Dog cloned = (Dog) super.clone();
    StringBuffer bClone = new StringBuffer(this.b);
    cloned.b = bClone;
    return cloned;
}
```

Listing 6: 'Deep Cloning'

```
Cat references are different
Cat members are different
Dog references are different
Dog members are different
c.i= 2 vs cClone.i = 1
d.i= Tom vs dClone.i = Tommy
```

Listing 7: 'Deep Cloning Output'

6 Inheritance

6.1 Inheritance from 'Object' Class

If we use 'clone' method of Object class then then the clone method should throw 'ClassCastException' exception.

6.2 Inheritance from parent class

If we use 'clone' method of parent class and if parent class returns a new object then we get *ClassCastException*

```java
public class Animal implements Cloneable{
  public Animal clone() throws CloneNotSupportedException{
    return new Animal();
  }
}

public class Dog extends Animal implements Cloneable{
  public Dog clone(){
    return (Dog) super.clone();
  }
}
```
Listing 8: ‘ClassCastException Example‘

```
Exception in thread "main" java.lang.ClassCastException: Animal
    cannot be cast to Dog
  at Dog.clone(Dog.java:30)
  at Timepass.main(Timepass.java:42)
```
Listing 9: ‘ClassCastException‘

So, if a class inherits a base class, then the base class must make a call to super.clone() in order to invoke Object.clone() method. The output of code ‘TypeCastingError‘ is ‘TypeCastingError output‘

```java
public static void main(String[] args) throws
    CloneNotSupportedException {
  Dog d = new Dog();
  Animal a = d;
  Dog dClone = (Dog) a.clone();
}
```
Listing 10: ‘TypeCastingError‘

```
Exception in thread "main" java.lang.ClassCastException: Animal
    cannot be cast to Dog
  at Dog.clone(Dog.java:17)
  at Dog.clone(Dog.java:1)
  at Timepass.main(Timepass.java:43)
```
Listing 11: ‘TypeCastingError output‘

We get error because after dynamic type inferencing the clone() method of Dog() is called and which in turn calls clone() method of Animal class which returns a new object of Animal class which cannot be typecasted to Dog.

6.3 TYPECASTING IN CLONING

Consider the code ‘TypeCasting in Cloning‘

```java
public static void main(String[] args) throws
    CloneNotSupportedException {
    Animal a = new Dog(new StringBuffer("Tom"));//Upcasting from Dog
         to Animal
    Animal aClone =  a.clone();
    Dog d = (Dog) a;              //DownCasting from Animal to Dog
    Dog dClone = (Dog)aClone;          //DownCasting from Animal to
        Dog
}
```
Listing 12: ‘TypeCasting in Cloning‘

In this we see we can convert from cloned ‘Dog‘ to ‘Animal‘ and then back to ‘Animal‘.

6.4 Cloneable Interface in parent class

Consider the code ‘Parent class need not implement Cloneable interface‘

```
public class Animal {
}
public class Dog extends Animal implements Cloneable{
  public Dog clone() throws CloneNotSupportedException{
    return (Dog)super.clone();
  }
}
public class Main{
  public static void main(String args[]){
    Dog d  = new Dog();
    Dog dClone = d.clone();
  }
}
```

Listing 13: ‘Parent class need not implement Cloneable interface‘

The code runs without any errors.

7 Acknowledgements

Prabhat Pandey (10498) and I, thoroughly discussed the assignment before we started making the report.

References

[1] Java clone, shallow copy and deep copy. http://www.javaexperience.com/java-cloning-tutorial/, 2009.

[2] Joe. Java clone, shallow copy and deep copy. http://javapapers.com/core-java/java-clone-shallow-copy-and-deep-copy/, 2009.

[3] Oracle. Interface cloneable. http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html.

[4] Suresh Sajja. Cloneable interface in java. http://coderevisited.com/cloneable-interface-in-java/, 2012.