Patrick Ian E. Cura
2154687

Programming 3 – Assignment 1 – Task 3 – Cloning

Cloning is used in Java if you want to create a copy of an object that you want to modify without affecting the original one. Performing actions on a cloned object should not change the object passed by the caller.

One of the approaches to doing cloning in Java is by using the clone() function. This method is found in the base class Object. However, its access modifier is protected and must be overridden with a child class's public clone method to be usable. An ArrayList is an example of a Java class that overrides the base class's protected clone method. Due to this, the clone method can be called from ArrayLists. It should be noted though that the return type of the clone method is an Object and so the returned object must be casted to the intended type: `ArrayList v = new ArrayList(); ArrayList v2 = (ArrayList) v.clone();`

However, not all Java classes override the clone method of the base class. For example, when the clone() method is called from an Integer type object, an error will be encountered.

Besides overriding the Base class's protected clone method, super.clone() must also be called inside a class's public clone method. Since all classes inherit from the Base class, the protected clone method is accessible inside a class and needs to be called to do the actual cloning. The Base's clone determines the size of an object, reserves the memory needed, and then copies the details from the original to the clone.

Lastly, to do a successful basic clone, the Cloneable interface must be implemented by a class. When the contents of the Cloneable interface is checked though, it is empty and does not contain any methods that are required to be implemented unlike regular interfaces. In a sense, it is just implemented to mark an object as being Cloneable when for example the instance of keyword is used on it. It is useful in letting programmers know if the object they are working on is possible to clone. Also, an object that does not implement the Cloneable interface will return a CloneNotSupportedException when super.clone() is called. The Base class's clone first checks if the class calling it is Cloneable before doing the cloning. A simple implementation of cloning can be seen below:

```java
class ObjectToClone implements Cloneable
{
	public Object clone()
	{
		Object o = null;
		try{
			o = super.clone();}
		catch (CloneNotSupportedException e)
		{
			System.err.println("ObjectToClone cannot perform clone");
		}return o;}}
```

However, just calling super.clone() is usually not enough if a class has Objects which contain Objects. In this case, a shallow copy is only generated and modifying the Objects inside a clone's Object fields will still modify the original's own objects. This can be seen in the following code example. Modifying obj2's String Array still affects obj1's.

```java
class ObjectToClone implements Cloneable
{
	public String[] value = {"old"};
	public Object clone()
	{
		Object o = null;
		try{
			o = super.clone();}
		catch (CloneNotSupportedException e)
		{
			System.err.println("ObjectToClone cannot perform clone");
		}return o;}}
```

```
public class ShallowCopyExample
{
        public static void main(String[] args)
        {
                ObjectToClone obj1 = new ObjectToClone();
                ObjectToClone obj2 = (ObjectToClone)obj1.clone();
                obj2.value[0] = "new";
                System.out.println("obj1.value = " + obj1.value[0] + "\nobj2.value = " + obj2.value[0]);}}
```

To perform a successful clone for the example above, a deep copy must be done. Besides calling super.clone(), the objects inside the class must also be cloned so that their references will be decoupled from the original. Implementing a deep copy on the code results in the following:

```
class ObjectToClone implements Cloneable
{
        public String[] value = {"old"};
        public Object clone(ObjectToClone o)
        {
                o = null;
                try{
                        o = (ObjectToClone)super.clone();
                }
                catch (CloneNotSupportedException e)
                {
                        System.err.println("ObjectToClone cannot perform clone");
                }
                o.value = o.value.clone(); return o; }}

public class DeepCopyExample
{
        public static void main(String[] args)
        {
                ObjectToClone obj1 = new ObjectToClone();
                ObjectToClone obj2 = (ObjectToClone)obj1.clone(obj1);
                obj2.value[0] = "new";
                System.out.println("obj1.value = " + obj1.value[0] + "\nobj2.value = " + obj2.value[0]); }}
```

As would be seen in the next code samples, cloning can still be used by classes that extend a Cloneable class. Cloning can however be prevented in a child class. This is done for example for security reasons. It can be still be re-enabled though by implementing a new method that will simulate the cloning process. Lastly, Cloning can be stopped permanently by declaring a class final so that extending it and overriding the clone method will not be possible anymore.

```
// This class will clone correctly
class CloneableClassSample implements Cloneable
{
        public Object clone() throws CloneNotSupportedException
        {
                return super.clone(); }}

// This child class of a cloneable class can still do a clone
class StillCloneable extends CloneableClassSample {}

// This class won't be able to clone and will throw an exception when clone is called
class NotCloneableAnymore extends CloneableClassSample
{
        public Object clone() throws CloneNotSupportedException
        {
                throw new CloneNotSupportedException(); }}

// This class still won't be able to clone and will throw an exception as it calls NotCloneableAnymore's clone
class StillNotCloneableAnymore extends NotCloneableAnymore
{
        public Object clone() throws CloneNotSupportedException
        {
                return super.clone(); }}
```

```
class TurnOnCloningViaADifferentMethod extends NotCloneableAnymore
{
        private TurnOnCloningViaADifferentMethod cloneAttempt(TurnOnCloningViaADifferentMethod t)
        {
                // Do something here that will do a clone
                return new TurnOnCloningViaADifferentMethod(); }

        public Object clone()
        {
                // Re-enable cloning by implementing a new way of cloning
                return cloneAttempt(this); }}

// Cloning is disabled completely as this class cannot be extended to implement a new clone method
// unlike TurnOnCloningViaADifferentMethod
final class DisableCloneCompletely extends NotCloneableAnymore {}
```

As could be seen from the previous code examples, using the Base class's clone method is quite an arduous task. To successfully do a deep clone, all object references must be accounted for. The clone's objects also has to be checked if it supports cloning and if it does not, some workarounds may need to be done. These factors clutter the class with all the checks and additional implementations needed to do a clone. However, using the Base class's clone method allows control and customization of the cloning process. Furthermore, it takes less processing time compared to some alternatives that will discussed next.

An alternative to doing the deep clone demonstrated above is by using serialization. This could be seen in the following code samples. It should be noted that using the serialization implementation takes more computer processing time as more steps are done by the machine to do the clone. However, from a programmers point of view, it is easier to setup.

```
import java.io.*;

class SerializableClass1 implements Serializable {}

class SerializableClass2 implements Serializable
{
        SerializableClass1 object1 = new SerializableClass1();}

public class SerializationCloningExample
{
        public static void main(String[] args) throws Exception
        {
                // create the object to be cloned using serialization
                SerializableClass2[] original = new SerializableClass2[30000];
                for (int i = 0; i < original.length; i++)
                        original[i] = new SerializableClass2();

                // setup the streams to store the original object
                ByteArrayOutputStream buffer = new ByteArrayOutputStream();
                ObjectOutputStream output = new ObjectOutputStream(buffer);

                // write the original object data to the stream
                for (int i = 0; i < original.length; i++)
                        output.writeObject(original[i]);

                ObjectInputStream input = new ObjectInputStream(new ByteArrayInputStream(buffer.toByteArray()));

                // copy the details from the stream to the clone object
                SerializableClass2[] clone = new SerializableClass2[30000];
                for (int i = 0; i < clone.length; i++)
                        clone[i] = (SerializableClass2) input.readObject(); }}
```

References: (Code Samples and Ideas based on the following)

Eckel, B. Thinking in Enterprise Java, 3rd Edition (http://www.mindviewinc.com/Books/)