I believe that the Java cloning mechanism is necessary because it provides a quick and effective way to make exact copies of objects. This is useful as it saves time in implementing new() with all of the objects fields when only some of the fields need to be changed. While it does require the object to implement the Cloneable interface. It is very useful for cloning simple classes with only primitive fields (shallow copying) and with some potentially tricky tweaking with deep copying. The java cloning mechanism is also faster then it’s alternatives during run time.

Java cloning mechanism. Java cloning involves using the clone() method to create a copy of a java object. For objects to be cloned. The class the object must implement the Cloneable interface and override the protected Object Clone Method. This will create a copy of an object instead of a reference. (Gupta, 2013)

**public** **class** daClone **implements** Cloneable

{

**private** String yoyo = "A yoyo";

**private** **int** over69Thousand = 69001;

**private** String somethngElse;

**private** **int** aNumber;

///

**public** daClone(String somethngElse, **int** aNumber) {

//super();

**this**.somethngElse = somethngElse;

**this**.aNumber = aNumber;

}

///

@Override

**public** Object clone() **throws** CloneNotSupportedException

{

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

}

}

The simplest way to make a java class cloneable (The getters and setters omitted to save space on paper but they’re in the source code)

daClone originalz = **new** daClone("the original", 6972);

daClone aReference = originalz;

daClone copyNotReference = **null**;

**try**

{

copyNotReference = (daClone) originalz.clone();

}

**catch** (CloneNotSupportedException e)

{

e.printStackTrace();

}

String originslVReference = (originalz == aReference)? " reference to originalz" : " copy of originalz";

String originsVClone = (originalz == copyNotReference)? " reference to originalz" : " copy of originalz";

System.***out***.println("aRefefrence is a" + originslVReference + "\n"+

"And copyNotReference is a" + originsVClone);

System.***out***.println((originalz.getSomethngElse().equals(copyNotReference.getSomethngElse()) ) ? "Copied but identical fields " + copyNotReference.getSomethngElse() : "Nope it's different. It's " + copyNotReference.getSomethngElse());

The simplest implementation of the java cloning mechanism.

aRefefrence is a reference to originalz

And copyNotReference is a copy of originalz

Copied but identical fields the original

The output proving that the cloning mechanism copies an object and not create a reference to it.

This is referred to as a shallow copy. Shallow copy clones copy the field values of primitive types but references the values of Objects. In order to copy all of the values. A Deep copy clone needs to make.

@Override

**public** Object clone() **throws** CloneNotSupportedException

{

DaClone clonedObjectField = (DaClone)**super**.clone();

clonedObjectField.setObjectField((Class2Prove) clonedObjectField.getObjectField().clone());

**return** clonedObjectField;

}

A clone method set up for deep copying.

deepClone.getObjectField().setFoosRoDah("NO!!\nTHIS IS PATRICK!!");

System.***out***.println("The orignal: " + original.getObjectField().getFoosRoDah() + "\n" +

"The deep clone " + deepClone.getObjectField().getFoosRoDah());

The orignal: Foos - RODA!!!

The deep clone NO!!

THIS IS PATRICK!!

The implementation of deep copying allows for the Java Clone mechanism to copy everything including the classes Object instances.

There are many advantages and disadvantages to the Java clone mechanism, the first one is that it allows you to create actual copies of objects which can be modified without modifying the original object. In comparison to copy Constructors it takes less lines of code to implement especially shallow clones and some deep clones depending on how many object fields an object has. It is the fastest way to copy arrays. (Joshi, 2017)

However, there are disadvantages. These disadvantages are that the implementation is that Object clone() method in the Cloneable interface is protected so for other classes to clone objects, the class needs to make a public override of it. The Object clone() method has no control over constructors which means that the cloning mechanism cannot modify final fields because they can only be changed in constructors. (Joshi, 2017)

An annoying disadvantage of cloning is that for deep cloning to work all class fields need to implement shallow cloning at minimum in order to use it. The class fields objects must have their clone methods implemented in the original class’s clone method.

The clone method needs to be implemented in all of the superclasses that the object inherit from in order for the clone method to inherit up to the original Object.clone() method. (Joshi, 2017)

There are two alternatives to the Java clone mechanism. These are Java Copy Constructor and Serialization both can be used for creating deep copies.

Java Copy Constructors are the simpler of the two as a Copy Constructor is a Constructor that takes the constructor’s class as a parameter. Then all of the fields of the class referenced in the parameter will be copied over to the new class using Class copyClass = new Class(originalClass) to copy the class. (Gupta, 2013). For classes that have no children, this is simple but for it to be implemented in the child classes, the super class must also have this type of constructor in it. Then its constructor has to then run the parent’s copy constructor. This results in the contractors creating an instance of the parent class.

A variance of the Copy Constructor is the implementation of a “static factory method” which have a class parameter and makes a new class object by using a class constructor. These “factory methods” will copy all of the old object data to another new class object. (Gupta, 2013)

The other way is through serialization. This method involves serializing the object and then un-serializing it provided that the class is serializable in the first place. Sterilization is both performance expensive and limited and hard to effectively implement. (Gupta, 2013)