**Java – Cloning Mechanism**

**Introduction:**

Developer can create objects in Java using different ways (new keyword, Class.newInstance() method, Constructor.newInstance() method, clone method and deserialization). In this documentation, I will only focus on clone method, advantages and disadvantages of this, how Java language requires the cloning mechanism, an example of how cloning is implemented, different options available to developers for implementing cloning when cloning inheritance hierarchies and last but not the least alternatives to the Java clone mechanism.

**What is Cloning Mechanism?**

Object cloning as the name suggests, refers to creation of exact copy of an object. It creates a new instance of the class of current object and then initializes all its fields with exactly the same contents of the corresponding fields of the object.

Example: Given an object, if you want to create multiple objects which are ‘exact’ copies of that object, but do not want to go through the process of defining a new object instance, then cloning is what you will do.

In order to implement cloning, the classes need to be configured and follow the below steps:

* Implement the Cloneable interface in our class or its superclass or interface.
* Define a clone() method that should handle CloneNotSupportedException (either throw or log).
* And, in most cases from our clone() method, we call the clone() method of the superclass.

There are two different type of cloning: Shallow Cloning and Deep Cloning.

1. Shallow Cloning - Generally clone() method of an object creates a new instance of the same class, and copies all the fields to the new instance and returns it. This is nothing but a shallow cloning. Object class provides a clone() method and provides a support for the shallow cloning. It returns ‘Object’ as type and you need to explicitly cast back to your original object.

**Note:** If original object has any references to other objects as fields, then only the references of those objects are copied into clone object, a copy of those objects is not created.

1. Deep Cloning - Deep cloning of an object will have an exact copy of all the fields of the original object, just like the shallow cloning. If the original object has any references to the other objects as fields, then a copy of those objects is also created by calling clone() method on them.   
   **Note:** To create a deep copy of an object, you must override the clone().

**Why Java Language requires Cloning mechanism?**

Objects in Java are referred using reference types, and there is no direct way to copy the contents of an object into a new object. One reference to another merely creates another reference to the same object. Therefore, a special clone() method exists for all reference types in order to provide a standard mechanism for an object to make a copy of itself.Also, Object Cloning saves the extra processing task for creating the exact copy of an object.

**Discussion and illustration by providing an example of how the cloning mechanism is implemented in Java.**

|  |
| --- |
| **/\* Student.Java \*/**  **package** task3;  **public** **class** Student **implements** Cloneable{  **private** String firstName;  **private** String lastName;  **private** **int** age;  **private** StudentAddress studentAddress;  /\* Student constructor \*/  **public** Student(String firstName, String lastName, **int** age, StudentAddress  studentAddress) {  **this**.firstName = firstName;  **this**.lastName = lastName;  **this**.age = age;  **this**.studentAddress = studentAddress;  }  /\* setters and getters for firstName, lastName, age and studentAddress \*/  **public** String toString(){  **return** "Student Name:"+ **this**.firstName + " " + **this**.lastName  +" \nAge:"+**this**.age  +" \nAddress:"+studentAddress;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (obj == **this**) {  **return** **true**;  }  **if** (!(obj **instanceof** Student)) {  **return** **false**;  }  Student studentObj = (Student) obj;  **return** **this**.age == studentObj.age  && **this**.firstName.equalsIgnoreCase(studentObj.firstName)  && **this**.lastName.equalsIgnoreCase(studentObj.lastName);  }  @Override  **public** Object clone() **throws** CloneNotSupportedException{  **return** **super**.clone();  }  }  **/\* StudentAddress.Java \*/**  **package** task3;  **public** **class** StudentAddress {  **private** String houseNo;  **private** String street;  **private** String city;  **private** **int** postalCode;  /\* Student Address constructor \*/  **public** StudentAddress(String houseNo, String street, String city, **int**  postalCode) {  **this**.houseNo = houseNo;  **this**.street = street;  **this**.city = city;  **this**.postalCode = postalCode;  }  /\* setters and getters for houseNo, street, city and postal code \*/  @Override  **public** String toString() {  **return** "StudentAddress{" +  "houseNo='" + houseNo + '\'' +  ", street='" + street + '\'' +  ", city='" + city + '\'' +  ", postalCode='" + postalCode + '\'' +  '}';  }  }  **/\* JavaCloning.Java \*/**  **package** task3;  **public** **class** JavaCloning {  **public** **static** **void** main(String args[]){  StudentAddress studentAddress=**new** StudentAddress("109A","Knight  Roads", "Lower Hutt Central", 5010);  Student student = **new** Student("Kristel", "Villanueva", 23,  studentAddress);  Student studentClone=**null**;  **try** {  studentClone=(Student) student.clone();  }**catch**(CloneNotSupportedException cnse){  cnse.printStackTrace();  }    System.***out***.println("Cloned Student Object: \n" + studentClone);  }  }  **/\* Output \*/** |

**The above example is using Shallow Cloning and the explanation is as follows:**

* **Student** class has 4 attributes – **firstname** (of type **String**), **lastname** (of type **String**), **age**(**Integer**) and **studentAddress**(**StudentAddress**).

**private** String firstName; **private** String lastName;

**private** **int** age; **private** StudentAddress studentAddress

* **StudentAddress** is a different class containing 4 attributes – **houseNo**, **street**, **city** and **postalCode.**

**private** String houseNo; **private** String street;

**private** String city; **private** **int** postalCode;

* I implement **Cloneable** interface in **Student.java**. It overrides **Object.clone()** method and calls **super.clone()**.

***Note*** that if the interface **Cloneable** is not implemented then a **java.lang.CloneNotSupportedException** exception is thrown on calling the **clone()** method on **Student** objects.

**public** **class** Student **implements** Cloneable{

….

@Override

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

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

}

}

* Inside the **main()** method of **JavaCloning.java** I create first an instance of **StudentAddress,** named **studentAddress** then I alsocreate an instance of **Student**, named **student** as shown below:

StudentAddress studentAddress=**new** StudentAddress("109A","Knight

Roads", "Lower Hutt Central", 5010);

Student student = **new** Student("Kristel", "Villanueva", 23,

studentAddress);

* Next, on the same **JavaCloning.java** I create a clone of **student** object by invoking **student.clone()** which creates **studentClone** object.

Student studentClone=**null**;

**try** {

studentClone=(Student) student.clone();

}**catch**(CloneNotSupportedException cnse){

cnse.printStackTrace();

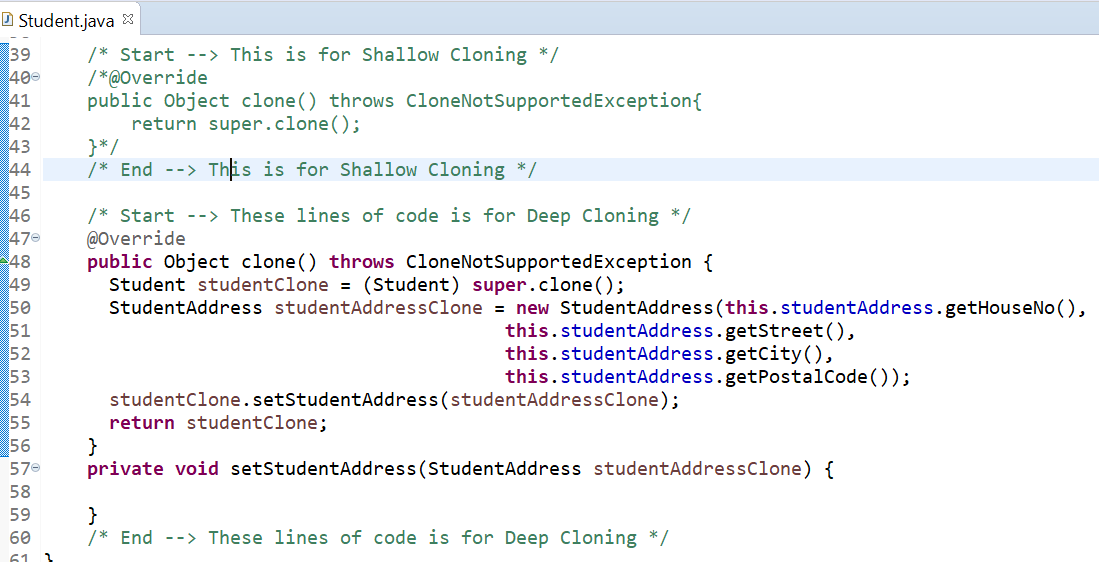
}

* And lastly, I print **studentClone** and find that it contains the same contents as **student**, implying that **studentClone** is a clone of **student**.

System.***out***.println("Cloned Student Object: \n" + studentClone);



**For Deep Cloning, StudentAddress.java** and **JavaCloning.java** will be the same. **Student.java** have to be change a bit, inside the overridden **clone()** method, instead of calling **super.clone()**, a clone of an **Student** object is constructed step-by-step as shown below:



**What are the Advantages and Disadvantages of using Cloning Mechanism?**

**Advantages**

* Cloning requires much fewer lines of code — just an abstract class with a 4- or 5-line long clone() method, but we will need to override it if we need deep cloning.
* It is the easiest way of copying objects, especially if we are applying it to an already developed or an old project. We just need to define a parent class, implement Cloneable in it, provide the definition of the clone() method, and we are ready. Every child of our parent will get the cloning feature.
* We should use clone to copy arrays because that’s generally the fastest way to do it.
* As of release 1.5, calling clone on an array returns an array whose compile-time  
  type is the same as that of the array being cloned, which clearly means calling a clone on arrays does not require type-casting.

**Disadvantages**

* Using the Object.clone() method requires us to add lots of syntax to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() and casting it on our object.
* The Cloneable interface lacks the clone() method. Actually, Cloneable is a marker interface and doesn’t have any methods in it, and we still need to implement it just to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* We don’t have any control over object construction because Object.clone() doesn’t invoke any constructor.
* If we are writing a clone method in a child class, e.g. Person, then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying, so the reference fields of our newly cloned object will still hold objects whose fields of our original object was holding. In order to overcome this, we need to implement clone() in every class whose reference our class is holding and then call their clone separately in our clone() method like in the example below.
* We cannot manipulate final fields in Object.clone() because final fields can only be changed through constructors. In our case, if we want every Person object to be unique by id, we will get the duplicate object if we use Object.clone() because Object.clone() will not call the constructor, and final id field can’t be modified from Person.clone().

**Suggested alternatives to the Java clone mechanism.**

Cloning mechanism has few alternatives as follows:

* [copy constructor](http://en.wikipedia.org/wiki/Copy_constructor) – a copy constructor is a constructor which accepts another instance of the same class as a parameter.
* [factory method](http://en.wikipedia.org/wiki/Factory_method_pattern) – these methods are not always adequate when the concrete type of the cloned object is not known in advance.
* Use of serialization and deserialization is another alternative to using clone.

**Different options available to developers for implementing cloning when cloning inheritance hierarchies.**

Different options for cloning inheritance hierarchies are: Java Deep cloning with serialization and deserialization, Copy Constructors and factory method.

**References:**

<https://howtodoinjava.com/java/cloning/a-guide-to-object-cloning-in-java/>

https://en.wikipedia.org/wiki/Cloning\_(programming)

http://techalpine.com/lets-clone-in-java/

https://stackoverflow.com/questions/5802118/why-do-we-use-the-clone-method-in-java

https://www.programmingmitra.com/2017/01/Java-cloning-copy-constructor-versus-Object-clone-or-cloning.html

https://javarevisited.blogspot.com/2013/09/how-clone-method-works-in-java.html