**Immutable class**

Immutable class is a class which once created, it's content can not be changed. Immutable classes are good choice for HashMap key as their state cannot be changed once they are created. Objects of immuable classes are also thread safe as threads can not change the value of its fields once it is created.

Creating a basic immutable class is very easy, but as you dig deep you will start facing many challenges. This makes Immutability a very famous interview topic for mid level java developers.

This article is a continuation of our article [**How to create Immutable Class in Java**](https://codepumpkin.com/create-immutable-class-java)where we have discussed about creating basic immutable class using traditional approach as well as using Builder Design Pattern approach. In this article, we will discuss about How we will achieve immutability, if it has member variables of any third party class which is mutable. Or what will you do if you have reference of any built-in java collection class which is mutable like ArrayList, LinkedList, etc.

**For Example,**class User which has three fields firstName, lastName and Address of type String. To make this class immutable,

* We will declare class as final and all the fields as private final.
* We will provide one parameterized constructor and getter methods.
* We will not provide any setter method, so that field values can not be changed after object creation.

**User class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public final class User {      private final String firstName;      private final String lastName;      private final String address;        public User(String firstName, String lastNa++m++e,+++ String address) {          super();          this.firstName = firstName;          this.lastName = lastName;          this.address = address;      }      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public String getAddress() {          return address;      }  } |

Above class is immutable as we can not change field values after object creation. Now, lets assume that there is a change in requirement and instead of storing address in String object, we have to store it in more organized way. We have to use rich third party Address class which has options to store address very efficiently using different fields for firstLine, secondLine and city.

**Address class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | public class Address implements Cloneable{      String firstLine;      String secondLine;      String city;        public Address(String firstLine, String secondLine, String city) {          super();          this.firstLine = firstLine;          this.secondLine = secondLine;          this.city = city;      }        public String getFirstLine() {          return firstLine;      }        public void setFirstLine(String firstLine) {          this.firstLine = firstLine;      }        public String getSecondLine() {          return secondLine;      }        public void setSecondLine(String secondLine) {          this.secondLine = secondLine;      }        public String getCity() {          return city;      }        public void setCity(String city) {          this.city = city;      }        @Override      public String toString() {          return "Address [firstLine=" + firstLine + ", secondLine=" + secondLine                  + ", city=" + city + "]";      }  } |

Lets change type of address from String to Address in User class.

**User class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public final class User {      private final String firstName;      private final String lastName;      private final Address address;        public User(String firstName, String lastName, Address address) {          super();          this.firstName = firstName;          this.lastName = lastName;          this.address = address;      }      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public Address getAddress() {          return address;      }  } |

Bingo..!! User class now stores address in more organized way. But **is User class still immutable?**

As of now all the fields of User class were String and String itself is immutable. But Address class has setter methods and hence now User class has one mutable member field.  **Can this break immutability of User class?** Lets try to break immutability of User class.

**ImmutableDemo class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class ImmutableDemo {        public static void main(String s[]) {            User u = new User("Sherlock",                            "Homes",                            new Address("221B", "Baker Street", "London"));            // fetch address from the User object and store it in local variable address          Address address = u.getAddress();          System.out.println(address);            // change address in local variable          address.setFirstLine("001D");          address.setSecondLine("Chandani Chawk");          address.setCity("Delhi");            // display address from User object          System.out.println(u.getAddress());      }  } |

**Output**

|  |  |
| --- | --- |
| 1  2 | Address [firstLine=221B, secondLine=Baker Street, city=London]  Address [firstLine=001D, secondLine=Chandani Chawk, city=Delhi] |

Here, first we got reference of address object using getAddress() method. We have stored this reference in new local variable, it was still pointing to the same address instance. So, when we changed value of firstLine, secondLine and city fields of address instance, it updated the address instance being used by User object. And hence, when we tried to get address of user, it printed updated address. **User class is no more Immutable.**

**How can we achieve immutablility in such case?**

**Option 1)**  
We will not provide any setter methods in Address class so that nobody can change properties of address class. If you will answer this, interviewer will counter you saying Address is a third-part java class i.e. It is being referred from third party jar file and we do not have access to the source code of Address.

**Option 2)**  
So **what should we do to change the behaviour of Address class and remove setter methods from it?**We can create child class of Address class, override all the setter methods and then explicitely throw **UnsupportedOperationException**from those setter methods.

**ChildAddress class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class ChildAddress extends Address{        public ChildAddress(String firstLine, String secondLine, String city) {          super(firstLine,secondLine,city);      }        public void setFirstLine(String firstLine) {          throw new UnsupportedOperationException();      }        public void setSecondLine(String secondLine) {          throw new UnsupportedOperationException();      }        public void setCity(String city) {          throw new UnsupportedOperationException();      }  } |

Now in our User class, we will create a reference of ChildAddress instead of Address class.

**User class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public final class User {      String firstName;      String lastName;      ChildAddress address;        public User(String firstName, String lastName, ChildAddress address) {          super();          this.firstName = firstName;          this.lastName = lastName;          this.address = address;      }      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public ChildAddress getAddress() throws CloneNotSupportedException {          return (ChildAddress) address.clone();      }    } |

**ImmutableDemo class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class ImmutableDemo {        public static void main(String s[]) throws CloneNotSupportedException{            User u = new User("Sherlock",                            "Homes",                            new ChildAddress("221B", "Baker Street", "London"));            // fetch address from the User object and store it in local variable address          Address address = u.getAddress();          System.out.println(address);            // change address in local variable          address.setFirstLine("001D");          address.setSecondLine("Chandani Chawk");          address.setCity("Delhi");            // display address from User object          System.out.println(u.getAddress());      }  } |

This works fine and nobody can now change value of any field of Address.

**Output**

|  |  |
| --- | --- |
| 1  2  3  4 | Exception in thread "main" java.lang.UnsupportedOperationException      at com.codepumpkin.miscellaneous.ChildAddress.setFirstLine(Address.java:57)      at com.codepumpkin.miscellaneous.ImmutableDemo.main(ImmutableDemo.java:16)  Address [firstLine=221B, secondLine=Baker Street, city=London] |

**Is there any problem with this approach?**

Well yes. what if some of the reference variables inside Address class is also Mutable Objects. In that case we need to override their  setter methods as well. This approch becomes more complex when there are many nested Mutable class references.

**Option 3)**

Another option is to modify getAddress method of User class. Instead of returning the original Address instance, we will return deep cloned copy of that Adress instance. Even if third party user makes any changes to this cloned address object, it will not affect the original address object of User object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public final class User {      private final String firstName;      private final String lastName;      private final Address address;        public User(String firstName, String lastName, Address address) {          super();          this.firstName = firstName;          this.lastName = lastName;          this.address = address;      }      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public Address getAddress() {          return address.clone();      }  } |

clone() method only works if Address has implemented Cloneable interface. If it has not implemented it, then we have to manually deep copy all the fields of Address class. But most of the user library has support for Cloneable and Serializable interfaces.

Most of the real life implementation uses this third approach. Let's check how robust this implementation is.

**ImmutableDemo class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class ImmutableDemo {        public static void main(String s[]) throws CloneNotSupportedException{            User u = new User("Sherlock",                            "Homes",                            new Address("221B", "Baker Street", "London"));            // fetch address from the User object and store it in local variable address          Address address = u.getAddress();          System.out.println(address);            // change address in local variable          address.setFirstLine("001D");          address.setSecondLine("Chandani Chawk");          address.setCity("Delhi");            // display address from User object          System.out.println(u.getAddress());      }  } |

Even if we changed values inside cloned address object, it has not affected original Address value inside User object. If interviewer ask you this question, you can directly tell him about this solution, but you should also be aware about other available options.

**Mutable Collections as field of Immutable Object**

**Example,**Assume that instead of storing just one Address, we are storing List of Addresses inside User class i.e. Primary Address, Secondary Address, Work Address etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | import java.util.ArrayList;    public final class User {      String firstName;      String lastName;      ArrayList<Address> addressList;        public User(String firstName, String lastName, ArrayList<Address> addressList) {          super();          this.firstName = firstName;          this.lastName = lastName;          this.addressList = addressList;      }      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public ArrayList<Address> getAddress(){          return (ArrayList)addressList.clone();      }    } |

**Option 1)**

When we have Collection like ArrayList or LinkedList as member variables, we should not use their in-built clone() method. Their clone() method returns shallow copy of this ArrayList instance. The elements themselves are not copied. So in such case, we can write our own method to deep copy ArrayList object.

**Option 2)**

We can also use the copy-constructors â€“ new ArrayList(originalList)to create a deep cloned ArrayList object. It would be easier than writing our own deep copy method.

**Option 3)**

Collection framework also provides implementation of Unmodifiable Collection classes in Collections utility class. These Unmodifiable collections are just a wrapper around normal Collection classes which throws UnsupportedOperationException from all the methods which tries to modify Collection i.e. add(), remove(), etc. You can check implementation of UnmodifiableList at grepcode over [here](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Collections.java#Collections.unmodifiableList%28java.util.List%29).

**So how we will use this Unmodifiable collections?**Insterad of returning plain ArrayList object, we now return UnmodifiableList of Addresses from getAddress() method of User class.

It is same as Option 2 of mutable fields inside Immutable class.

**User class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | import java.util.ArrayList;  import java.util.Collections;    public final class User {      String firstName;      String lastName;      ArrayList<Address> addressList;        public User(String firstName, String lastName, ArrayList<Address> addressList) {          super();          this.firstName = firstName;          this.lastName = lastName;          this.addressList = addressList;      }      public String getFirstName() {          return firstName;      }      public String getLastName() {          return lastName;      }      public ArrayList<Address> getAddressList(){          return (ArrayList)Collections.unmodifiableCollection(addressList);      }    } |

**ImmutableDemo class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import java.util.ArrayList;    public class ImmutableDemo {        public static void main(String s[]) throws CloneNotSupportedException{            ArrayList<Address> addressList = new ArrayList<>();          addressList.add(new Address("221B", "Baker Street", "London"));          addressList.add(new Address("66", "Perry Street", "West Village"));              User u = new User("Sherlock",                            "Homes",                            addressList);            // fetch address list from the User object and store it in local variable localAddressList          ArrayList<Address> localAddressList = u.getAddressList();          System.out.println(localAddressList);            // remove address at 0th position in ArrayList          localAddressList.remove(0);            // display address list from User object          System.out.println(u.getAddressList());      }  } |

**Output**

|  |  |
| --- | --- |
| 1  2  3 | Exception in thread "main" java.lang.ClassCastException: java.util.Collections$UnmodifiableCollection cannot be cast to java.util.ArrayList      at com.codepumpkin.miscellaneous.User.getAddressList(User.java:24)      at com.codepumpkin.miscellaneous.ImmutableDemo.main(ImmutableDemo.java:19) |