Immutable Class

**Q1) what is an immutable class?**

Ans) Immutable class is a class which once created, it’s contents can’t be changed. Immutable objects are the objects whose state can’t be changed once constructed. e.g. String class

An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

**Q2) How to create an immutable class?**

Ans) To create an immutable class following steps should be followed:

1. Create a final class.

2. Set the values of properties using constructor only.

3. Make the properties of the class final and private

4. Do not provide any setters for these properties.

5. If the instance fields include references to mutable objects, don't allow those objects to be changed:

1. Don’t provide methods that modify the mutable objects.

2. Don’t share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

E.g.

public final class FinalPersonClass {

private final String name;

private final int age;

public FinalPersonClass(final String name, final int age) {

super();

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

public String getName() {

return name;

}

}

**Q3) Immutable objects are automatically thread-safe –true/false?**

Ans) True. Since the state of the immutable objects cannot be changed once they are created they are automatically synchronized/thread-safe.

**Q4) Which classes in java are immutable?**

Ans) All wrapper classes in java.lang are immutable –

String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

•java.lang.String (already mentioned)

•The wrapper classes for the primitive types: java.lang.Integer, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Boolean, java.lang.Long, java.lang.Double, java.lang.Float

•java.lang.StackTraceElement (used in building exception stacktraces)

•Most enum classes are immutable, but this in fact depends on the concrete case. (Don't implement mutable enums, this will screw you up somewhen.) I think that at least all enum classes in the standard API are in fact immutable.

•java.math.BigInteger and java.math.BigDecimal

•java.io.File. Note that this represents an object external to the VM (a file on the local system), which may or may not exist, and has some methods modifying and querying the state of this external object. But the File object itself stays immutable. (All other classes in java.io are mutable.)

•java.awt.Font - representing a font for drawing text on the screen (there may be some mutable subclasses, but this would certainly not be useful)

•java.awt.BasicStroke - a helper object for drawing lines on graphic contexts

•java.awt.Color - (at least objects of this class, some subclasses may be mutable or depending on some external factors (like system colors)), and most other implementations of java.awt.Paint like

•java.awt.GradientPaint,

•java.awt.LinearGradientPaint

•java.awt.RadialGradientPaint,

•(I'm not sure about java.awt.TexturePaint)

•java.awt.Cursor - representing the bitmap for the mouse cursor (here too, some subclasses may be mutable or depending on outer factors)

•java.util.Locale - representing a specific geographical, political, or cultural region.

•java.util.UUID - a as much as possible globally unique identifier

•while most collections are mutable, there are some wrapper methods in the java.util.Collections class, which return an unmodifiable view on a collection. If you pass them a collection not known anywhere, these are in fact immutable collections. Additionally, Collections.singletonMap(), .singletonList, .singleton return immutable one-element collections, and there are also immutable empty ones.

•java.net.URL and java.net.URI - representing a resource (on the internet or somewhere else)

•java.net.Inet4Address and java.net.Inet6Address, java.net.InetSocketAddress

•Most subclasses of java.security.Permission (representing permissions needed for some action or given to some code), but not java.security.PermissionCollection and subclasses.

One could say the primitive types are immutable, too - you can't change the value of 42, can you?

--------------------------------------------------------------------------------

is Class AccessControlContext a immutable class

AccessControlContext does not have any mutating methods. And its state consists of a list of ProtectionDomains (which is an immutable class) and a DomainCombiner. DomainCombiner is an interface, so in principle the implementation could do something different on each call.

In fact, also the behaviour of the ProtectionDomain could depend on the current policy in force - it is disputable whether to call such an object immutable.

and AccessController?

There are no objects of type AccessController, since this is a final class with no accessible constructor. All methods are static. One could say AccessController is neither mutable nor immutable, or both.

The same is valid for all other classes which can't have objects (instances), most famously:

•java.lang.System (but this has some mutable static state - in, out, err)

•java.lang.Math (this too - the random number generator)

•java.lang.reflect.Array

•java.util.Collections

•java.util.Arrays

**Q5) What are the advantages of immutability?**

The advantages are:

* Immutable objects are automatically thread-safe, the overhead caused due to use of synchronization is avoided.
* Once created the state of the immutable object cannot be changed so there is no possibility of them getting into an inconsistent state.
* The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state cannot be changed ever after construction.
* The best use of the immutable objects is as the keys of a map.