What is immutable Object?

An object is considered immutable if its state cannot change after it is constructed.

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.

This means that the public API of an immutable object guarantees us that it will behave in the same way during its whole lifetime.

Immutable Classes in JDK

- java.lang.String
- 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
- 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
- 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 - an 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.

- All classes of java.time except DateTimeException are immutable. Most of the classes of the subpackages of java.time are immutable too.

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

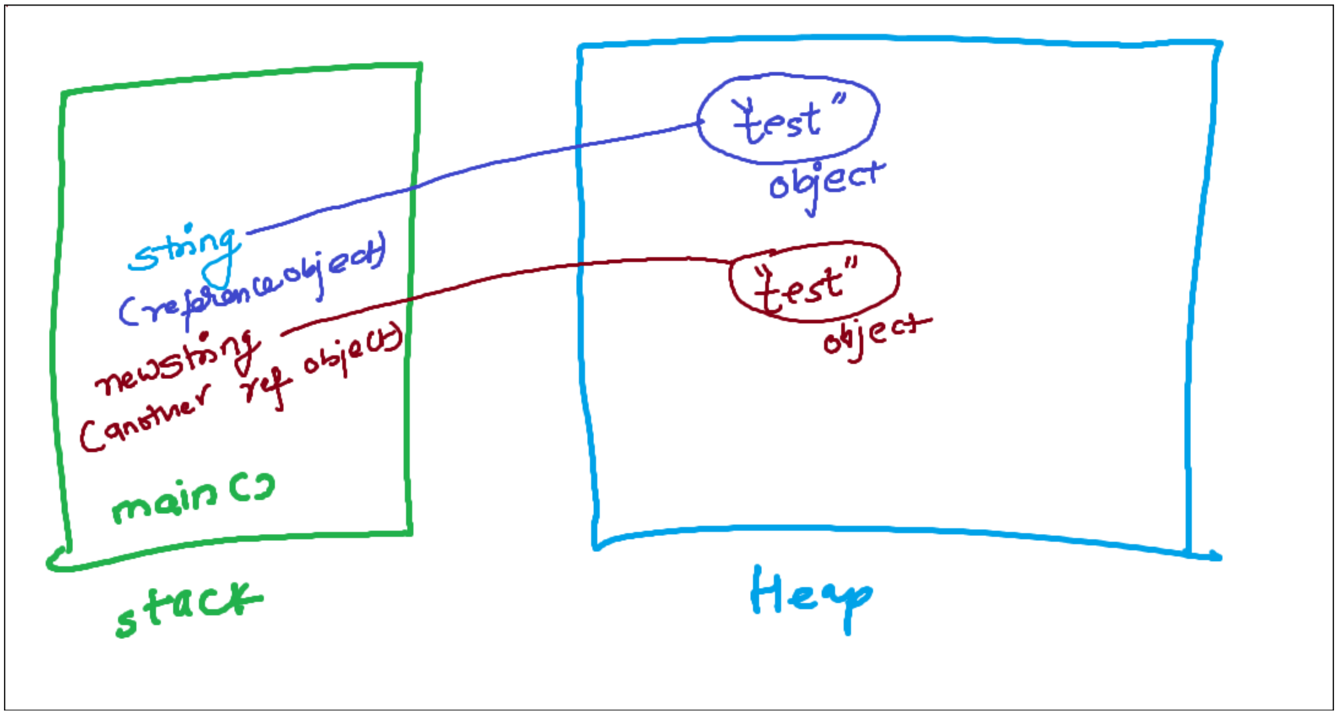
Advantages of Immutability

Immutable objects provide a lot of advantages over mutable objects. Let us discuss them.

- **Predictability**: guarantees that objects won’t change due to coding mistakes or by 3rd party libraries. As long as we reference a data structure, we know it is the same as at the time of its creation.
- **Validity**: is not needed to be tested again and again. Once we create the immutable object and test its validity once, we know that it will be valid indefinitely.
- **Thread-safety**: is achieved in the program as no thread can change immutable objects. It helps in writing code in a simple manner without accidentally corrupting the shared data objects.
- **Catchability**: can be applied to immutable objects without worrying about state changes in the future. Optimization techniques, like memorization, are only possible with immutable data structures.

Eg.

```
String string = "test";
String newString = string.toLowerCase();  //Creates a new String
```



Immutability in Collections

Similarly, for Collections, Java provides a certain degree of immutability with three options:

- Unmodifiable collections
- Immutable collection factory methods (Java 9+)
- Immutable copies (Java 10+)

```
Collections.unmodifiableList(recordList);  //Unmodifiable list
```

```
List.of(new Record(1, "test"));  //Factory methods in Java 9
```

```
List.copyOf(recordList);  //Java 10
```

Note that such collections are only shallowly immutable, meaning that we can not add or remove any elements, but the collection elements themselves aren’t guaranteed to be immutable. If we hold the reference of a collection element, then we can change the element’s state without affecting the collection.

In the following example, we cannot add or remove the list items, but we can change the state of an existing item in the list.

```
List<String> students = List.of("John","Marry");
students.add("Jerry"); //cannot add
/**
 * Exception in thread "main" java.lang.UnsupportedOperationException
 *     at java.base/java.util.ImmutableCollections.uoe
 */
```

```
List<Integer> luckyNumber = new ArrayList<>();
luckyNumber.add(11);
luckyNumber.add(21);

List<Integer> unmodifiableList = Collections.unmodifiableList(luckyNumber);
unmodifiableList.add(31);//Exception in thread "main" java.lang.UnsupportedOperationException
```

```
List<Integer> luckyNumber = new ArrayList<>();
luckyNumber.add(11);
luckyNumber.add(21);

List<Integer> integers = List.copyOf(luckyNumber);
integers.add(31);//Exception in thread "main" java.lang.UnsupportedOperationException
```

How to Create an Immutable Class?

Java documentation itself has some guidelines identified to write immutable classes in this link. We will understand what these guidelines actually mean.

- **Do not provide setter methods**. Setter methods are meant to change an object’ s state, which we want to prevent here.
- **Make all fields final and private**. Fields declared private will not be accessible outside the class, and making them final will ensure that we can not change them even accidentally.
- **Do not allow subclasses to override methods**. The easiest way is to declare the class as final. Final classes in Java can not be extended.
- Special attention to **“immutable classes with mutable fields “**. Always remember that member fields will be either mutable or immutable. Values of immutable members (primitives, wrapper classes, String etc) can be returned safely from the getter methods.

For mutable members (POJO, collections etc), we must copy the content into a new Object before returning from the getter method.
Let us apply all the above rules to create an immutable custom class. Notice that we are returning a new copy of ArrayList from the getTokens() method. By doing so, we are hiding the original tokens list so no one can even get a reference of it and change it.

```
public final class Employee {

  private final Long id; //Wrapper classes are immutable
  private final String name; //String are immutable
  private final List<String> tokens; //List is mutable, we need to handle with care.

  public Employee (Long id, String name, List<String> tokens) {
    this.id = id;
    this.name = name;
    this.tokens = tokens;
  }

  public long getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public List<String> getTokens() {
    return new ArrayList<>(tokens);
  }
}
```

- getTokens() returns a copy of tokens, so actual tokens with Employee is unmodifiable.
- In constructor you can set this.tokens = Collections.unmodiflableList(tokens)
- getTokens() you can also return Collections.unmodiflableList(tokens) or List.copyOf(records)
- If there is a java.util.Date field, Date is mutable you can make an internal copy of this.dob= new Date(dateParater.getTime);
- Since is mutable, you can use LocalDate or LocalDateTime provided in java-8 which are immutable, so prefer using LocalDate over Date.

Javadoc - https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

Assignment: Write a User Immutable class having attributes id, name, dateOfBirth, List of addresses.