Mutability and immutability in java

Mutable objects - The objects in which you can change the fields and states after the object is created are known as Mutable objects.
Example: java.util.Date,
StringBuilder, \
StringBuffer etc.

When we make a change in existing mutable objects, no new object will be created; instead, it will alter the value of the existing object. These object's classes provide methods to make changes in it.

The Getters and Setters (get() and set() methods) are available in mutable objects. The Mutable object may or may not be thread-safe.

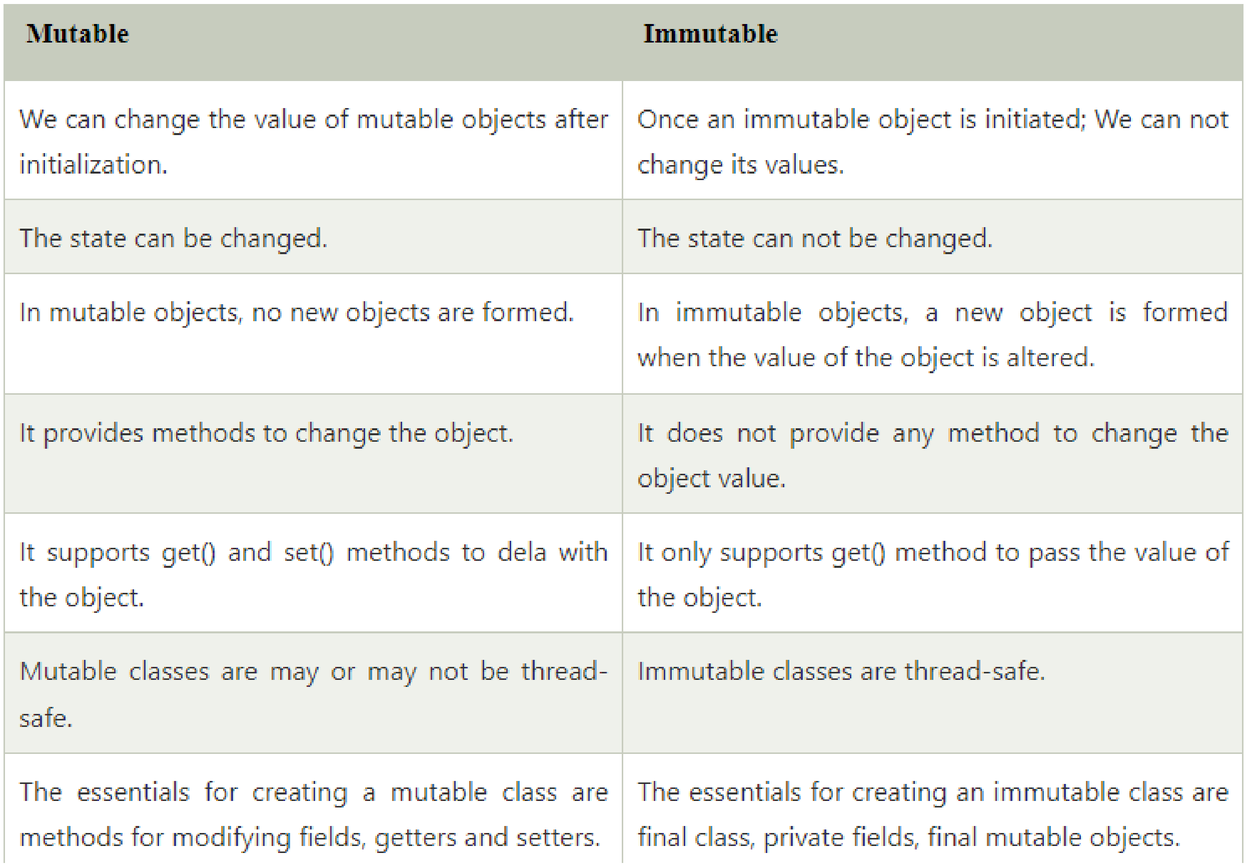
Immutable objects - The immutable objects are objects whose value can not be changed after initialization, i.e. we cannot change the state of the object or the content of the object once it is created.

Ex - Wrapper classes Integer, Float, Double
Strings
Immutable collection classes such as Collections.singletonMap() etc.
java.lang.StackTraceElement
java.util.Locale
java.util.UUID

In a nutshell, immutable means unmodified or unchangeable. Once the immutable objects are created, its object values and state can not be changed.

Only Getters (get() method) are available, not Setters (set() method) for immutable objects.

Key differences between mutable and immutable objects -



Mutable	**Immutable**
We can change the value of mutable objects after initialization.	Once an immutable object is initiated; We can not change its values.
The state can be changed.	The state can not be changed.
In mutable objects, no new objects are formed.	In immutable objects, a new object is formed when the value of the object is altered.
It provides methods to change the object.	It does not provide any method to change the object value.
It supports get() and set() methods to dela with the object.	It only supports get() method to pass the value of the object.
Mutable classes are may or may not be thread-safe.	Immutable classes are thread-safe.
The essentials for creating a mutable class are methods for modifying fields, getters and setters.	The essentials for creating an immutable class are final class, private fields, final mutable objects.

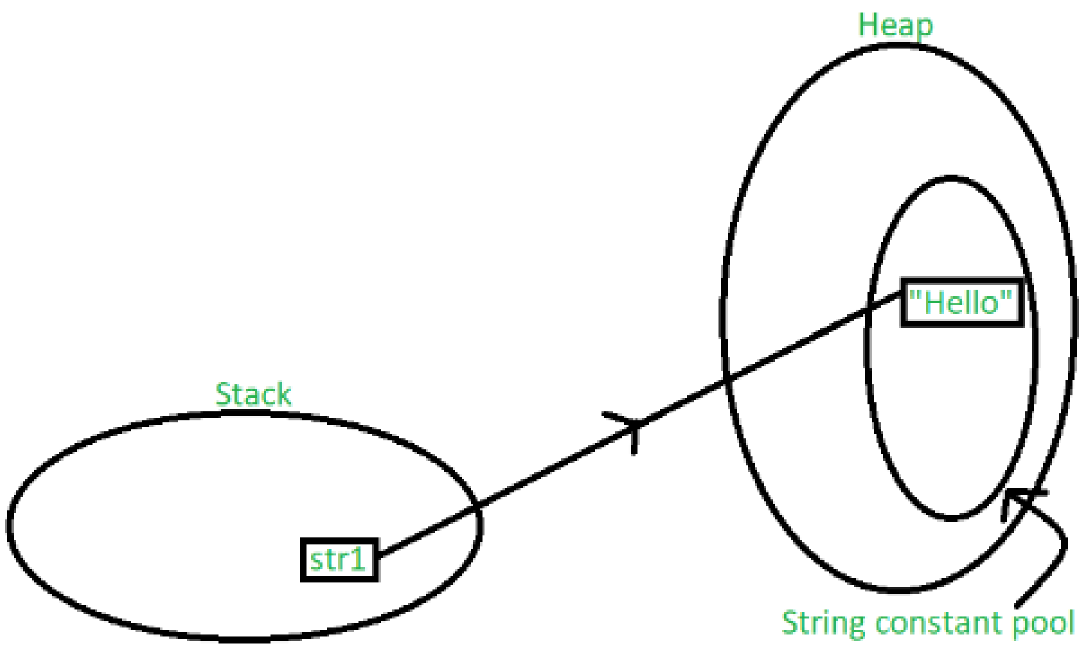
Understanding immutability using Strings, how is the class String immutable?

String is a sequence of characters. One of the most important characteristics of a string in Java is that they are immutable. In other words, once created, the internal state of a string remains the same throughout the execution of the program. This immutability is achieved through the use of a special string constant pool in the heap.

A string constant pool is a separate place in the heap memory where the values of all the strings which are defined in the program are stored. When we declare a string, an object of type String is created in the stack, while an instance with the value of the string is created in the heap. On standard assignment of a value to a string variable, the variable is allocated stack, while the value is stored in the heap in the string constant pool.

```
String str1 = "Hello";
```

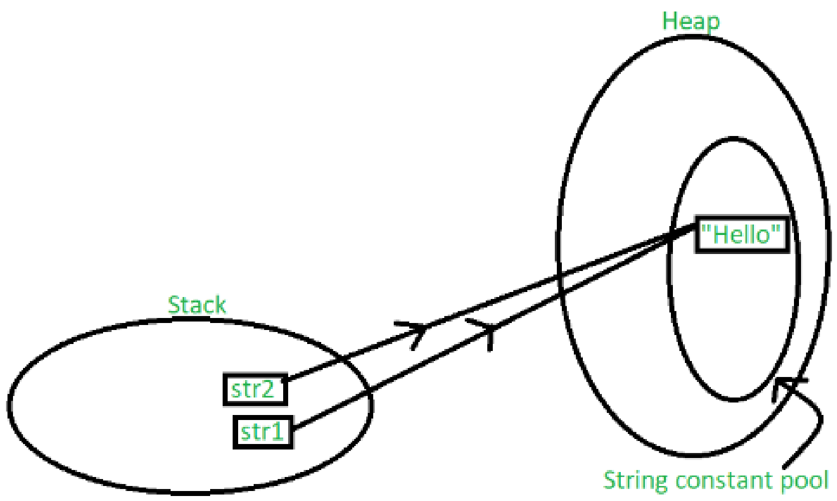
The following illustration explains the memory allocation for the above declaration:



Now, let's take the same example with multiple string variables having the same value as follows:

```
String str1 = "Hello";
String str2 = "Hello";
```

The following illustration explains the memory allocation for the above declaration:



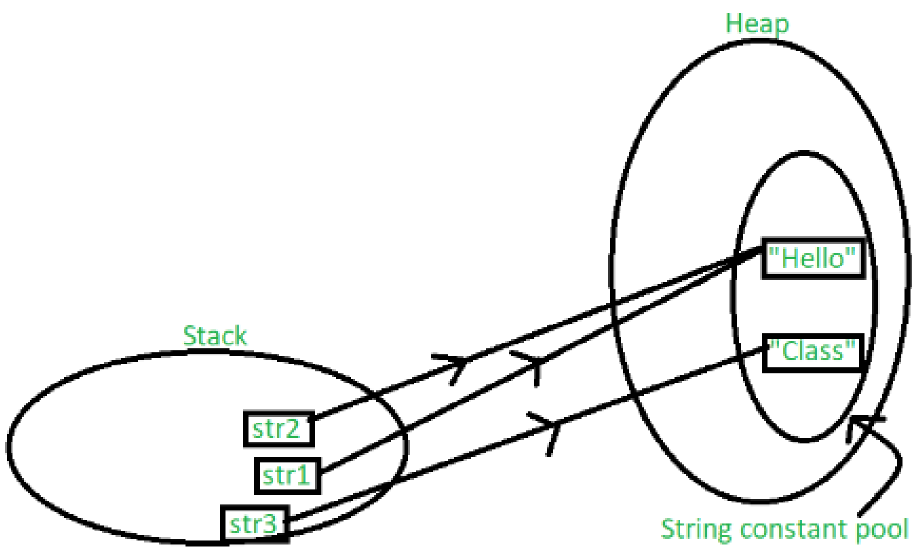
In this case, both the string objects get created in the stack, but another instance of the value "Hello" is not created in the heap. Instead, the previous instance of "Hello" is re-used.

In other words, the string constant pool exists mainly to reduce memory usage and improve the re-use of existing instances in memory. When a string object is assigned a different value, the new value will be registered in the string constant pool as a separate instance.

Lets understand this with the following example:

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = "Class";
```

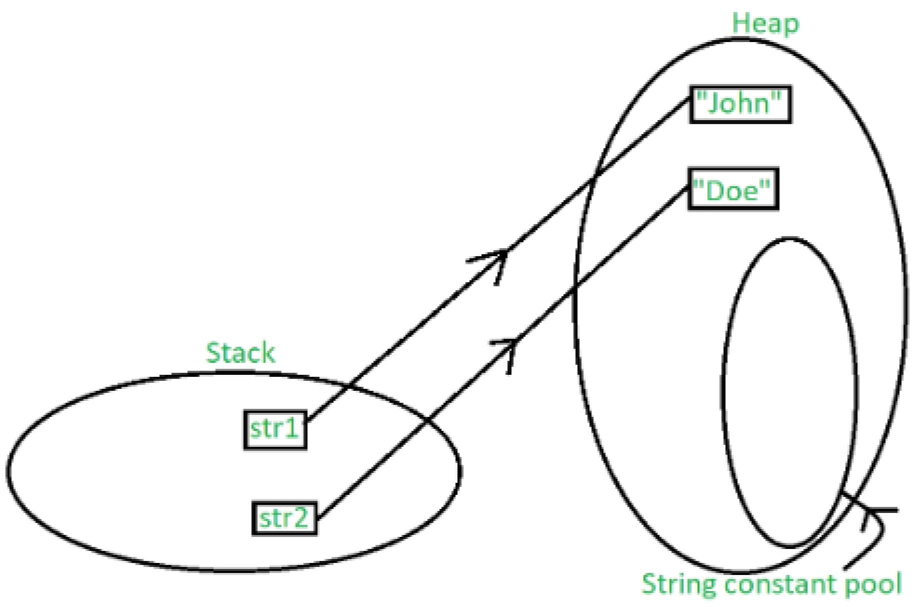
The following illustration explains the memory allocation for the above declaration:



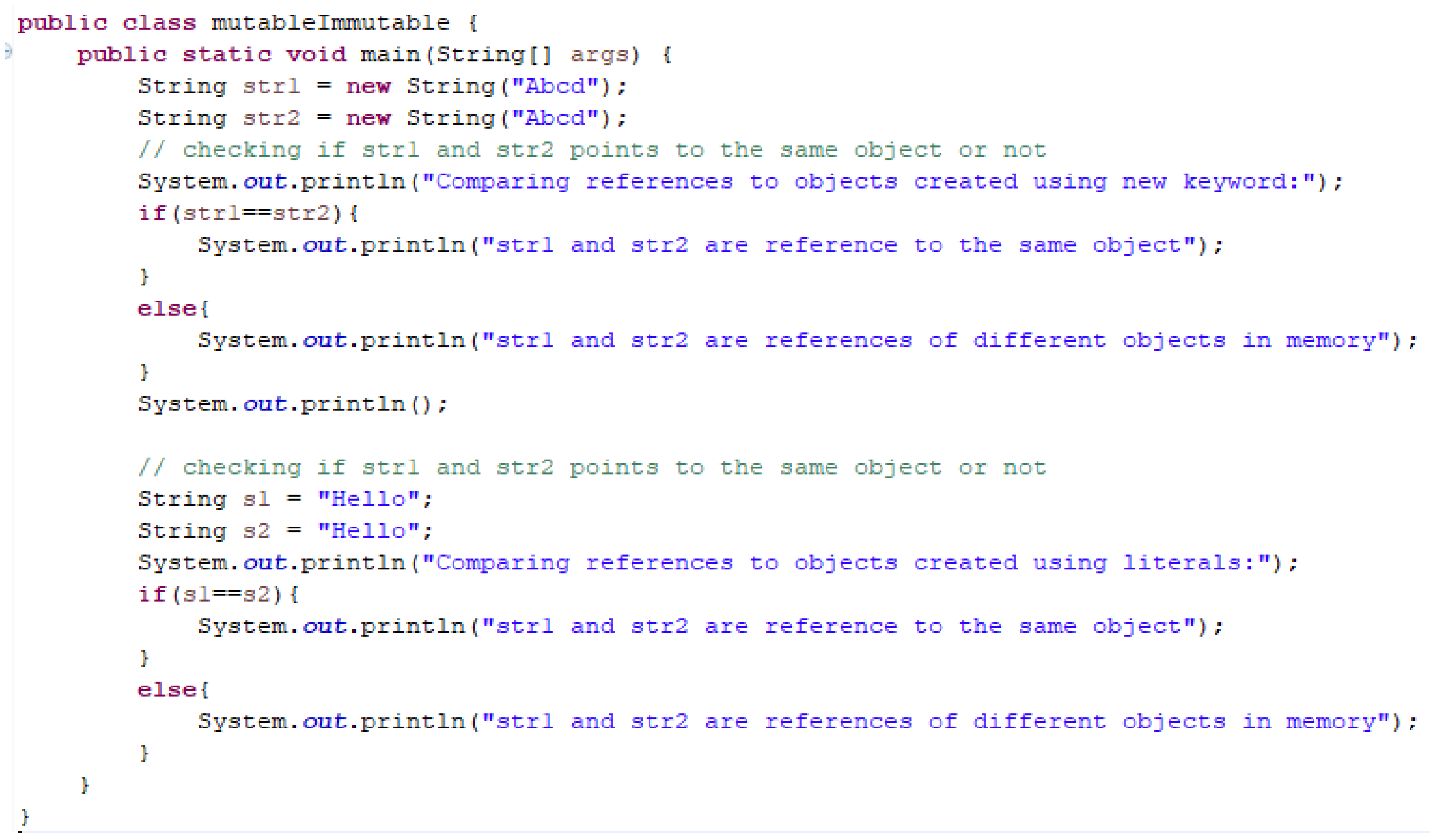
PLEASE NOTE:

The ‘new’ keyword forces a new instance to always be created regardless of whether the same value was used previously or not. Using ‘new’ forces the instance to be created in the heap outside the string constant pool

```
String str1 = new String("John");
String str2 = new String("Doe");
```

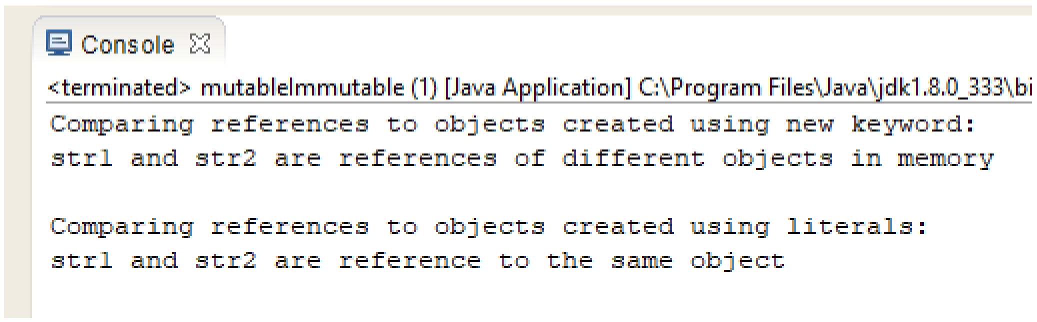


Program to check immutability of Strings in Java -



```java
public class mutableImmutable {
    public static void main(String[] args) {
        String str1 = new String("Abcd");
        String str2 = new String("Abcd");
        // checking if str1 and str2 points to the same object or not
        System.out.println("Comparing references to objects created using new keyword:");
        if(str1==str2){
            System.out.println("str1 and str2 are reference to the same object");
        }
        else{
            System.out.println("str1 and str2 are references of different objects in memory");
        }
        System.out.println();

        // checking if str1 and str2 points to the same object or not
        String s1 = "Hello";
        String s2 = "Hello";
        System.out.println("Comparing references to objects created using literals:");
        if(s1==s2){
            System.out.println("str1 and str2 are reference to the same object");
        }
        else{
            System.out.println("str1 and str2 are references of different objects in memory");
        }
    }
}
```



```
<terminated> mutableImmutable (1) [Java Application] C:\Program Files\Java\jdk1.8.0_333\bi
Comparing references to objects created using new keyword:
str1 and str2 are references of different objects in memory

Comparing references to objects created using literals:
str1 and str2 are reference to the same object
```

How to create an immutable class of your own

As per the definition of immutable class, it means any object once created of this class you can not change the content of that class.

These are some general guidelines to keep in mind while making an immutable class -

- Make all fields private and final

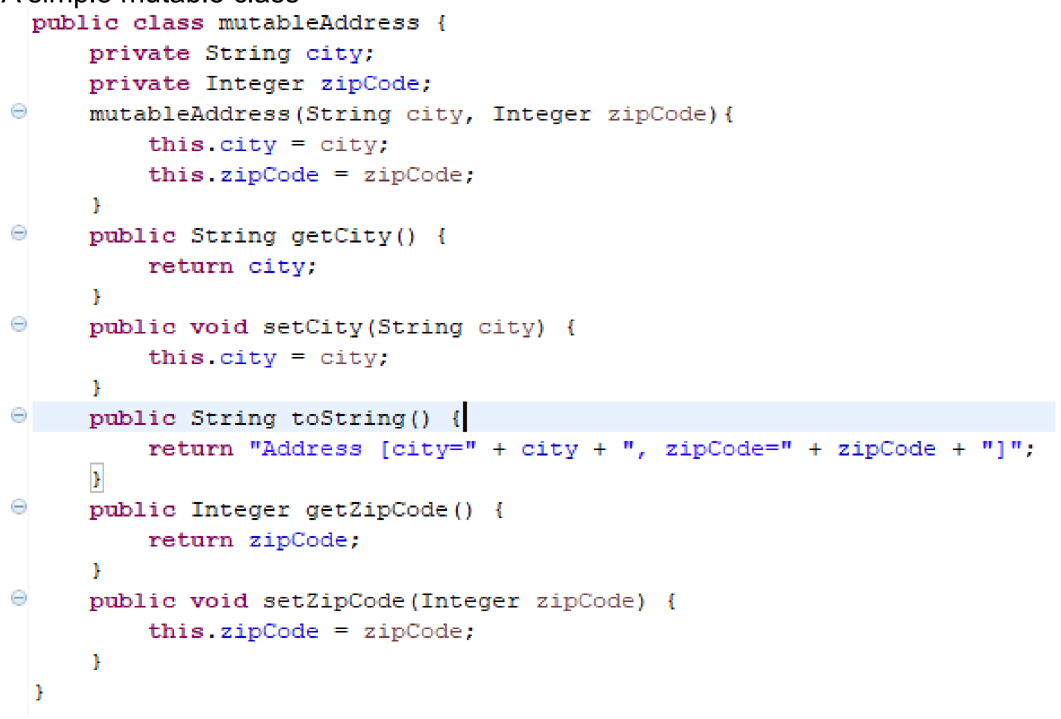
- Don’t provide setter methods that modify fields or objects referred to by fields

- Don’t allow subclasses to override methods. One simple way of doing this is by declaring the class as final as final classes in java cannot be extended.

- Special attention when having mutable instance variables -
  - Initialize the mutable fields in the constructor by performing deep copy.
  - Always return a copy of the reference by performing a deep copy, and never return the actual object reference.

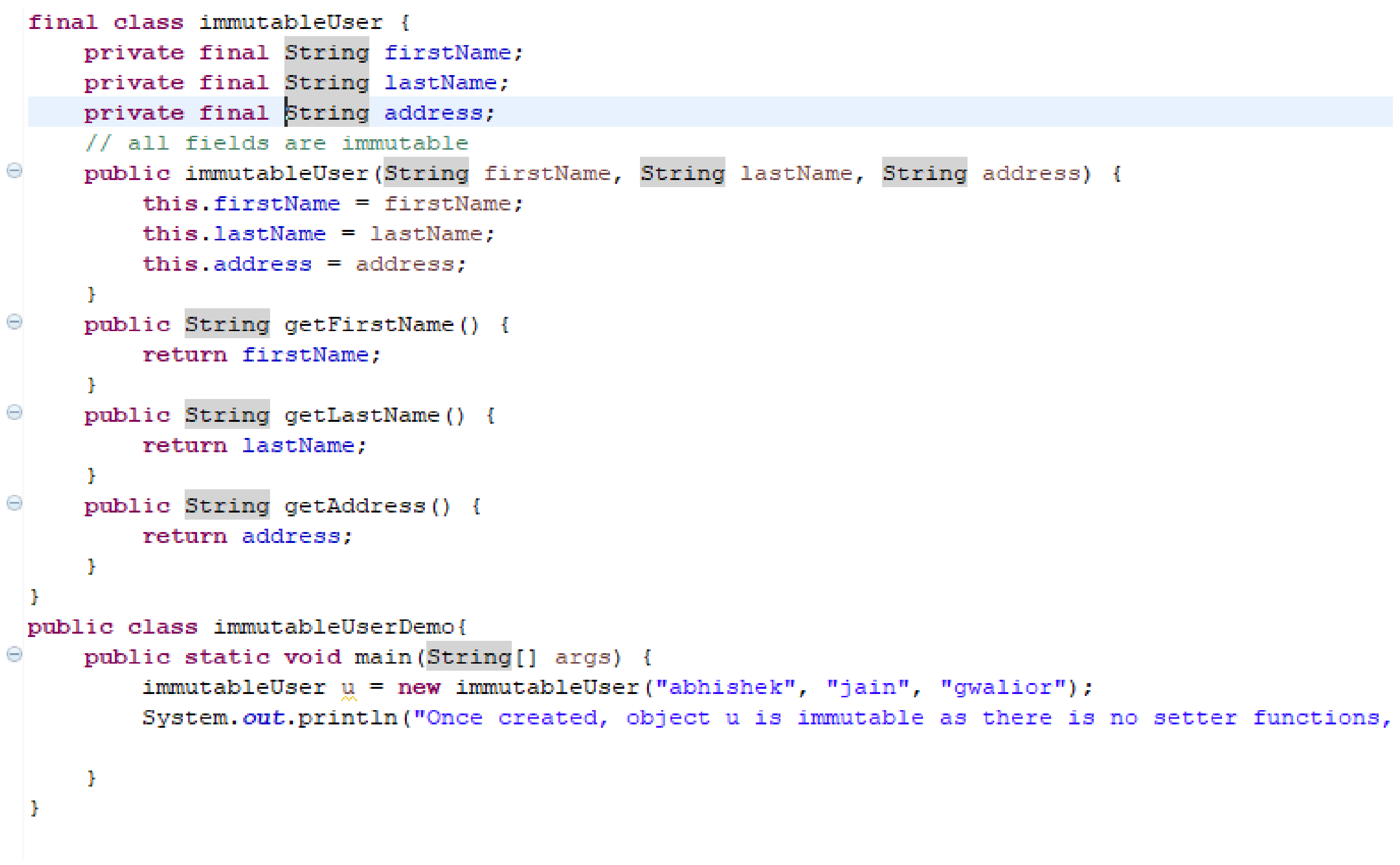
  Now let’s see some examples

  A simple mutable class -



```
public class mutableAddress {
    private String city;
    private Integer zipCode;
    mutableAddress(String city, Integer zipCode){
        this.city = city;
        this.zipCode = zipCode;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String toString() {
        return "Address [city=" + city + ", zipCode=" + zipCode + "]";
    }
    public Integer getZipCode() {
        return zipCode;
    }
    public void setZipCode(Integer zipCode) {
        this.zipCode = zipCode;
    }
}
```

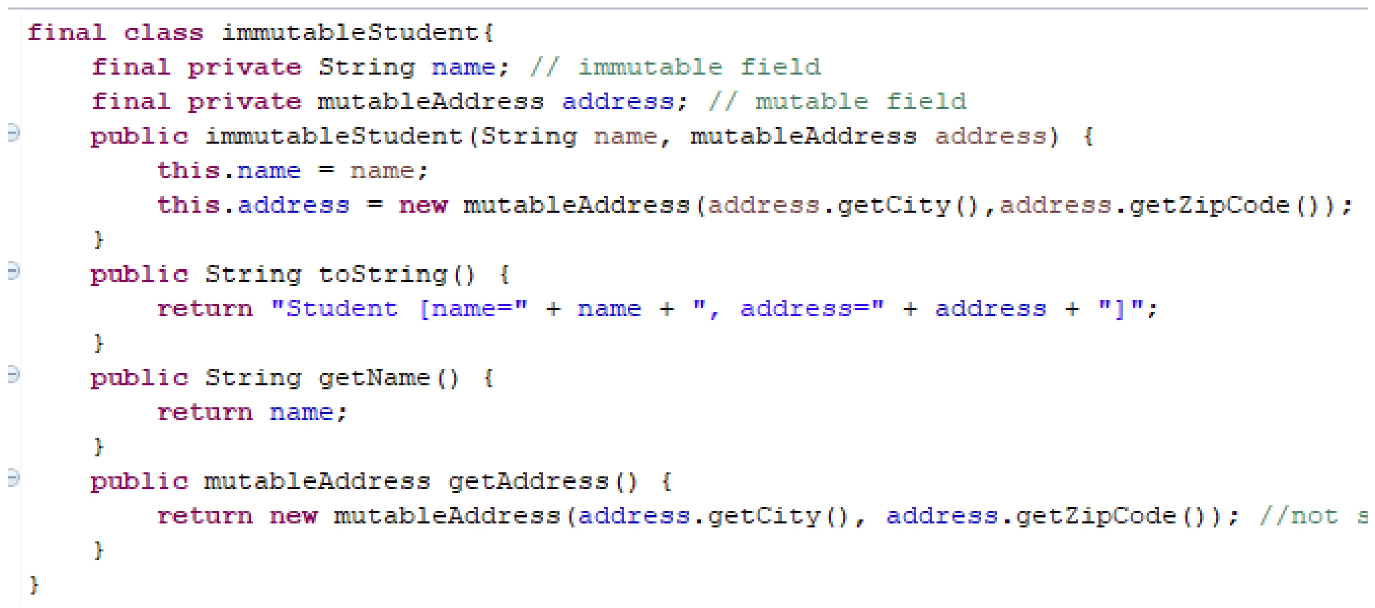
Next an immutable class with immutable fields -



```java
final class immutableUser {
    private final String firstName;
    private final String lastName;
    private final String address;
    // all fields are immutable
    public immutableUser(String firstName, String lastName, String address) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.address = address;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public String getAddress() {
        return address;
    }
}
public class immutableUserDemo{
    public static void main(String[] args) {
        immutableUser u = new immutableUser("abhishek", "jain", "gwalior");
        System.out.println("Once created, object u is immutable as there is no setter functions,

    }
}
```

Next an immutable class with mutable fields -



```java
final class immutableStudent{
    final private String name; // immutable field
    final private mutableAddress address; // mutable field
    public immutableStudent(String name, mutableAddress address) {
        this.name = name;
        this.address = new mutableAddress(address.getCity(),address.getZipCode());
    }
    public String toString() {
        return "Student [name=" + name + ", address=" + address + "]";
    }
    public String getName() {
        return name;
    }
    public mutableAddress getAddress() {
        return new mutableAddress(address.getCity(), address.getZipCode()); //not s
    }
}
```

References -

1) https://howtodoinjava.com/java/basics/how-to-make-a-java-class-immutable/
2) https://www.digitalocean.com/community/tutorials/how-to-create-immutable-class-in-java
3) https://codepumpkin.com/immutable-class-with-mutable-member-fields-in-java/
4) https://www.javatpoint.com/mutable-and-immutable-in-java#:~:text=The%20immutable%20objects%20are%20objects,immutable%20means%20unmodified%20or%20unchangeable.
5) https://www.geeksforgeeks.org/java-string-is-immutable-what-exactly-is-the-meaning/#:~:text=The%20String%20is%20safe%20for,the%20correct%20class%20by%20 Classloader