# Practical 7 – Mutability – Part 1

“Immutable objects are simple. They can only be in one state, which is carefully controlled by the constructor. One of the most difficult elements of program design is reasoning about the possible states of complex objects. Reasoning about the state of immutable objects, on the other hand, is trivial.   
  
Immutable objects are also safer. Passing a mutable object to untrusted code, or otherwise publishing it where untrusted code could find it, is dangerous — the untrusted code might modify its state, or, worse, retain a reference to it and modify its state later from another thread. On the other hand, immutable objects cannot be subverted in this manner by malicious or buggy code, so they are safe to share and publish freely without the need to make defensive copies.”

—**Brian Goetz,** Java Concurrency in Practice

## Introduction

An immutable object is one whose externally visible state cannot change after it is instantiated. The String, Integer, and BigDecimal classes in the Java class library are examples of immutable objects -- they represent a single value that cannot change over the lifetime of the object.

## Lab Instructions:

**Consider the following code:**

**A screen shot of a computer code

Description automatically generated**

Is this program mutable?

1. If you think the code is **mutable**, changes the code to make it immutable.
2. If you think it is already immutable, why?
3. Test your code to see: Create a **separate** tester class with a main method, create an instance of Course and see if you can manipulate the course details.
4. Did you notice anything about the Date object? It’s Deprecated! Do some research as to why and what should be used instead.

# Practical 7 – Mutability – Part 2

**Objective:**

To create a custom class and then develop tests to verify whether instances of this class are mutable or immutable.

**Instructions:**

You must implement a class in Java, then create a new class with a method to modify the instances of your initial class, and observe the behaviour to understand how the class is designed as mutable or immutable.

**Design Your Class**

* **Task 1:** Create a Java file named **MyCustomClassMutable**.**java**
* **Implementation Steps:**
  1. Design a class that holds several fields. Design this class to be **immutable**.
     + Give it an **int** called ‘value’ and a **string** called ‘name’
     + For an **immutable** class, ensure all fields are private and final, and provide only getter methods. Do not provide methods that modify the state.
* **Task 2:** Create a Java file named **MyCustomClassImmutable**.**java**
* **Implementation Steps:**
  1. Design a class that holds several fields. Design this class to be **mutable**.
     + Give it an **int** called ‘value’ and a **string** called ‘name’
     + For a **mutable** class, don’t make fields private and final, and provide setter and getter methods.

**Test Your Class**

* **Task 2:** Create a Java file named **TestMutability.java**
* **Implementation Steps:**
  1. Instantiate your **MyCustomClassMutable**
  2. Try to modify the instance through any setter methods or by changing its fields directly if accessible.
  3. Print the object's state before and after the modification attempts to verify if the object maintains its initial state or not.
  4. Instantiate your **MyCustomClassImmutable**
  5. Try to modify the instance through any setter methods or by changing its fields directly if accessible.
  6. Demonstrate that despite attempts to modify the original object, and that a new object must be created for any change.