# ADVANCED PROGRAMMING CONCEPTS

# USING JAVA

# (CSX-331)

ASSIGNMENT

COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DR. B R AMBEDKAR NATIONAL INSTITUTE OF TECHNOLOGY

JALANDHAR – 144011, PUNJAB (INDIA)

JULY-DECEMBER, 2017

| SUBMITTED TO: | SUBMITTED BY: |
|---|---|
| Dr. Paramveer Singh | Arjun Goyal |
| Asst. Professor | 15103026 |
| Department of CSE | G-1 |

# Assignment: 1

## Q1: How to make JAVA objects immutable and what are its advantages over normal objects?

**Ans:** The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    - Don't provide methods that modify the mutable objects.
    - 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.

Doing so will often restrict the way you can call the class and its methods. It will also make you redesign your software and rethink your algorithms. However, there are many benefits of programming with immutable objects.

1. Immutable objects are thread-safe so you will not have any synchronization issues.
2. Immutable objects are good **Map** keys and **Set** elements, since these typically do not change once created.
3. Immutability makes it easier to write, use and reason about the code (class invariant is established once and then unchanged)
4. Immutability makes it easier to parallelize your program as there are no conflicts among objects.
5. The internal state of your program will be consistent even if you have exceptions.
6. References to immutable objects can be cached as they are not going to change.

## Q2: Discuss the usage of jtree and jtable?

## Ans: Jtree:

The JTree class is used to display the tree structured data or hierarchical data. JTree is a complex component. It has a 'root node' at the top most which is a parent for all nodes in the tree. It inherits JComponent class.

### JTree class declaration

Let's see the declaration for javax.swing.JTree class.

```
public class JTree extends JComponent implements Scrollable, Accessible
```

**Commonly used Constructors:**

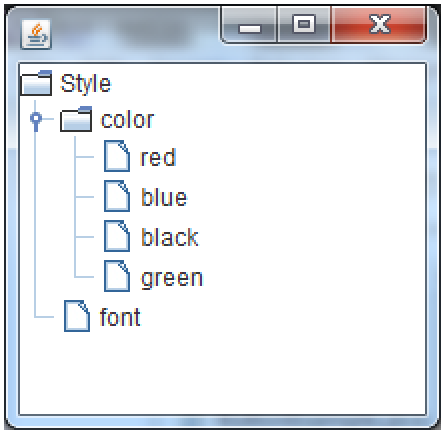
| Constructor | Description |
|---|---|
| JTree() | Creates a JTree with a sample model. |
| JTree(Object[] value) | Creates a JTree with every element of the specified array as the child of a new root node. |
| JTree(TreeNode root) | Creates a JTree with the specified TreeNode as its root, which displays the root node. |

---

Java JTree Example

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample {
JFrame f;
TreeExample(){
    f=new JFrame();
    DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
    DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
    DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");
    style.add(color);
    style.add(font);
    DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
    DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
    DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
    DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
    color.add(red); color.add(blue); color.add(black); color.add(green);
    JTree jt=new JTree(style);
    f.add(jt);
    f.setSize(200,200);
    f.setVisible(true);
}
public static void main(String[] args) {
    new TreeExample();
}}
```

Output:



## Jtable:

The JTable class is used to display data in tabular form. It is composed of rows and columns.

### JTable class declaration

Let's see the declaration for javax.swing.JTable class.

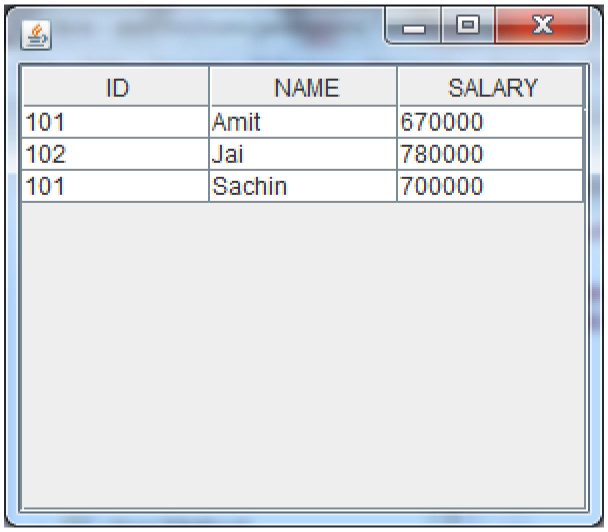
### Commonly used Constructors:

| Constructor | Description |
|---|---|
| JTable() | Creates a table with empty cells. |
| JTable(Object[][] rows, Object[] columns) | Creates a table with the specified data. |

Java JTable Example

```
import javax.swing.*;
public class TableExample {
    JFrame f;
    TableExample(){
    f=new JFrame();
    String data[][]={ {"101","Amit","670000"},
                          {"102","Jai","780000"},
                          {"101","Sachin","700000"}};
    String column[]={"ID","NAME","SALARY"};
    JTable jt=new JTable(data,column);
    jt.setBounds(30,40,200,300);
    JScrollPane sp=new JScrollPane(jt);
    f.add(sp);
    f.setSize(300,400);
    f.setVisible(true);
}
public static void main(String[] args) {
    new TableExample();
}
}
```

**Output:**



| ID | NAME | SALARY |
|---|---|---|
| 101 | Amit | 670000 |
| 102 | Jai | 780000 |
| 101 | Sachin | 700000 |

## Use of mutability in Jtree

A mutable tree node is one which can be altered. It is created using DefaultMutableTreenode.

A tree node may have at most one parent and 0 or more
children. DefaultMutableTreeNode provides operations for examining and modifying a node's parent and children and also operations for examining the tree that the node is a part of. A node's tree is the set of all nodes that can be reached by starting at the node and following all the

possible links to parents and children. A node with no parent is the root of its tree; a node with no children is a leaf. A tree may consist of many subtrees, each node acting as the root for its own subtree.

DefaultMutableTreeNode has three constructors:

```
public DefaultMutableTreeNode()


public DefaultMutableTreeNode(Object userObject)


public DefaultMutableTreeNode(Object userObject,   boolean allowsChildren)
```

The first constructor creates a node with no associated user object; you can associate one with the node later using the setUserObject method. The other two connect the node to the user object that you supply. The second constructor creates a node to which you can attach children, while the third can be used to specify that child nodes cannot be attached by supplying the third argument as false.
Using DefaultMutableTreeNode, you can create nodes for the root and for all of the data you want to represent in the tree, but how do you link them together? You could use
the insert method that we saw above, but it is simpler to use
the DefaultMutableTreeNode add method:

```
public void add(MutableTreeNode child);
```

This method adds the given node as a child of the node against which it is invoked and at the end of the parent's list of children. By using this method, you avoid having to keep track of how many children the parent has. This method, together with the constructors, gives us all you need to create a workable tree.

**Assignment: 2**

## Q1: How do you use the advance features of new javafx?

**Ans:** JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

JavaFX 2.2 and later releases have the following features:

- **Java APIs**. JavaFX is a Java library that consists of classes and interfaces that are written in native Java code. The APIs are designed to be a friendly alternative to Java Virtual Machine (Java VM) languages, such as JRuby and Scala.
- **FXML and Scene Builder**. FXML is an XML-based declarative markup language for constructing a JavaFX application user interface. A designer can code in FXML or use JavaFX Scene Builder to interactively design the graphical user interface (GUI). Scene Builder generates FXML markup that can be ported to an IDE where a developer can add the business logic.
- **WebView**. A web component that uses WebKitHTML technology to make it possible to embed web pages within a JavaFX application. JavaScript running in WebView can call Java APIs, and Java APIs can call JavaScript running in WebView.
- **Swing interoperability**. Existing Swing applications can be updated with new JavaFX features, such as rich graphics media playback and embedded Web content.
- **Built-in UI controls and CSS**. JavaFX provides all the major UI controls required to develop a full-featured application. Components can be skinned with standard Web technologies such as CSS
- **Canvas API**. The Canvas API enables drawing directly within an area of the JavaFX scene that consists of one graphical element (node).
- **Multitouch Support**. JavaFX provides support for multitouch operations, based on the capabilities of the underlying platform.
- **Hardware-accelerated graphics pipeline**. JavaFX graphics are based on the graphics rendering pipeline (Prism). JavaFX offers smooth graphics that render quickly through Prism when it is used with a supported graphics card or graphics processing unit (GPU). If a system does not feature one of the recommended GPUs supported by JavaFX, then Prism defaults to the Java 2D software stack.
- **High-performance media engine**. The media pipeline supports the playback of web multimedia content. It provides a stable, low-latency media framework that is based on the GStreamer multimedia framework.

- **Self-contained application deployment model**. Self-contained application packages have all of the application resources and a private copy of the Java and JavaFX runtimes. They are distributed as native installable packages and provide the same installation and launch experience as native applications for that operating system. See the Deploying JavaFX Applications document.

  **Example program**:

```
package helloworld;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

## Q2: Discus the usage and functionality of volatile and transient keyword?

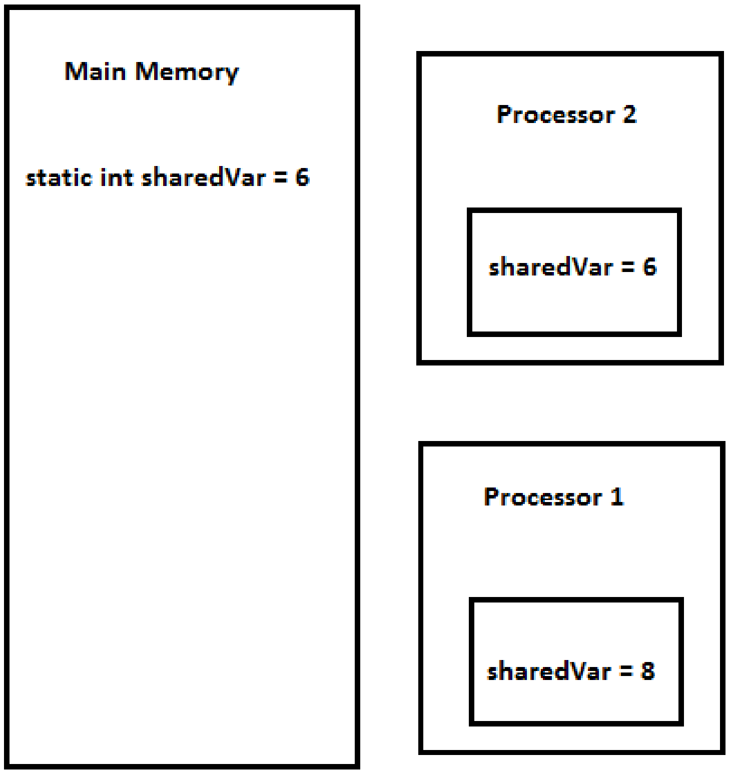
## Ans: Volatile:

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Consider below simple example.

```
class SharedObj
{
   // Changes made to sharedVar in one thread
   // may not immediately reflect in other thread
   static int sharedVar = 6;
}
```

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVar**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.
Below diagram shows that if two threads are run on different processors, then value of **sharedVar** may be different in different threads.



Note that write of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called sequential consistency). Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it’s not a good practice to rely on hardware for to ‘fix’ a faulty application.

```
class SharedObj

{

  // volatile keyword here makes sure that

  // the changes made in one thread are

  // immediately reflect in other thread

  static volatile int sharedVar = 6;
}
```

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

**Volatile vs synchronized:**
Before we move on let’s take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

## Transient:

**Transient** is a variables modifier used in serialization. At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient** keyword, it ignores original value of the variable and save default value of that variable data type.
**transient** keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file. Another use of **transient** keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.
Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use **transient** keyword with private confidential fields of a class during serialization.

```
// A sample class that uses transient keyword to
// skip their serialization.
class Test implements Serializable
{
   // Making password transient for security
   private transient String password;

   // Making age transient as age is auto-
```

```
    // computable from DOB and current date.
    transient int age;

    // serialize other fields
    private String username, email;
    Date dob;

    // other code
}
```

**transient and static :** Since **static** fields are not part of state of the object, there is no use/impact of using **transient** keyword with static variables. However there is no compilation error.
**transient and final :** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as **transient**. There is no compile-time error though.