1 User Interfaces

What's in This Set of Notes ?

The most dominant part of this course is related to designing and implementing complete applications that have user interfaces. In our previous course, our user interface was either keyboard input or test programs. In this set of notes we will look at some user interface terminology and learn how to keep our user interface separate from our model classes as well as examine a simple text-based user interface.

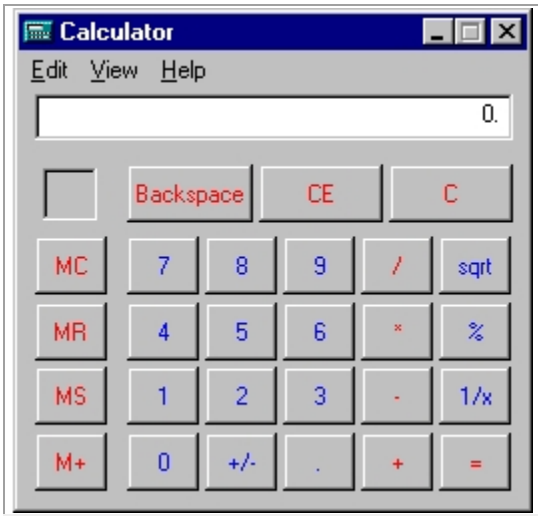
Here are the individual topics found in this set of notes (click on one to go there):

1.1 User Interface Terminology

All applications require some kind of ***user interface*** which allows the user to interact with the underlying program/software. Most user interfaces have the ability to take-in information from the user and also to provide visual or audible information back to the user. Sometimes the interface has physical/hardware interactive components (e.g., a bank machine or bank teller) and sometimes the components are software-based (e.g., menus, buttons, text fields).

a **Bank** has BankTellers as well as BankMachines, calculators have physical components:

a software application has buttons/text fields/menus:



an **On-line Shopping Cart** has webpages containing forms, text fields, lists, buttons:

In this course, we will consider software-based user interfaces (such as the software calculator shown above).

Some programs/applications do not really have any interactive user interface. For example, our **Team/League** example from COMP1405/1005 had a **main** method as a "pretend" user interface. We simply ran the code and it spewed output onto the console. We did not get to interact much at all with it, except to say "Go!" by running it.

What about simple text-based interfaces ? Recall the style of the basic text-based user interfaces that we have seen in COMP1405/1005:

```
public class AverageTest {
    public static void main(String args[]) {
        java.util.Scanner  keyboard = new
java.util.Scanner(System.in);
        int sum = 0.0;
        for (int i=0; i<5; i++) {
            System.out.println("Enter the number " + i + ":");
            sum += keyboard.nextInt();
        }
        System.out.print("The average is " + sum / 5.0);
    }
}
```

Here we received input from the keyboard, did a calculation, and then printed out the result. Later we wrote code that did not require any user input from the keyboard, but instead used simple test methods to simulate some "fixed scenario":

```
public static void main(String args[]) {
    League aLeague = new League("NHL");

    aLeague.addTeam(new Team("Ottawa Senators"));
    aLeague.addTeam(new Team("Montreal Canadians"));
    aLeague.addTeam(new Team("Toronto Maple Leafs"));
    aLeague.addTeam(new Team("Vancouver Cannucks"));

    aLeague.recordWinAndLoss("Ottawa Senators", "Toronto Maple Leafs");
    aLeague.recordWinAndLoss("Montreal Canadians", "Toronto Maple Leafs");
    aLeague.recordWinAndLoss("Ottawa Senators", "Vancouver Cannucks");
    aLeague.recordTie("Vancouver Cannucks", "Toronto Maple Leafs");
    aLeague.recordWinAndLoss("Toronto Maple Leafs", "Montreal Canadians");

    System.out.println("\nHere are the teams:");
    aLeague.showTeams();
}
```

Nevertheless, all of our programs so far have been ***Java Applications*** which:

- can be invoked from command line (e.g., **java MyApplication**), or with a button click when using **JCreator,**
- are always executed by the JAVA Interpreter (i.e., JVM), and
- must have **main()** method in order to run.

Something was fundamentally common between all our programs:

- they all had underlying classes representing objects that were specific to the application (e.g., the **Team** and **League** classes)
- the testing code (which was our "user interface") always made use of these underlying objects

As it turns out, these underlying objects altogether are called the ***Model*** of the application.

A ***Model***:

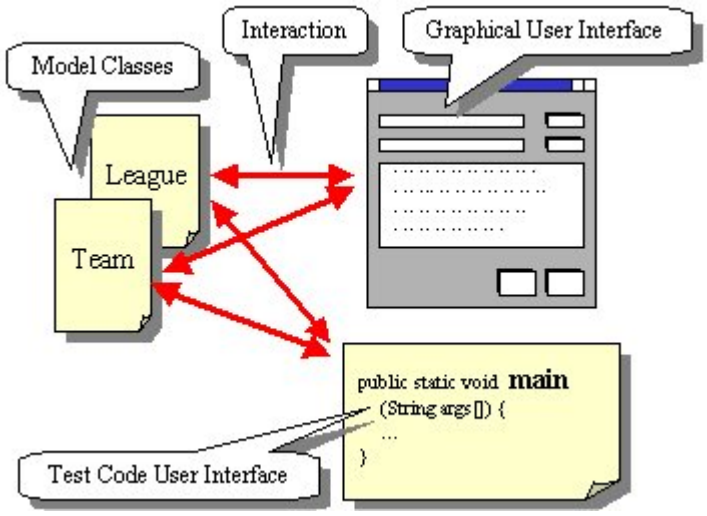
- may consist of one or more classes representing the *"business logic"* part of the application
- represents the underlying system on which a user interface is attached
- is developed separately from the user interface
- should not assume any knowledge about the user interface (e.g., may not assume that a **System.out** is available).

The ***User Interface***:

- is "attached to" a model, so we often develop the model first
- handles user interaction and does NOT deal with the business logic
- "uses" the model classes' methods
- often causes the model to change according to user interaction and these model changes are often reflected back on the user interface

A ***Graphical User Interface (GUI)***:

- is a user interface that has one or more windows (e.g., such as the calculator application shown above)
- is often preferred over text-based interfaces (more natural ... more like real-world)
    - imagine the internet as being only text-based.



We often split up the user interface as well into two separate pieces called the ***view*** and the ***controller***:

A ***View*** is:

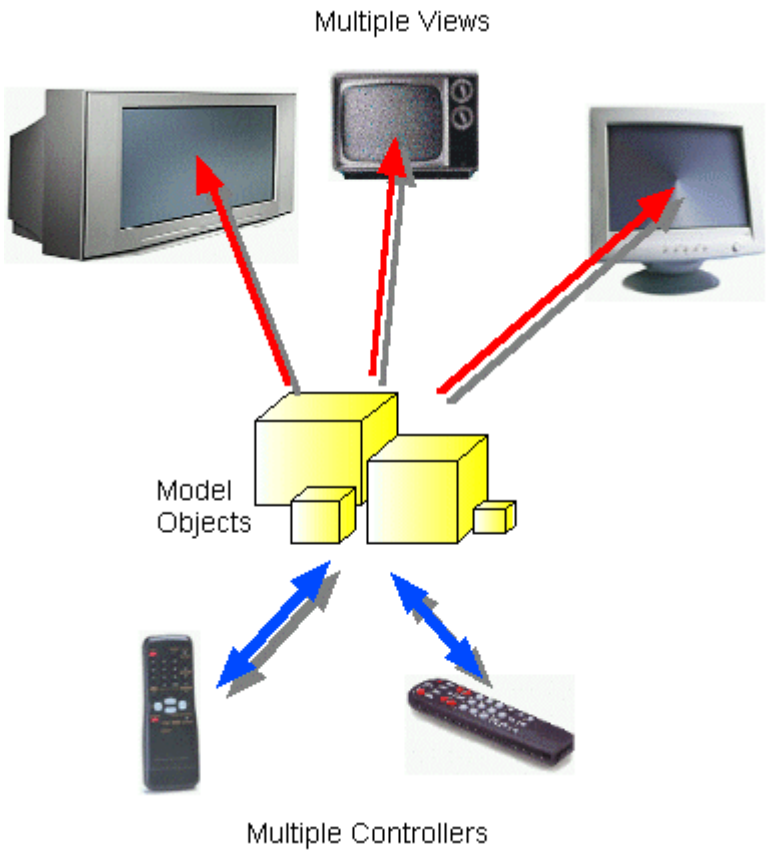
- the portion of the user interface code that specifies how the model is shown visually (i.e., its appearance).

A ***Controller*** is:

- the portion of the user interface code that specifies how the model interacts with the view (i.e., its behaviour).
- code that serves as a "mediator" between the model and the view.

It is ALWAYS a good idea to separate the model, view and controller (MVC):

- code is cleaner and easier to follow when the view and controller are separated
- we may want to have multiple views or controllers on the same model.
- we may want to have multiple models for the same user interface.



1.2 A Simple Text-Based User Interface

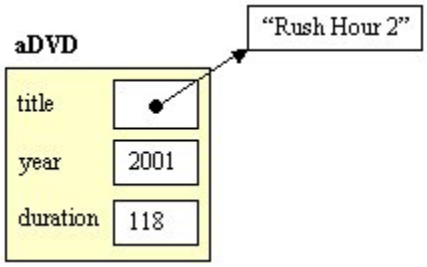
Let us now look at an example of how to separate our model from our user interface. We will create an application that allows us to maintain a small database to store the DVDs that we own. We must think of what we want to be able to do with the application. These are the ***requirements***:

- We want to keep the DVD's title, year and duration (i.e., how long the DVD lasts) when played (e.g., 120 minutes).
- We should be able to **add** new DVDs to our collection as well as **remove** them (if we lose or sell them).
- We should be able to **list** our DVDs sorted by title.

So, that's it. It will be a simple application. Let us begin by creating the model, since we did this already a few times in COMP1405/1005.
What objects do we need to define ?

- a DVD object (maintains title, year, duration)
- an object to represent the collection of DVDs (similar to our **League** class in COMP1405/1005).

Here is a simple DVD class:



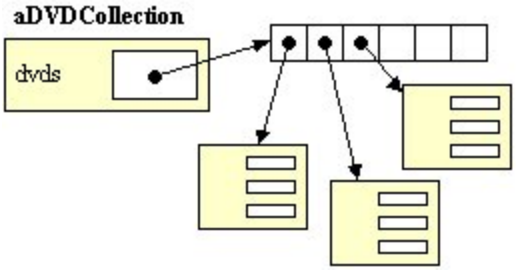

```
public class DVD {
    private   String title;
    private   int year;
    private   int duration;

    public DVD () { this("", 0, 0); }
    public DVD (String newTitle, int y, int minutes) {
        title = newTitle;
        year = y;
        duration = minutes;
    }

    public String getTitle() { return title; }
    public int getDuration() { return duration; }
    public int getYear() { return year; }
    public void setTitle(String t) { title = t; }
    public void setDuration(int d) { duration = d; }
    public void setYear(int y) { year = y; }

    public String toString() {
        return ("DVD (" + year + "): " + title + " with length: "
+ duration);
    }
}
```

There is not much to this class. Now what about the DVD collection itself ? It should probably look something like this in its most basic form:




```
import java.util.*;
public class DVDCollection {
    private ArrayList<DVD>   dvds;

    public DVDCollection() { dvds = new ArrayList<DVD>(); }
```

```
    public ArrayList<DVD> getDvds() { return dvds; }
    public String toString() { return ("DVD Collection of size "
+ dvds.size()); }

    public void add(DVD aDvd) { dvds.add(aDvd); }
    public boolean remove(String title) {
        for (DVD aDVD: dvds) {
            if (aDVD.getTitle().equals(title)) {
                dvds.remove(aDVD);
                return true;
            }
        }
        return false;
    }
}
```

Notice when adding a DVD to the collection, we simply add it to the ArrayList. When removing, it is more convenient to have a method that removes according to title. The **remove()** method therefore takes a string and searches for the DVD with that title.

OK. So now we can test the adding/removing and listing with a **main()** method as follows:

```
public static void main (String[] args) {
    DVDCollection c = new DVDCollection();
    c.add(new DVD("Star Wars", 1978, 124));
    c.add(new DVD("Java is cool", 2002, 93));
    c.add(new DVD("Mary Poppins", 1968, 126));
    c.add(new DVD("The Green Mile", 1999, 148));
    c.remove("Mary Poppins");
    // List the DVDs
    for (DVD aDVD: c.getDvds())
        System.out.println(aDVD);
}
```

Let us now make a user interface for this model. We should make a new class for this. We will make a simple text-based interface. It should perhaps bring up a menu and repeatedly prompt for a user choice. Here is what we will make appear on the screen:

```
Welcome to the Dvd Collection User Interface
--------------------------------------------
1.  Add DVD
2.  Delete DVD
3.  List DVDs
4.  Exit

Please make a selection:
```

If the user makes an invalid selection, we print an error message out. This is considered to be our main menu. Once we make a selection and the action has been performed, this menu will be displayed again. Here is the user interface process:

1. Display a menu with choices
2. Wait for the user to make a selection

3. Perform what needs to be done from that selection
    1. possibly more input is required
    2. the user may want to quit the program
    3. computations may be made, output may be displayed
4. Go back up to step 1.

We will design our code so that the main user interface holds onto a DVD collection as its model:

```
public class DVDUI {
    private   DVDCollection   model;

    public DVDUI() { model = new DVDCollection(); }
    public DVDCollection getModel() { return model; }

    public void showMainMenu() {
        System.out.println("1.  Add DVD");
        System.out.println("2.  Delete DVD");
        System.out.println("3.  List DVDs");
        System.out.println("4.  Exit");
        System.out.println("\nPlease make a selection");
    }

    public static void main (String[] args) {
        System.out.println("Welcome to the Dvd Collection User Interface");
        System.out.println("-----------------------------------------");

        new DVDUI().showMainMenu();
    }
}
```

Notice that the instance variable is named **model** which holds onto the **DVDCollection** which the interface is attached to. We could have picked any name for this variable, but we chose **model** here to help you understand that the rest of the code represents the user interface.

The model is now "plugged-into" the user interface through this instance variable. However, we will want to make use of the model's methods. Let us now handle the keyboard input and make the main menu repeat until 4 is entered:

```
    public void showMainMenu() {
        while(true) {
            System.out.println("1.  Add DVD");
            System.out.println("2.  Delete DVD");
            System.out.println("3.  List DVDs");
            System.out.println("4.  Exit");
            System.out.println("\nPlease make a selection");
            int   selection = new java.util.Scanner(System.in).nextInt();
            switch(selection) {
                case 1:   /* Handle the adding of DVDs   */ break;
                case 2:   /* Handle the removing of DVDs */ break;
                case 3:   /* Handle the listing of DVDs  */ break;
                case 4:   System.exit(0);
                default:  System.out.println("Invalid Selection");
```

```
        }
    }
}
```

If we run our code now, we will notice that it repeatedly brings up the main menu, waits for a response from the user and then displays the menu again. This repeats until the user selects option 4 to quit the program.

So we now have:

- The **Model**: which is the DVD collection class and the DVD class
- The **View**: which is the menu system

We now need to create the **Controller** part of the program which is responsible for linking the model to the view through proper interaction. The controller decides what happens to the model when the user interacts with the program through the view. So we just need to handle the menu choices. We can create helper methods so that the main method stays simple. We can call these helper methods from the switch statement:

```
switch(selection) {
   case 1:   addDVD();    break;
   case 2:   deleteDVD(); break;
   case 3:   listDVDs();  break;
   case 4:   System.exit(0);
   default:  System.out.println("Invalid Selection");
}
```

We must now decide what needs to be done in each of the helper methods:

- The **addDVD()** method should:
  - make a new DVD and add it to the collection (i.e., to the model)
  - allow the user to enter the title, year and duration of the DVD

```
private void addDVD() {
     DVD    aDVD = new DVD();
     System.out.println("Enter DVD Title:  ");
     aDVD.setTitle(new java.util.Scanner(System.in).nextLine());
     System.out.println("Enter DVD Year (e.g., 2001):");
     aDVD.setYear(new java.util.Scanner(System.in).nextInt());
     System.out.println("Enter DVD Duration (minutes):");
     aDVD.setDuration(new java.util.Scanner(System.in).nextInt());
     model.add(aDVD);
}
```

- The **deleteDVD()** method should:
  - prompt the user for the name of the DVD to remove
  - remove the DVD from the collection (i.e., from the model)

```
private void deleteDVD() {
    System.out.println("Enter DVD Title:  ");
```

```
            model.remove(new java.util.Scanner(System.in).nextLine());
        }
```

- The **listDVDs()** method should:
  - list all the DVDs in the collection (i.e., model) to the console.

```
        private void listDVDs() {
            for (DVD aDVD: model.getDvds())
                System.out.println(aDVD);
        }
```

The above code for listing DVDs displays them in the order that they were added. What about having them displayed sorted by title ? To do this, we can make use of the sort method in the Collections class:

```
    private void listDVDs() {
        java.util.Collections.sort(model.getDvds());
        for (DVD aDVD: model.getDvds())
            System.out.println(aDVD);
    }
```

Of course, this implies that our DVDs implement the **Comparable** interface ... which means that they have to have a **compareTo()** method. We will need to make the following changes to our **DVD** class definition:

```
public class DVD implements Comparable {

    ...
    public int compareTo(Object obj) {
        if (obj instanceof DVD) {
            DVD aDVD = (DVD)obj;
            return title.compareTo(aDVD.title);
        }
       return 0;
    }
    ...
}
```

So sorting by title merely compares the string titles using the **String** class's **compareTo()** method.

Click here for the combined code of everything we have so far.

We can make finishing touches on the application by:

- ensuring that the output displayed on the screen is pleasant on the eyes (e.g., properly tabbed)

- ensuring that our interface handles all likely errors gracefully (e.g., deleting a DVD that doesn't exist)

The BEST person to try out a user interface is a small child !! They'll be sure to find problems that you never imagined could occur. For example, our interface currently crashes when we enter a non-integer menu choice, dvd year or dvd duration. We can fix this by catching the **InputMismatchExceptions** that may occur. We can modify the while loop in our main menu method as follows:

```
while(true) {
     System.out.println("1.  Add DVD");
    System.out.println("2.  Delete DVD");
    System.out.println("3.  List DVDs");
    System.out.println("4.  Exit");
    System.out.println("\nPlease make a selection");
     try {
         int selection = new
java.util.Scanner(System.in).nextInt();
         switch(selection) {
             case 1:   addDVD();    break;
             case 2:   deleteDVD(); break;
             case 3:   listDVDs();  break;
             case 4:   System.exit(0);
             default:  System.out.println("Invalid Selection");
         }
     }
     catch(java.util.InputMismatchException e) {
         System.out.println("Invalid Selection");
     }
     System.out.println("\n");
}
```

We will also want to change our **addDVD()** method similarily:

```
private void addDVD() {
     DVD    aDVD = new DVD();
     System.out.println("Enter DVD Title:  ");
     aDVD.setTitle(new
java.util.Scanner(System.in).nextLine());
     int entered = -1;
     do {
         System.out.println("Enter DVD Year (e.g., 2001):");
         try {
             entered = new
java.util.Scanner(System.in).nextInt();
         }
         catch (java.util.InputMismatchException e) {
             System.out.println("Invalid Year");
```

```
            }
        }
        while (entered == -1);
        aDVD.setYear(entered);

        entered = -1;
        do {
            System.out.println("Enter DVD Duration (minutes):");
            try {
                entered = new
    java.util.Scanner(System.in).nextInt();
            }
            catch (java.util.InputMismatchException e) {
                System.out.println("Invalid Duration");
            }
        }
        while (entered == -1);
        aDVD.setDuration(entered);
        model.add(aDVD);
    }
```

We should also display a nice message to provide feedback to the user upon deletion:

```
    private void deleteDVD() {
        System.out.println("Enter DVD Title:  ");
        String title = new java.util.Scanner(System.in).nextLine();
        boolean success = model.remove(title);
        if (success)
            System.out.println("\nDVD: " + title + " was deleted successfully
\n");
        else
            System.out.println("\n*** Error: Could not find DVD: " + title +
"\n");
    }
```

I am sure you can think of many ways to improve this code. Here are the important things to remember from this code:

- The model was created separately from the user interface
- The user interface was created with a particular model in mind (in this case a DVDCollection)
- The model was "tied to" the user interface through an instance variable in the user interface class
- The user interface accesses and modifies the model by calling **public** methods in the model classes

---