Framework : set of classes and interfaces that provide the framework for building an application
- Software developer \textbf{reuses} or \textbf{customizes} parts of the framework instead of developing everything from scratch
- Framework handles \textbf{control flow} (complex in GUI)
- Ex. Java provides framework for developing GUIs: Abstract Window Toolkit (AWT) & Swing & Java Event Model

\textbf{Customization by inheritance} (creating subclass): developer must understand the inheritance hierarchy of the class that he is extending

\textbf{Customization by composition} (implementing interface): no need to understand rest of framework

Benefits of using framework:
- Application using framework may be more \textbf{reliable} because framework has been tested
- \textbf{Less expertise and time} (compared to building from scratch)
- \textbf{Consistent "look-and-feel"} to interface
Disadvantages of using framework:
- Effort to learn framework
- Give up some control over design: e.g. control flow, look and feel
- May not even be able to customize framework to meet requirements of application
- Forced to change application to keep up with new versions of framework

Java version 1.1 AWT: provides GUI components
- Component: visual object that responds to user action
- Ex.: buttons, panels, dialog boxes, menus, text fields, lists
- heavyweight components: implemented by local platform's native GUI system
  - E.g. same button may look different when program runs under Windows & under Macintosh
  - Look-and-feel consistent with OS user is used to
  - Less portable: behavior may vary across platforms

Java version 1.2 (Java 2) Swing components
- more types of components than AWT
- lightweight components: not depend on platform
  - consistent look-and-feel across platforms
  - can be chosen at runtime
- can only be run by Java 1.2 interpreters (not 100% available at UNCG yet)
Benefits of using Java for GUI development:

1. Object-oriented **GUI framework**

2. **Robustness**: compile-time and run-time error checking (exception handling), automatic garbage collection, no explicit pointers

3. Features supporting Internet applications
   - Concurrent programming: perform more than one task at a time (e.g. send data over network while update screen)
   - Portability: compile once, run on any machine that has Java Virtual Machine (interpreter)
   - Security: programs running on user’s computer via web browser cannot access files on user’s computer
   - Programs can be loaded over network when needed

Disadvantages of using Java for GUI development:
- Not rapid prototyping language - not WYSIWYG
- Designed for programmers
CSC540 - MVC Architecture

**Model**: implements data and operations that it is the goal of the program to manipulate
- Sometimes called the "backend application"
- Example: class model of bank application has account information and methods for withdrawing and depositing

**View(s)**: implements visual display(s) of model
- A program may present user with different views of the model at the same time
- Sometimes called the "presentation module(s)"
- Example: frame showing user's account info as checkbook, another frame showing as bank statement
- Whenever the model changes, the views must be notified by the model so they can be updated
  - Region of screen that no longer consistent with model is called **damaged**

**Controller**: implements interaction with user
- Translates input events from user (e.g. mouse click) into messages to model
- Sometimes called the "translation module"
- May communicate with view to get more information about event

MVC architecture
Java example of MVC organization of chess playing program (UML class diagram):

Model: code to keep track of state of game and generate opponent's moves
Controller: code that responds to user input (ex. moving virtual pawn by mouse dragging action)
View: code to display current state of game as chessboard

MVC Architecture is used to organize nontrivial interactive applications because
- **Good to separate implementation of model (backend) from implementation of view/controller (GUI) for several reasons**
  - Might want new interface to backend sometime
  - Might want more than one view or controller
    - ex. different way of showing chessboard
    - ex. user can choose commands or mouse dragging to make moves
  - Might want to simultaneously maintain two versions of the model for user,
    - e.g. keep track of old and new versions of game (would require separate view for each too)
  - Might want to provide multiple views of same model to user at the same time
• **Good to separate implementation of view from controller** because
  • may want to change behavior (controller) but not appearance (view)
  • may want to change appearance but not behavior

**Java example of MVC organization with multiple views**

![Diagram of MVC organization]

Model: keeps track of state of bank accounts
Controller: handles user's actions on bank account
2 Account Views of same model:
- AccountCheckRegister shows each deposit/withdrawal in chronological order
- AccountStatement shows deposits/withdrawals divided up into 2 sub-areas:
  - withdrawal by check: in order of check number
  - deposit and withdrawal by ATM card: in chronological order
CSC540 Using Observer Pattern to implement Model-View-Controller in Java

Observer Design Pattern

- Invented so that when something happens to an object of an **Observable** class, the object can notify any **observers** that it has changed
  
  - Observers must register themselves

  - Observable just sends one notifyObservers message when it changes

  - All registered Observers are notified by Java

  - Observable does not have to know what objects are observing it

- Useful for implementing **Model-View** of MVC architecture

  - **Supported in Java** through its **Observer Interface** and **Observable** class

  - **Model**: an **Observable** subclass
    - Does not need to know what Views are observing it

  - **Views**: **Observers**
    - Each view that wants to be notified registers itself
    - Observer views notified when Model changes
    - Each observer view can then update itself in its own way
**Initialization phase**

1. The Model is defined as an Observable

   - Syntax example:
     ```java
     public class DemoModel
     extends Observable {
     ```

2. The View is defined as implementing the Observer interface:

   - Syntax example:
     ```java
     public class DemoScreen
     extends Frame
     implements Observer {
     ```

3. The View **registers** as an Observer object with the Observable Model

   - Syntax example:
     ```java
     dModel.addObserver(this);
     // ask dModel to register this view
     ```
Update-handling phase

1. The model is updated
   • For example, the model gets a message from the controller to update itself

2. The model sends Java a notifyObservers message

3. Java calls the update method of each registered Observer
   • Method interface defined in Observer interface:
     public void update( java.util.Observable model, java.lang.Object screen)
   • body is implemented by each Observer

Example Java program using MVC design

import java.util.Observable;
import java.awt.*;

// DemoModelF02 is a class whose purpose is to implement the Model.
// It is a subclass Observable to notify the Views of changes.

class DemoModelF02 extends Observable {

    private int sum = 0;    // state of the model

    public void incDemoSum (int inc) {
        sum = sum + inc;
        // Next 2 calls to Observable's methods will notify
        // Observers (Views) that there was a change in Model:
        setChanged();
        notifyObservers();
    }

    public int getSum () {
        return sum;
    }
}
// end of class
import java.awt.*;
import java.awt.event.*;

public class DemoMainF02 implements ActionListener {

// DemoMainF02 is the main control and handles events for DemoScreenF02.

// instance fields:
private DemoModelF02 m;
private DemoScreenF02 f1;
private DemoScreenF02 f2;

// instance method:
public void initDemo() {
    m  = new DemoModelF02();
    // create two views:
    //    arguments are: this Listener object, model m, and screen 1 or 2
    f1 = new DemoScreenF02(this, m, 1);
    f2 = new DemoScreenF02(this, m, 2);
    f1.setSize(250,250 );
    f1.setVisible(true);
    f2.setSize(250,250 );
    f2.setVisible(true);
} // end initDemo()

public static void main(String[] args){
    DemoMainF02 theListener = new DemoMainF02();
    theListener.initDemo();
} // end main

public void actionPerformed (ActionEvent e) {
    String action = e.getActionCommand();
    if (action.equals("panic")) {
        System.out.println("Do not panic"); // echo to console
        System.exit(0);
    }
    else if (action.equals("increment")) {
        m.incDemoSum(1); // notify Model
    }
} // end actionPerformed
}
import java.awt.*;
import java.awt.event.*;
import java.util.Observer;

public class DemoScreenF02 extends Frame implements Observer {
    // DemoScreen implements Observer so that it can be notified
    // by DemoModel when it changes

    // instance variables for screen
    private Button panic = new Button("Panic");
    private Button add_1 = new Button("Add 1");
    private Label sum_display = new Label("Sum: 0", Label.CENTER);

    // instance variable to hold pointer to model:
    private DemoModelF02 dModel;

    // instance methods
    public DemoScreenF02( DemoMainF02 dl, DemoModelF02 m, int variety) {
        dModel = m; // set pointer to model
        dModel.addObserver(this); // register this view to be notified

        // define screen:
        if (variety == 1) {
            setBackground(Color.white);
        } else {
            setBackground(Color.cyan);
        }

        setTitle("This demo is brought to you by Nancy");
        setLayout(new BorderLayout());
        // add(BorderLayout.NORTH,logo);
        add(BorderLayout.WEST,panic);
        add(BorderLayout.EAST,add_1);
        add(BorderLayout.CENTER,sum_display);

        // initialize event handling:
        panic.addActionListener( dl );
        add_1.addActionListener( dl );
        panic.setActionCommand("panic");
        add_1.setActionCommand("increment");
    }

    // method required by Observer interface:
    public void update( java.util.Observable model, java.lang.Object
    screen) {
        updateDemoSum(dModel.getSum());
    }

    // instance method to update the View with new_sum
    public void updateDemoSum (int new_sum) {
        sum_display.setText("Sum: " + new_sum);
    }
}
Messages:
1. user pushes add_1 button of DemoScreen
2. actionPerformed()
3. incDemoSum(1)
4. notifyObservers()
5. update()
6. getSum()
7. updateDemoSum(..)