Computer Science 212: Object-Oriented Programming in Java

Darshan Patel

Fall 2016

Contents

1 Introduction to Java

- Java code can be written once and run anywhere
- Java source code goes to a compiler for a machine and made into source code which can be run on any machine
- API: application program interface- class libraries (MATH, GUI, Database...)
- JRI: Java Runtime Environment: JVM plus API (can run Java programs)
- JDK: Java Development Kit: JRE plus compiler, javadoc, ...
- Example: Java source code is written and stored in a .java file, compiler then calls "javac name.java" making a class file "name.class" which can then be run "run name"
- The Java virtual machine ensures a program will run the same way on any computer
- When running on native hardware, you are dependent on the machine's architecture
- The JVM levels the playing field

2 Classes and Objects

- Primitives: single-valued data items, not objects
  - byte: 8 bit signed two's complement integer
  - short: 16 bit signed two's complement integer
  - int: 32 bit signed two's complement integer
  - long: 64 bit signed two's complement integer
  - float: 32 bit single precision IEEE 754
  - double: 64 bit single precision IEEE 754
  - boolean: true/false
  - char: 16 bit Unicode character
- The number of bits and the range of values:

$$n = 32$$

$$-(2^{n-1}) \dots (2^{n-1}) + 1$$

$$-2^{31} \dots 2^{31} - 1$$

$$-2,147,483,648 \dots 2,147,483,647$$

- ASCII: 8 bit char code
$$2^8 = 256 \text{ char}$$
- Unicode: 16 char code
$$2^{16} = 2^6 \times 2^{10} = 64,000 \text{ char}$$
- Class: a blueprint or template that describes the properties and behaviors of an object
- Object: an instance of a class which has specific properties
- Instantiation: making an instance of a class
- Static variables belong to a class - only one variable for every instance of the class
- Instance variables belong to an object - the object is an instance of the class
- Static variables can be changed
- Use the final modifier to make constants
- Methods: defines the behavior of an object

3 Arrays and Sorting

-
```
private static int inputFromFile(String filename, short[] numbers){
      TextFileInput in = new TextFileInput(filename);
      int lengthFilled = 0;
      String line = in.readLine();
      while ( lengthFilled < numbers.length && line != null ) {
         numbers[lengthFilled++] = Short.parseShort(line);
         line = in.readLine();
      } // while
      if ( line != null ) {
         System.out.println("File contains too many numbers.");
         System.out.println("This program can process only " +
                            numbers.length + " numbers.");
         System.exit(1);
      } // if
      in.close();
      return lengthFilled;
   }  // method inputFromFile


   private static void selectionSort (short[] array, int length) {
      for ( int i = 0; i < length - 1; i++ ) {
         int indexLowest = i;
         for ( int j = i + 1; j < length; j++ )
            if ( array[j] < array[indexLowest] )
```

```
            indexLowest = j;
        if ( array[indexLowest] != array[i] ) {
           short temp = array[indexLowest];
           array[indexLowest] = array[i];
           array[i] = temp;
        } // if
      } // for i
   } // method selectionSort
```

4 Methods and Parameter Passing

- Primitive type parameters are passed by value
- Pass by value means a copy of the value of the parameter is given to the method
- Each variable in the main program and the method has its own memory location
- Object type parameters are passed by reference
- Pass by reference means a reference to the parameter is given to the method
- Each variable in the main program and the method has its own memory location for the reference
- Since we have a reference to an object in a method, any changes made to that object are permanent
- Changes happen to the object whose reference is used in a method call
-

```
private static void selectionSort
                       (short[] array, int length) {
for ( int i = 0; i < length - 1; i++ ) {
  int indexLowest = i;
  for ( int j = i + 1; j < length; j++ )
    if ( array[j] < array[indexLowest] )
       indexLowest = j;
       if ( array[indexLowest] < array[i] ) {
         swap( array,indexLowest,i);
       }
   } // for i
} // method selectionSort
```

```
public static void swap
               (int[] a, int x, int y) {
  int temp;
  temp = a[x];
```

```
  a[x]=a[y];
  a[y]=temp;
}
```

5 Program Modularity and Error Checking

- Program modularity: break a program down into smaller parts (methods), test each method separately, understand the relationship between the methods: parameters and their expected values
- Date validation: data entered by a use (at a prompt), data entered by a user (into a file), data values received from other methods
- A method should verify what it receives and returns
- System.exit(0) causes the program to end
- "Run-time" error messages can provide useful information to the programmer or to the user
- Errors could be due to bad code or bad data
- Bad code and testing: test the program with all kinds of possible data
- From method to method, all data variables in the program should be in a "correct" state
- Ex: a method that sorts should produce a sorted object
- Assertions: used during program development, testing for errors in the logic of the program
- They are usually "turned off" when a program is run by the user
- Such errors should not occur in the final version of the program
- Once an assertion is "thrown," the program terminates
- These errors should not happen in "real life"
- The string in the assertion statement is printed along with a stack trace
- Exception: an error that can be "thrown" by a method
- Ex: Integer.parseInt("abc") will throw an exception called "IllegalArgumentException"
- Our program can also throw these common exceptions and terminate

- To throw an exception:

```
throw new IllegalArgumentException
                        ("SSN must have only digits.");
```

6 GUIs and Inheritance

- It is common for the "main" application and the GUI to be separate classes
- By using inheritance, all things a JFrame can do is brought to the class

```
public class SSNGUI extends JFrame{?

}
```

- Non-Application Classes (no "main" method)
  - Classes without "main" methods are true objects
  - They cannot exist without being instantiated
  - They may inherit from other classes so little extra code must be written
  - When an object is instantiated, a special method called a "constructor" is automatically executed
  - The name of the constructor is the same as the name of the class
  - The constructor has no "return" attributes
  - The constructor is the initialization method for the object
  - The constructor may take parameters from the instantiating method for initial values

```
import javax.swing.*;
import java.awt.*;
public class SSNJFrame3{
   static final int LIST_SIZE = 10;
   static String ssn;
   static String[] ssnList;
   static int ssnSize;
   static TextFileInput inFile;
   static String inFileName = "SSN.txt";
   static JFrame myFrame;

   public static void main(String[] args) {
      initialize();
      readSSNsFromFile(inFileName);
      printSSNList(ssnList,ssnSize);
```

```
   printSSNtoJFrame(myFrame,ssnList,ssnSize);
}
public static void initialize() {
   ssn="";
   ssnList= new String[LIST_SIZE];
   ssnSize=0;
    inFile = new TextFileInput(inFileName);
    myFrame=new JFrame();
    myFrame.setSize(400,200);
    myFrame.setLocation(100, 100);
    myFrame.setTitle("Social Security Numbers");
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}
```

7 Defining a Simple Class

- An object is defined by its attributes and behavior
- Attributes: data values
- Behavior: defined by the methods
- The reason the data value is private is to ensure that the object contains correct data values
- Public data values can be changed by the user
- Methods that assign values to the data values should check for validity and throw an exception if they are not valid
- A method is private if it is not to be called from outside the class
- A method is static if it is not the behavior of an object
- Since all classes inherit from class Object, it automatically has methods:

```
equals(Object o);
toString();
```

- These methods may not behave the way we want to
- The "this" operator is a reference to the class in which it is used

8 Dynamic vs. Static Structures: Linked Lists

- Static storage structures: once declared, they are fixed in size
- the size may be a "variable" or a constant but once declared, the size is fixed
- Dynamic structures use only as much memory as they need
- Dynamic structures rely on themselves to determine the order of the data items they contain
- Dynamic structures are not stored in contiguous memory
- A node has 2 components: the actual data of the node and a reference linking it to the next node in the list
- It is helpful to have an empty dummy node at the beginning of the list
-

```
public class ListNode {
   String data;
   ListNode next;

   public ListNode(String data, ListNode next) {
      this.data = data;
      this.next = next;
   } // constructor

   public ListNode() {
      this.data = null;
      this.next = null;
   } // constructor

   public ListNode(String data) {
      this.data = data;
      this.next = null;
   } // constructor

}
```

```
public class LinkedListIterator {

   private ListNode node;
   public LinkedListIterator(ListNode first) {
        node = first;
   }

   public boolean hasNext() {
      return ( node != null );
```

```
   }

   public String next() {
      if ( node == null )
         throw new NullPointerException("Linked list empty.");
      String currentData = node.data;
      node = node.next;
      return currentData;
   }

}
```

-

```
public void append (String s) {
  ListNode n = new ListNode(s);
  last.next = n;
  last = n;
  length++;
}
```

```
public void printList () {
  ListNode p = first.next;
  while (p != null) {
    System.out.println(p.data);
    p = p.next;
  }
  }
```

-

```
public ListNode find (String s) {
  ListNode p = first.next;
  while (p != null && !(p.data).equals(s)) {
    p = p.next;
  } // while
  return p;
}
```

9 Inheritance and Polymorphism

- Extending a class is based on the ”is a” relationship
- The protected modifier grants access only from descendant classes
- Public grants access from any class
- Private grants access only to instances of the same class

- When a class is instantiated, the first thing it must do is "construct" its super class
- Calling one of the constructors of the super class is done using the method super(<optional parameters>)
- The name of the constructor is the same as the name of the class and has no return type
- An abstract class cannot be instantiated
- A class is abstract if
  - it is declared as abstract
  - it contains an abstract method
  - it inherits an abstract method and does not overload it
- The instanceof operator tests if an object (instance) is a subtype of a given type

10 GUIs and Event-Driven Programming

- JFrames: complete Window objects, they don't do anything until told to
- An event, such as a menu choice, signals the JFrame to respond
- Three Tier Architecture: graphical user interface, program logic - decisions based on the user's choices, data (files) - data available for the user's choice
- All the main program needs to do is instantiate the GUI
- An event is something that happens while the program is running
- An Event Handler is a method that is automatically called when an event, such as choosing a menu item, occurs
- An Event Handler can handle more than one event but each event needs a handler (or nothing will happen)
- Event Handlers are written in a class that implements an interface called ActionListener
- An Interface is a collection of method headings only (not bodies)
- Interfaces are implemented by a Java class
- If an interface is implemented, all methods specified in the interface must be provided by that class
- An interface, if implemented, guarantees that all methods will be defined
- The interface ActionListener contains a method called actionPerformed(ActionEvent)

- The actionPerformed method is called when an event happens
- Each event needs to be registered with some ActionListener
- When an ActionEvent object is created, the actionPerformed method of the handler that is registered with the event is called and the ActionEvent is passed to it as a parameter
-

```
import javax.swing.*;
import java.awt.*;
public class SSNGUI extends JFrame {

   public SSNGUI(String title, int height, int width) {
       setTitle(title);
       setSize(height,width);
       setLocation (400,200);
       createFileMenu();
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setVisible(true);
   } //SSNGUI

   private void createFileMenu( ) {
      JMenuItem  item;
      JMenuBar   menuBar = new JMenuBar();
      JMenu      fileMenu = new JMenu("File");
      FileMenuHandler fmh = new FileMenuHandler(this);

      item = new JMenuItem("Open"); //Open...
      item.addActionListener( fmh );
      fileMenu.add( item );

      fileMenu.addSeparator();        //add a horizontal separator line

      item = new JMenuItem("Quit");    //Quit
      item.addActionListener( fmh );
      fileMenu.add( item );

      setJMenuBar(menuBar);
      menuBar.add(fileMenu);

   } //createMenu

} //SSNGUI
```

```
import java.awt.event.*;
import java.io.*;
public class FileMenuHandler implements ActionListener {
```

```
   JFrame jframe;
   public FileMenuHandler (JFrame jf) {
      jframe = jf;
   }
   public void actionPerformed(ActionEvent event) {
      String menuName;
      menuName = event.getActionCommand();
      if (menuName.equals("Open"))
         openFile( );
      else if (menuName.equals("Quit"))
         System.exit(0);
   } //actionPerformed
   }
```

11 Exception Handling

- To throw an exception:

```
public SSN (String s) {
      if (isValidSSN(s) )
         SSNumber = s;
      else
         throw new IllegalSSNException("Invalid SSN: "+s);
   }
```

- Extending an existing exception:

```
public class IllegalSSNException
       extends IllegalArgumentException {
   public IllegalSSNException(String message) {
      super (message);
   }
}
```

- When an exception is thrown, the RunTime System looks for a method that can handle the exception
- If no such method is found, the Runtime System handles the exception and terminates the program
- The Runtime System looks at the most recently called method and backs up all the way to the main program
- If any one of the previous methods knew how to handle the exception, it would be an exception catcher

- Multiple exceptions can be taught by a try-catch block
- The JVM will go through the catch blocks top to bottom until a matching error is found
- This is why the order of the exceptions listed is important, because of the class hierarchy and inheritance
- When an exception is not caught, the program terminates and does not continue
- The finally block is executed whether or not an exception occurs
- It is executed even if there is a return statement prior to the final code
- Excluding exceptions in the class RuntimeException, the compiler must find a catcher or a propagator for every exception
- RuntimeException is an unchecked exception while all other exceptions are checked exceptions

12 Regular Expressions

- A Regular Expression (regex) is a pattern that can be matched against a string
-
```
import java.util.regex.*;

public static isValidSSN(String ssn) {
   Pattern p;
   Matcher m;
   String SSN_PATTERN =           ;
   p = Pattern.compile(SSN_PATTERN);
   m = p.matcher(ssn);
   return matcher.matches();

}
```

- Constants: match exactly the string inside the regex
- Character Classes (match any character inside []

abc - a, b, or c (simple class)

âbc - any character except a, b, or c (negation)

a-zA-Z - a through z, or A through Z, inclusive (range)

a-d[m-p] - a through d or m through p: [a -dm-p] (union)

a-z&&[def] - d, e or f (intersection)

a-z&&[b̂c] - a through z, except for b and c: [ad-z]

a-z&&[m̂-p] - a through z, and not m through p: [a-lq-z]

- Predefined Character Classes
  - . - any character (may or may not match line end)
  - \d - a digit: [0-9]
  - \D - a non-digit: [0̂-9]
  - \s - a whitespace character: [\t\n\x0B\f\r]
  - \S - a non-whitespace character: [\ŝ]
  - \w - a word character: [a-zA-Z_0-9]
  - \W - a non-word character: [\ŵ]
- Quantifiers
  - X? - X, once or not at all
  - X* - X, zero or more times
  - X+ - one or more times
  - Xn - X, exactly n times
  - Xn, - X, at least n times
  - Xn,m - X, at least n but not more than m times
- ^ beginning of regex, $ - end of regex

13 Maps

- In a hashmap, a "hash function" maps key to index
- Issues: search in time O(c), collisions, growth
- ```
  import java.util.HashMap;
  import java.util.Iterator;

  public class HashMapExample{
        public static void main(String args[]){
           HashMap hashMap = new HashMap();
           hashMap.put("One", new Integer(1));
           hashMap.put("Two", new Integer(2));
           hashMap.put("Three", new Integer(3));

          Integer myInt = hashMap.get("Two");

          if(hashMap.containsValue(new Integer(1)))
              System.out.println("HashMap contains 1 as value");
  ```
  ```

```
        else{
           System.out.println("HashMap does not contain 1 as value");

        if( hashMap.containsKey("One") )
           System.out.println("HashMap contains One as key");
        else
           System.out.println("HashMap does not contain One as value");
}
}
}
```

- To get the keys and values out of the HashMap
-

```
      Iterator itr;
      System.out.println("Retrieving all keys from the HashMap");

      itr = hashMap.keySet().iterator();
      while(itr. hasNext()){
         System.out.println(itr.next());
       }


       System.out.println("Retrieving all values from the HashMap");

       itr = hashMap.entrySet().iterator();
       while(itr. hasNext()){
          System.out.println(itr.next());
      }
```

- The order items went in is not the same as how they come out
- A TreeMap arranges the data keys so they come out in order when using the iterator
-

```
TreeMap <String, String> french =
        new TreeMap<String, String> ( );
```

- entrySet() returns a collection of key/value pairs
- interface Map.Entry is a key/value pair
-

```
Set set = french.entrySet();
Iterator i = set.iterator();
Map.Entry <String,String> me;

while(i.hasNext()) {
   me = (Map.Entry)i.next();
```

```
   System.out.print(me.getKey() + ": ");
   System.out.println(me.getValue());
}
```

- Order can be assumed if the class implements Comparable
-
```
TreeMap <String, String> french =
        new TreeMap<String, String> ( );
```

- For user-defined objects, the TreeMap needs to know how to order the keys
-
```
TreeMap <SSN, Integer> treeMap =
        new TreeMap (new SSNComparator());
```

- Comparator is a class that implements Comparator which has a method int compare (Object, Object)
-
```
import java.util.Comparator;

public class SSNComparator implements Comparator <SSN> {
   public int compare(SSN num1, SSN num2) {
      return num1.compareTo(num2);
   }
```

- The TreeMap is efficient because it keeps item in order and has O(n) for adding new values
- The TreeMap is based on the Red-Black tree

14 File Input and Output

- File I/O is done through the operating system: file system to operating system to program
- Files can be stored on a variety of devices, read/written by many operating systems
- Constructor for TextFileInput

```
public TextFileInput(String filename)
   {
      this.filename = filename;
      try {
         br = new BufferedReader(
                 new InputStreamReader(
                    new FileInputStream(filename)));
```

```
      } catch ( IOException ioe ) {
         throw new RuntimeException(ioe);
      } // catch
   } // constructor
}
```

- public FileInputStream(Stringname) throws FileNotFoundException

  Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system. A new FileDescriptor object is created to represent this file connection.

  Parameters: name - the system-dependent file name.
  Throws: FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

- public int read() throws IOException
  Reads a byte of data from this input stream. This method blocks if no input is yet available.

- An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

```
public TextFileInput(String filename)
   {
      this.filename = filename;
      try {
         br = new BufferedReader(
                 new InputStreamReader(
                     new FileInputStream(filename)));
      } catch ( IOException ioe ) {
         throw new RuntimeException(ioe);
      } // catch
   } // constructor
```

- The FileInputStream reads ASCII bytes from the file and delivers a stream of 32-bit int values

- The InputStreamReader converts the ints to a stream of Unicode characters (the default character set)

- public String readLine() throws IOException
  Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return (''), or a carriage return followed immediately by a linefeed.

Returns: A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached
Throws: IOException - If an I/O error occurs

- The BufferedReader separates the stream of Unicode characters into "lines" of the file (a line is terminated with lineFeed \n, carriageReturn or \n\r)
- The class File is an abstract representation of file and directory pathnames
-

```
import java.io.File;
import javax.swing.*;

public class SingleFile {
   public static void main (String args[]){
   JFileChooser fileChooser = new JFileChooser();
   fileChooser.showOpenDialog(null);
   File myFile = fileChooser.getSelectedFile();
   System.out.println("getName(): "+myFile.getName());
   System.out.println("getParent(): "+myFile.getParent());
   System.out.println("getPath(): "+myFile.getPath());
   System.out.println("lastModified(): "+myFile.lastModified());
   System.out.println("length(): "+myFile.length());
   }
}
```

```
import java.io.File;
import javax.swing.*;
public class ListFiles {

   public static void main(String[] args) {
        JFileChooser fd = new JFileChooser();
//       mode - the type of files to be displayed:
//           * JFileChooser.FILES_ONLY
//           * JFileChooser.DIRECTORIES_ONLY
//           * JFileChooser.FILES_AND_DIRECTORIES
       fd.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
       fd.showOpenDialog(null);
     File f = fd.getSelectedFile();
     listFiles(f,"");
   }
   public static void listFiles(File f, String indent) {
     File files[] = f.listFiles();

     for (int i = 0; i<files.length; i++) {
        File f2 = files[i];
        System.out.print(f2.getName());
        if (f2.isDirectory())
```

```
            listFiles(f2, indent+" ");
         System.out.print("...");
         System.out.print(f2.length());
         System.out.println();
      }
   }
}
```

15 Generics

- Generic: of, applicable to, or referring to all the members of a genus, class, group, or kind; general
- The advantage of using generics is to make generalized variables so it can be widely used without edited
-

```
public class ListNode <E> {
   E data;
   ListNode next;
   public ListNode(E myData) {
      data=myData;
      next=null;
   }
   public ListNode() {
      data=null;
      next=null;
   }
}
```

```
public class LinkedList<E> {
   private ListNode first;
   private ListNode last;
   private int length;

   public LinkedList() {
      ListNode ln = new ListNode();
      first = ln;
      last = ln;
      length = 0;
   }
   public void append ( E myData){
      ListNode n = new ListNode(myData);
      last.next = n;
      last = n;
      length++;
```

```
    }
    ....
}
```

- To make a LinkedList, write "LinkedList<String> stringList = new LinkedList<String>(); "

16 Recursion and the Run Time Stack

- A recursive method is a method that calls itself
- A recursive algorithm is one that solves a problem by using the same algorithm to solve a smaller part of the problem
- Ex: to list all the presidents of the US, identify the first president and then list the rest of the presidents
-

```
public class BinomialCoefficient {

   public static void main(String[] args) {
      int[] n = {4,4,4,4,4};
      int[] r = {0,1,2,3,4};
      int x,y;
      for (int i=0; i<n.length; i++) {
         x=n[i];
         y=r[i];
         System.out.println(x+" choose "+y+" is "+bc(x,y));
      }
   }
   private static int bc (int n, int r) {
       if (n==0 || r==0 ||n==r) {
            return 1;
      }
      else
         return bc(n-1,r)+bc(n-1,r-1);
   }
}
```

```
public class EuclidianGCD {

   public static void main(String[] args) {
      int[] testNumerators = {4,6,1,0,15,20};
      int[] testDenominators = {8,18,2,5,225,225};
```

```
      int n,d;
      for (int i=0; i<testNumerators.length; i++) {
         n=testNumerators[i];
         d=testDenominators[i];
         System.out.println("The greatest common divisor of "+n+" and "+d+
               " is "+gcd(n,d));
      }
   }
   private static int gcd (int n, int d) {
         if (d == 0) return n;
         else return gcd(d, n % d);
   }
}
```

-

```
public class Factorial {

   public static void main(String[] args) {
      int[] testValues = {4,6,1,0};
      int n;
      for (int i=0; i<testValues.length; i++) {
         n=testValues[i];
         System.out.println("Factorial("+n+") = "+factorial(n));
      }
   }
   private static int factorial (int n) {
      if (n==0)
         return 1;
      else
         return n*factorial(n-1);
   }
}
```

```
public class Fibonacci {

   public static void main(String[] args) {
      int[] testValues = {4,6,1,0,9,18};
      int n;
      for (int i=0; i<testValues.length; i++) {
         n=testValues[i];
         System.out.println("The "+n+suffix(n)+" Fibonacci number is
             "+fibonacci(n));
      }
   }
   private static int fibonacci (int n) {
      if (n==0)
```

```
        return 0;
      if
        (n==1)
        return 1;
      return fibonacci(n-1)+fibonacci(n-2);
   }
}
```

- ---

```
import javax.swing.*;
public class Pascal {

     public static void main (String[] args){
        int numRows =
            Integer.parseInt(JOptionPane.showInputDialog(null,"How many
            rows?"))-1;
        for (int i=0;i<= numRows;i++){
           for (int j=0; j<=i; j++)
              System.out.print(bc(i,j)+" ");
           System.out.println();
        }
     }
     private static int bc (int n, int r) {
        if (n==0 || r==0 ||n==r) {
             return 1;
       }
       else
          return bc(n-1,r)+bc(n-1,r-1);
    }

}
```

```
public class TowersOfHanoi {

   public static void main(String[] args) {
      moveRings(3,"A","B","C");

   }
   private static void moveRings(int numberOfRings, String fromTower,
                                 String toTower, String tempTower) {
      if (numberOfRings == 0) return;
      moveRings (numberOfRings-1, fromTower, tempTower, toTower);
      System.out.println("Move a ring from tower "+fromTower+" to tower "+
                         toTower);
      moveRings (numberOfRings-1, tempTower, toTower, fromTower);
   }
```

```
}
```

17 Model-View-Controller (MVC)

- Model: a representation of the data with no concern for how it will appear to the user - extends Observable
- View: displays the data using GUI components by observing the Model - implements Observer
- Controller: a Listener that responds to events and updates the Model
- Regular Program

```
public class TemperatureModel {
    private double temperatureF = 32.0;
    public double getF() {
      return temperatureF;
  }
  public double getC(){
      return (temperatureF - 32.0) * 5.0 / 9.0;
  }
  public void setF(double tempF)
  {
      temperatureF = tempF;
  }
  public void setC(double tempC)
  {  temperatureF = tempC*9.0/5.0 + 32.0;
  }
}
```

- Model Class

```
import java.util.Observable;
public class TemperatureModel extends Observable {
    private double temperatureF = 32.0;
    public double getF() {
      return temperatureF;
   }
  public double getC(){
      return (temperatureF - 32.0) * 5.0 / 9.0;
   }
  public void setF(double tempF)
   {
      temperatureF = tempF;
```

```
      setChanged();
      notifyObservers();
   }
  public void setC(double tempC)
   {  temperatureF = tempC*9.0/5.0 + 32.0;
      setChanged();
      notifyObservers();
   }
}
```

- Controller (Listener)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

  class UpListener implements ActionListener {
     TemperatureModel model;

      public UpListener(TemperatureModel m) {
      model = m;
   }

      public void actionPerformed(ActionEvent e) {
         model.setF(model.getF() + 1.0);
      }
   }
```

- GUI class

```
import java.awt.*;
import java.awt.event.*;
abstract class TemperatureGUI implements java.util.Observer {
   private String label;
   private TemperatureModel model;
   private Frame temperatureFrame;
   private TextField display = new TextField();
   private Button upButton = new Button("Raise");
   private Button downButton = new Button("Lower");

   TemperatureGUI(String theLabel, TemperatureModel tModel, int h, int v) {
      label = theLabel;
      model = tModel;
      Frame temperatureFrame;
      temperatureFrame = new Frame(label);
      temperatureFrame.add("North", new Label(label));
      temperatureFrame.add("Center", display);
      Panel buttons = new Panel();
```

```
        buttons.add(upButton);
        buttons.add(downButton);
        temperatureFrame.add("South", buttons);
        temperatureFrame.addWindowListener(new CloseListener());
        model.addObserver(this); // Connect the View to the Model
        temperatureFrame.setSize(200,100);
        temperatureFrame.setLocation(h, v);
        temperatureFrame.setVisible(true);

    public void setDisplay(String s){
        display.setText(s);}
    public double getDisplay() {
        return Double.valueOf(display.getText()).doubleValue();
    }
        continued?
}
}
```

- If it is to operate in Fahrenheit

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

public class FarenheitGUI extends TemperatureGUI {

    public FarenheitGUI(TemperatureModel model, int h, int v) {
        super("Farenheit Temperature", model, h, v);
        setDisplay(""+model.getF());
        addUpListener(new UpListener(model));
        addDownListener(new DownListener(model));
        addDisplayListener(new DisplayListener(model,this));
    }

    public void update(Observable t, Object o)
    // automatically called when the model is changed
    {
        setDisplay("" + model().getF());
    }

}
```

18 Threads

- Thread: an instance of program execution - generalized as a Process

- A single Java application could be considered a Thread but a Java application may contain multiple threads
- The value of threads can be seen by looking at process states that are linked in a continuous loop
- Process States:
  - Ready: threads that are ready to run
  - Running: threads that are running on a CPU
  - Waiting: threads that are waiting for something
- The operating system (or the JVM) is responsible for moving processes (threads) from state to state
- To instantiate a new thread: "Thread t = new Thread();"
- t.start() will make the thread go from ready to running
- t.sleep(ms), t.suspend(),t.wait() will make the thread go from running to waiting
- t.stop() will kill the thread
- t.resume() will make the thread go from waiting to ready
- Making a timer work

```
import java.awt.*;
import javax.swing.*;
public class TimerJFrame extends JFrame implements Runnable {
   private int secondsRemaining;
   private JTextArea text = new JTextArea();
   public TimerJFrame (int seconds) {
      secondsRemaining = seconds;
      setTitle("Time Remaining...");
      setSize(150,150);
      setLocation (400,200);
      Container cp = getContentPane();
      text.setFont(new Font("Arial",2,72));
      cp.add(text);
      text.append(Integer.toString(secondsRemaining));
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      Thread timer = new Thread(this);
      timer.start();
   }
   public void run() {
  System.out.println("The game has started...");
  while (secondsRemaining > 0) {
```

```
    try {
         Thread.sleep(1000);
         secondsRemaining--;
         text.setText(Integer.toString(secondsRemaining));
         setVisible(true);
      }
        catch (InterruptedException ie) {
           System.out.println("Timer is interrupted");
        }
   }
   JOptionPane.showMessageDialog(null,"Time is up!");
   }
   }
```

- Another example

```
public class LoggingThread extends Thread {
  private LinkedList linesToLog = new LinkedList();
  private volatile boolean terminateRequested;

  public void run() {
    try {
      while (!terminateRequested) {
        String line;
        synchronized (linesToLog) {
          while (linesToLog.isEmpty())
            linesToLog.wait();
          line = (String) linesToLog.removeFirst();
        }
        doLogLine(line);
      }
    } catch (InterruptedException ex) {
      Thread.currentThread().interrupt();
    }
  }
  private void doLogLine(String line) {
    // ... write to wherever
  }
  public void log(String line) {
    synchronized (linesToLog) {
      linesToLog.add(line);
      linesToLog.notify();
    }
  }
}
```