CN208 Introduction to Computer Programming

Lecture #12

Streams (Continued), Applets, Event Handling, Graphical User Interface (GUI)

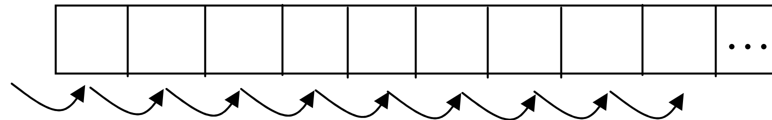
Pimarn Apipattanamontre

Email: ***pimarn@pimarn.com***

Sequential Access File

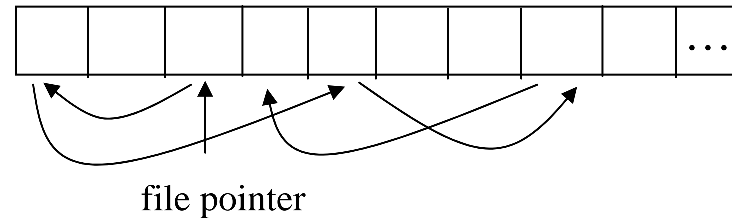
- In sequential file access, a file is processed one byte at a time in the order data items appear in the file.
- To read data from a file, we start from the beginning of the file and read each byte until we reach a particular byte of the desired data item.

Sequential Access



Random Access File

Random Access



- In random file access, everything is processed in the binary format.
- A random access file behaves like a large array of bytes stored in the file system.
- There is a kind of index for this implied array, called the *file pointer.*
- We can go directly to a specific location (by moving the file pointer to that location) and then access the byte at that location without first reading the preceding bytes.

The RandomAccessFile Class

- To construct a *RandomAccessFile* object, we (1) supply the filename or an object of the *File* class and (2) specify the open mode:

```
RandomAccessFile f = new RandomAccessFile("accounts.dat", "rw");
```

- There are two possible open modes:
  - "r" opens the random access file for reading only.
  - "rw" opens the random access file for reading and writing.
- Because a file can be very large, the location of the file pointer is of type *long*.
- To move the file pointer to the *nth* byte counted from the beginning of the file (the first byte of the file is the *0th* byte), we use

```
f.seek(n); // n is a long integer
```

- To find out the current position of the file pointer (counted from the beginning of the file), we use

```
long n = f.getFilePointer();
```

- To get the number of bytes in a random access file, use the *length( )* method.

```
long fileSize = f.length();
```

The RandomAccessFile Class

- Input operations read bytes starting at the current location of the file pointer and advance the file pointer past the bytes read.

```
double x = f.readDouble();
```

The *readDouble( )* method reads an 8-byte data item starting from the current location of the file pointer.

```
int x = f.readInt();
```

The readInt( ) method reads a 4-byte data item starting from the current location of the file pointer.

The RandomAccessFile Class

- If the random access file is created in read/write mode, then output operations are also available.
- Output operations write bytes starting at the file pointer and advance the file pointer past the bytes written.

```
double x = 29.95;
f.writeDouble(x);
```

The *writeDouble( )* method writes an 8-byte data item into the file starting from the current location of the file pointer.

```
int y = 50;
f.writeInt(y);
```

The *writeInt( )* method writes a 4-byte data item into the file starting from the current location of the file pointer.

Example of Processing a Random Access File

File SavingsAccount.java

```
public class SavingsAccount extends BankAccount
{   // instance variable
    private double interestRate;
    public static final int DOUBLE_SIZE = 8;
    public static final int ACCOUNT_SIZE = 2 * DOUBLE_SIZE;

    // methods
    public void addInterest()
    {  double interest = getBalance() * interestRate / 100;
       deposit(interest);
    }
    public double getInterestRate()
   {   return interestRate; }

    // constructors
    public SavingsAccount(double initBalance, double rate)
    {  super(initBalance);
       interestRate = rate;
    }
}
```

File RandomAccFile.java

```
import java.io.*;

public class RandomAccFile
{
   public static void main(String[] args) throws IOException
   { File accFile = new File("accounts.dat");
     RandomAccessFile f = new RandomAccessFile(accFile, "rw");
     f.seek(0);
     for (int i=1; i<=2; i++)
    {  SavingsAccount anAcc = new SavingsAccount(1000*i, 1*i);
       f.writeDouble(anAcc.getBalance());
       f.writeDouble(anAcc.getInterestRate());
    }

    f.seek(1*SavingsAccount.ACCOUNT_SIZE);
    double balance = f.readDouble();
    double interestRate = f.readDouble();
    SavingsAccount aTempAcc = new SavingsAccount(balance, interestRate);
    aTempAcc.addInterest();
    f.seek(1*SavingsAccount.ACCOUNT_SIZE);   ⇦
    f.writeDouble(aTempAcc.getBalance());
    f.writeDouble(aTempAcc.getInterestRate());

    f.close();
  }
}
```

File accounts.dat (in binary format)

1000.0	1.0	2000.0	2.0

0 8 16 24 byte position

Example of Processing a Random Access File

File SavingsAccount.java

```
public class SavingsAccount extends BankAccount
{   // instance variable
    private double interestRate;
    public static final int DOUBLE_SIZE = 8;
    public static final int ACCOUNT_SIZE = 2 * DOUBLE_SIZE;

    // methods
    public void addInterest()
    {  double interest = getBalance() * interestRate / 100;
       deposit(interest);
    }
    public double getInterestRate()
   {   return interestRate; }

    // constructors
    public SavingsAccount(double initBalance, double rate)
    {  super(initBalance);
       interestRate = rate;
    }
}
```

File accounts.dat (in binary format)

1000.0	1.0	2040.0	2.0

0 8 16 24 byte position

File RandomAccFile.java

```
import java.io.*;

public class RandomAccFile
{
   public static void main(String[] args) throws IOException
  {  File accFile = new File("accounts.dat");
     RandomAccessFile f = new RandomAccessFile(accFile, "rw");
     f.seek(0);
     for (int i=1; i<=2; i++)
    { SavingsAccount anAcc = new SavingsAccount(1000*i, 1*i);
       f.writeDouble(anAcc.getBalance());
       f.writeDouble(anAcc.getInterestRate());
    }

    f.seek(1*SavingsAccount.ACCOUNT_SIZE);
    double balance = f.readDouble();
    double interestRate = f.readDouble();
    SavingsAccount aTempAcc = new SavingsAccount(balance, interestRate);
    aTempAcc.addInterest();
    f.seek(1*SavingsAccount.ACCOUNT_SIZE);
    f.writeDouble(aTempAcc.getBalance());
    f.writeDouble(aTempAcc.getInterestRate());

    f.close();
  }
}
```

Applets

- Applets are programs that run inside a web browser.
- The code for an applet is stored on a web server and downloaded into the browser while we access a web page that contains the applet.
- Applets can be executed in multiple platforms because they are delivered as Java bytecode (.class files).
- To display an applet together with a web page, we need to write and compile a Java file to generate the applet code.
- Then, we tell the web browser how to find the applet code and how much screen space to reserve for the applet in a web-page file (HTML file).

HTML file format

File rectangle.html

```
<P>Here is my <I> first applet: </I> </P>
<APPLET CODE = “RectangleApplet.class” WIDTH=300 HEIGHT=300>Here
is a rectangle applet</APPLET>
```

- HTML files are made up of text and pairs of tags. The closing tag is like the opening tag but it is prefixed by a slash (/).
  - A paragraph is delimited by the tag pair <P> and </P>.
  - The tag pair <I> and </I> directs the web page to display the text inside the tags as *italics*.
  - A tag can have attributes. Each attribute has a name and a value.
  - The <APPLET> </APPLET> tag has three attributes: CODE, WIDTH, HEIGHT.
  - The CODE attribute tells where to find the applet code while the WIDTH and HEIGHT attributes specify screen space to reserve for the applet.
  - The text between the <APPLET> and </APPLET> tags are displayed when the browser cannot run the applet.

Applets

- We can also run an applet by using the *applet viewer*, a program that is included with Java Development Kit from Sun Microsystems, Inc.
- To run the applet viewer, we start the viewer and give the name of the HTML file that contains our applets.

```
%appletviewer rectangle.html
```

- The applet viewer brings up one window for each applet in the HTML file.

A Simple Applet

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class RectangleApplet extends Applet
{
  public void paint(Graphics g)
  {  Graphics2D g2 = (Graphics2D)  g;
     Rectangle aBox = new Rectangle(5, 10, 20, 30);
     g2.draw(aBox);

     aBox.translate(15, 25);
     g2.draw(aBox);
  }
}
```

- An applet is programmed as a class, like any other program in Java.
- The class is declared *public* and it extends the *Applet* class.
- The applet does not have the *main( )* method since the web browser (or applet viewer) is responsible for starting up the Java virtual machine, loading the applet code, and starting the applet.

A Simple Applet

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class RectangleApplet extends Applet
{
  public void paint(Graphics g)
  {  Graphics2D g2 = (Graphics2D)  g;
     Rectangle aBox = new Rectangle(5, 10, 20, 30);
     g2.draw(aBox);

     aBox.translate(15, 25);
     g2.draw(aBox);
  }
}
```

- In our simple applet, we override the *paint( )* method of the *Applet* class.
- The *paint( )* method is used for drawing graphical shapes (in this case, two rectangles) in the applet.
- The *paint( )* method is automatically called when the applet is shown for the first time or when the web page is refreshed.

A Simple Applet

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class RectangleApplet extends Applet
{
  public void paint(Graphics g)
  {  Graphics2D g2 = (Graphics2D)  g;
      Rectangle aBox = new Rectangle(5, 10, 20, 30);
      g2.draw(aBox);

      aBox.translate(15, 25);
      g2.draw(aBox);
  }
}
```

- The *paint( )* method needs an object of type *Graphics*.
- The *Graphics* object stores the graphics state: the current color, font, and so on, that are used for the drawing operations.
- The *Graphics2D* class is a newer version of the *Graphics* class and it currently replaces the *Graphics* class.
- To maintain the backward compatibility for the *paint( )* method, the *Graphics* class is still maintained as its parameter.

A Simple Applet

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class RectangleApplet extends Applet
{
  public void paint(Graphics g)
  {  Graphics2D g2 = (Graphics2D)  g;
     Rectangle aBox = new Rectangle(5, 10, 20, 30);
     g2.draw(aBox);

     aBox.translate(15, 25);
     g2.draw(aBox);
  }
}
```

- The Java designer defines the *Graphics2D* class as the subclass of the *Graphics* class.
- When the *paint( )* method is called, the *Graphics2D* object is actually passed as its parameter from the calling point.
- We can recover the object reference of the *Graphics2D* by using a cast inside the *paint( )* method.
- We use the *draw( )* method of the Graphics2D class to draw shapes such as rectangles, ellipses, arcs, etc.

A Simple Applet

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;

public class RectangleApplet extends Applet
{
  public void paint(Graphics g)
  {  Graphics2D g2 = (Graphics2D)  g;
     Rectangle aBox = new Rectangle(5, 10, 20, 30);
     g2.draw(aBox);

     aBox.translate(15, 25);
     g2.draw(aBox);
  }
}
```

- The *Graphics*, *Graphics2D*, and *Rectangle* classes are part of the *java.awt* package.
- AWT stands for Abstract Windowing Toolkit. It is the original toolkit supplied for Java to draw graphical user interface components.
- The newer toolkit to draw graphical user interface components is defined in the *javax.swing* package. (The *javax* package denotes the standard extension of Java.)

Event Handling

- In the console application programs we have written so far, user input is under control of the program.
- The program asks the user to supply input in a fixed order.
- In a program with a modern graphical user interface, the user is in control.
- The user can manipulate many parts of the user interface such as entering information into text fields, pulling down the menus, drag the scroll bars, in any order.
- The program must react the user commands, in whatever order they arrive.

Event Sources, Event Listeners, and Events

- To write Java programs that can react to user interface events such as mouse clicks, button pushes, there are three classes involved:

  (a) The event source

  - The event source is the user interface component that can generate a particular event.

    For example:

    - A button is an event source for a button click event.
    - A menu item is an event source for a menu selection event.
    - An applet is an event source. When the user clicks anywhere inside the applet, the mouse click event is generated.

  (b) The event class

  - Objects of the event class are generated from the event source when a user reacts to the user interface component (the event source),.

    For example:

    - When the user clicks a mouse inside the applet, a MouseEvent object (containing detailed information about the event) is generated.

Event Sources, Event Listeners, and Events

(c) The event listener

- In practical, there can be a huge number of events generated by an event source.

  For example:

  - When we move the mouse inside an applet, a lot of mouse move events are generated.

- In order not to be flooded by boring events, we will install the event listener object to the event source to listen (or select) *only* the events in which we are interested.
- When an event occurs (generated by the event source), the event source notifies all installed event listeners and supplies the event object to the listener.

Mouse Listener Interface

- The event listener is actually an interface that contains abstract methods for handling the event.
- The MouseListener interface contains several methods that can be called when a user reacts to the applet (the event source) with a mouse.

File MouseListener.java

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button is pressed on  the applet
   void mouseRelased(MouseEvent event);
   // Called when a mouse button is released on the applet
   void mouseClicked(MouseEvent event);
   // Called when a mouse button is clicked on the applet
   void mouseEntered(MouseEvent event);
   // Called when a mouse button enters the applet
   void mouseExited(MouseEvent event);
   // Called when a mouse button exits the applet
}
```

Mouse Listener Interface

- We need to define a mouse listener class that implements the *MouseListener* interface.
- The *MouseSpy* class implements each method by printing out the cause of the event and the x- and y- position of the mouse at which the event occurs.

File MouseListener.java

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button is pressed on  the applet
   void mouseRelased(MouseEvent event);
   // Called when a mouse button is released on the applet
   void mouseClicked(MouseEvent event);
   // Called when a mouse button is clicked on the applet
   void mouseEntered(MouseEvent event);
   // Called when a mouse button enters the applet
   void mouseExited(MouseEvent event);
   // Called when a mouse button exits the applet
}
```

File MouseSpy.java

```
import java.awt.event.MouseListener;
import java.awt.event MouseEvent;
public class MouseSpy implements MouseListener
{
   public void mousePressed(MouseEvent event)
   { System.out.println("Mouse pressed. x = " + event.getX( ) +
          " y = " + event.getY( ) );
   }
   public void mouseRelased(MouseEvent event)
  { System.out.println("Mouse released. x = " + event.getX( ) +
          " y = " + event.getY( ) );
   }
   public void mouseClicked(MouseEvent event)
  { System.out.println("Mouse clicked. x = " + event.getX( ) +
          " y = " + event.getY( ) );
   }
   public void mouseEntered(MouseEvent event)
   { System.out.println("Mouse entered. x = " + event.getX( ) +
          " y = " + event.getY( ) );
   }
   public void mouseExited(MouseEvent event)
   { System.out.println("Mouse exited. x = " + event.getX( ) +
          " y = " + event.getY( ) );
   }
}
```

Mouse Listener Interface

- To install the mouse listener, we call the *addMouseListener( )* method of the event source (the applet that receives the mouse clicks).

File MouseListener.java

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button is pressed on  the applet
   void mouseRelased(MouseEvent event);
   // Called when a mouse button is released on the applet
   void mouseClicked(MouseEvent event);
   // Called when a mouse button is clicked on the applet
   void mouseEntered(MouseEvent event);
   // Called when a mouse button enters the applet
   void mouseExited(MouseEvent event);
   // Called when a mouse button exits the applet
}
```

File MouseApplet.java

```
import java.applet.Applet;
public class MouseApplet extends Applet
{
  public MouseApplet( )
  {
    MouseSpy listener = new MouseSpy()
    addMouseListener(listener);
  }
}
```

File MouseSpy.java

```
import java.awt.event.MouseListener;
import java.awt.event MouseEvent;
public class MouseSpy implements MouseListener
{
   public void mousePressed(MouseEvent event)
   { System.out.println("Mouse pressed. x = " + event.getX( ) +
         " y = " + event.getY( ) );
   }
   public void mouseRelased(MouseEvent event)
  { System.out.println("Mouse released. x = " + event.getX( ) +
         " y = " + event.getY( ) );
   }
   public void mouseClicked(MouseEvent event)
  { System.out.println("Mouse clicked. x = " + event.getX( ) +
         " y = " + event.getY( ) );
   }
   public void mouseEntered(MouseEvent event)
   { System.out.println("Mouse entered. x = " + event.getX( ) +
         " y = " + event.getY( ) );
   }
   public void mouseExited(MouseEvent event)
   { System.out.println("Mouse exited. x = " + event.getX( ) +
         " y = " + event.getY( ) );
   }
}
```

Adapter Class

- In general, a program is interested only in some listener notifications.
- For example, we may be interested only in the mouse-pressed event and may not care about any other events.
- One possible solution we can do is that we supply a listener class that implements only the methods in which we are interested while leaving other methods as “do nothing” methods.

File MouseListener.java

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button is pressed on  the applet
   void mouseRelased(MouseEvent event);
   // Called when a mouse button is released on the applet
   void mouseClicked(MouseEvent event);
   // Called when a mouse button is clicked on the applet
   void mouseEntered(MouseEvent event);
   // Called when a mouse button enters the applet
   void mouseExited(MouseEvent event);
   // Called when a mouse button exits the applet
}
```

File MousePressListener.java

```
import java.awt.event.MouseListener;
import java.awt.event MouseEvent;
public class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   { System.out.println(“Mouse pressed. x = ” + event.getX( ) +
         “ y = ” + event.getY( ) );
   }
   public void mouseClicked(MouseEvent event) { }
   public void mouseReleased(MouseEvent event) { }
   public void mouseEntered(MouseEvent event) { }
   public void mouseExited(MouseEvent event)   { }
}
```

Adapter Class

- Java defines a corresponding adapter class for each listener class.
- For example, the mouse adapter class implements the mouse listener class by defining all methods as “do-nothing” methods.
- We can define a listener class by extending the MouseAdapter class and overriding only the methods that we care about.

File MouseListener.java

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
   // Called when a mouse button is pressed on  the applet
   void mouseRelased(MouseEvent event);
   // Called when a mouse button is released on the applet
   void mouseClicked(MouseEvent event);
   // Called when a mouse button is clicked on the applet
   void mouseEntered(MouseEvent event);
   // Called when a mouse button enters the applet
   void mouseExited(MouseEvent event);
   // Called when a mouse button exits the applet
}
```

File MouseAdapter.java

```
public class MouseAdapter implements MouseListener
{
   public void mousePressed(MouseEvent event) { }
   public void  mouseClicked(MouseEvent event) { }
   public void  mouseReleased(MouseEvent event) { }
   public void  mouseEntered(MouseEvent event) { }
   public void  mouseExited(MouseEvent event)   { }
}
```

File MousePressListener.java

```
import java.awt.event.MouseAdapter;
import java.awt.event MouseEvent;
public class MousePressListener extends MouseAdapter
{
   public void mousePressed(MouseEvent event)
   { System.out.println(“Mouse pressed. x = ” + event.getX( ) +
         “ y = ” + event.getY( ) );
   }
}
```

Processing Mouse Input

- Suppose that we want to write a program that moves a rectangle to the mouse press position.

File MouseApplet.java

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;
public class MouseApplet  extends Applet
{      public MouseApplet( )
       { box = new Rectangle(100, 200, 20, 30);
         //add mouse listener to the applet (the event source)
         MousePressListener listener = new MousePressListener( );
         addMouseListener(listener);
       }
       public paint(Graphics g)
       { Graphics2D g2 = (Graphics2D) g;
         g2.draw(box);
       }

       private Rectangle box;
}
```

⇦ The *constructor* is called when the object of this class is constructed by Java virtual machine.

⇦ The *paint( )* method is called when the applet is first loaded or when the web page is refreshed.

File rectangle.html

```
<P>Here is my <I> mouse applet: </I> </P>
<APPLET CODE = “MouseApplet.class” WIDTH=300
HEIGHT=300>Here is a mouse applet</APPLET>
```

File MousePressListener.java

```
import java.awt.event.MouseAdapter;
import java.awt.event MouseEvent;
public class MousePressListener extends MouseAdapter
{
   public void mousePressed(MouseEvent event)
   {   int x = event.getX( );
       int y = event.getY( );
       box.setLocation(x, y);
   }
}
```

⇦ We have the problem here! We want to move the box to (x, y) but the *MousePressListener* class cannot access the private instance variable *box* of the *MouseApplet* class.

Inner Class

File MouseApplet.java

```
import java.applet.Applet;
import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.MouseAdapter;
import java.awt.event MouseEvent;
public class MouseApplet extends Applet
{  public MouseApplet( )
   {  box = new Rectangle(100, 200, 20, 30);
      // add mouse listener to the applet (the event source)
      MousePressListener listener = new MousePressListener( );
      addMouseListener(listener);
   }
   public paint(Graphics g)
   { Graphics2D g2 = (Graphics2D) g;
     g2.draw(box);
   }
   private Rectangle box;

private class MousePressListener extends MouseAdapter
   {   public void mousePressed(MouseEvent event)
       {  int x = event.getX( );
          int y = event.getY( );
          box.setLocation(x, y);
          repaint( );
       }
   }
}
```

⇦ We can call the *repaint( )* method of the *MouseApplet* class (inherited from its superclass) from the method of the inner class to tell the applet to repaint itself. The *paint( )* method is generally used by Java virtual machine and is not recommended for us to use.

- The problem can be solved by making the mouse listener into an ***inner class***.
- An inner class is a class defined inside another class.
- The inner class is like a regular class. We can construct objects of the inner class and can call methods through its objects.
- The methods of the inner class are allowed to
  - access the private instance variables of the outer class, and
  - call the methods of the outer class.

Inner Class

- Inner classes are very useful when defining event listener classes.
- Whenever a method of an event listener class needs to access instance variables or call methods of a class, we define the listener class as the inner class of that particular class by using the following outline:

File MyClass.java

```
public class MyClass
{
   public MyClass( )
   {  . . .
      // construct a listener object and add it to the event source
      MyListener listener = new MyListener( );
      anEventSource.addAListener(listener);
   }
   // outer-class methods are defined here
   . . .
   // outer-class instance variables are defined here
   . . .
   // inner-class definition
   private class MyListener extends AnEventAdapter
   {  public void anEventOccurred(AnEvent event)
      {   . . .
          // OK to access outer-class instance variables or
          // call outer-class methods here
      }
   }
}
```

Writing Graphical Applications

- All graphical programs that we wrote have been *applets*, programs that run inside a web browser or the applet viewer.
- We can also write regular graphical programs in Java that do not run inside another program.
- Graphical programs may have several graphical user interface components such as buttons, text fields, menus, etc.
- All graphical user interface components are defined in the *javax.swing* package.

Frame Windows

- Every graphical program requires one or more frame windows.
- A frame window is designed as an area to put graphical user interface components.
- To show a frame, we use the *JFrame* class in the *javax.swing* package.
- To create a frame window object, use

```
JFrame f = new Jframe( );
```

  Constructs a new frame with a size of 0 x 0 pixels.
- To set the frame with a specific size (*width* x *height* pixels), use

```
f.setSize(int width, int height);
```

- To set the title of the frame, use

```
f.setTitle(String title);
```

- To show the frame, use

```
f.show();
```

- To set the frame to be the size that can fit all graphical interface components, use

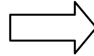
```
f.pack();
```

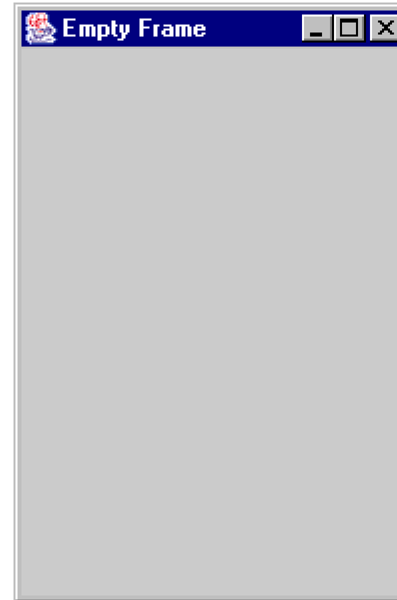
Empty Frame

File FrameTest1.java

```
import javax.swing.JFrame;

public class FrameTest1
{  public static void main(String[] args)
   {  JFrame f = new JFrame();
      f.setSize(200, 300);
      f.setTitle("Empty Frame");
      f.show();
   }
}
```

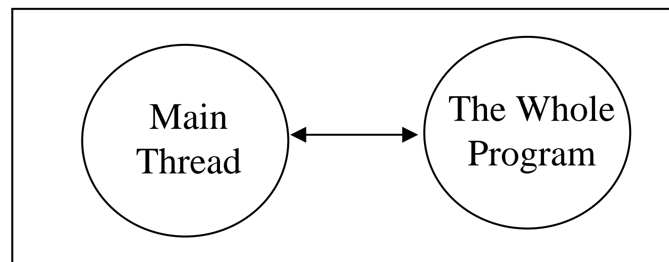




- Unlike applets, graphical applications have a *main( )* method.
- The *main( )* method of the *FrameTest1.java* program constructs a *JFrame* object, sets up the title of the object and calls the *show( )* method to pop up an empty frame.
- Although the *main( )* method is finished, the user interface component (the frame window) is still running. Thus, the program is still running.
- Although we close the frame window, the program is still running.
- We can terminate (kill) the program by typing Ctrl+C in the console window.

Single-Threaded Applications

- There is an important difference between console programs and graphical programs.
- In console programs:
  - There is only one unit of virtual processor (called a *thread*).
  - We can view a thread as a processor that has an environment set up appropriately to run a particular part of the program.
  - When the console program is executed, a so-called *main thread* is set up and starts to execute the program sequentially from beginning to end.
  - When the program is completely executed, the *main thread* is finished.



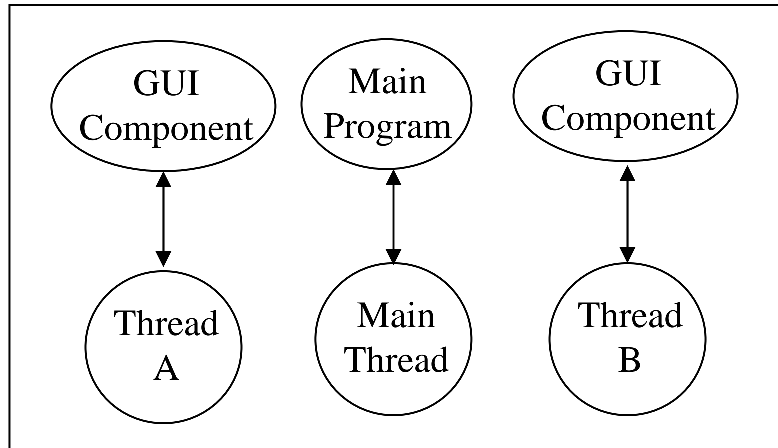
Single-Threaded Applications (Console Programs)

Multi-Threaded Applications

- In graphical programs:
  - There can be more than one thread while the program is being executed.
  - The main thread executes instructions at the main program (do the main processing part).
  - Whenever a user interface component (for example, a frame window object) is created, there will be a corresponding thread constructed and set up for taking care of that frame window.
  - At a particular time, there can be many threads executing parts of the program *concurrently*.
  - When the main program is finished, the main thread is completed.
  - The program does not terminate since there are still other threads running.
  - To terminate the program (kill all the threads), use

```
System.exit(0)
```

Multi-Threaded Applications



Multi-Threaded Applications (Graphical Programs)

GUI = Graphical User Interface