

CS 326

Programming Languages, Concepts and Implementation

Instructor: Mircea Nicolescu

Java

The Java Programming Language

- Developed in the early 1990s by James Gosling and associates at Sun Microsystems
- Initially intended for embedded systems – microwave ovens, TV sets, etc
- Main characteristics:
 - portable, platform independent
 - loaded dynamically from a network
 - robust (strongly typed, no pointers, garbage collection)
 - closer to the OO paradigm
 - support for multi-threading, graphics, remote communication

Portability

- Java compiler (**javac**):
 - translates source code into byte code (machine-independent)
- Java interpreter (**java**):
 - interprets and runs byte code
 - implemented for a particular architecture (machine-dependent)
 - machine + interpreter = Java Virtual Machine
- Efficiency?
 - since it uses an interpreter – would be slower than a compiled program
 - improvement – just-in-time compilation
 - first translate byte code into machine target code, then run it

Class Definitions: A Peek

```
public class Point {  
    private int x,y;  
    private Color myColor;  
}
```

field definitions

```
    public int currentX() {  
        return x;  
    }
```

```
    public int currentY() {  
        return y;  
    }
```

```
    public void move(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }
```

method definitions

- Everything must be inside some class
- No stand-alone functions
- No global variables

Data Types

- **Primitive types**
 - **boolean**: **true** and **false**
 - **char**: 16 bit Unicode character set
 - **byte**: 8 bits
 - **short**: 16 bits
 - **int**: 32 bits
 - **long**: 64 bits
 - **float**: 32 bits floating point (IEEE standard)
 - **double**: 64 bits floating point (IEEE standard)

Data Types

- **Constructed types**
 - Any class name, like `Point`
 - Any array type, like `Point[]` or `int[]`
- Variable model
 - **Value model** for primitive types
 - **Reference model** for constructed types
- “Java is like C++ without pointers”
 - Not really – in fact, any variable (for a constructed type) is a pointer
 - But you don’t see (or have access to) pointers explicitly

Strings

- Predefined but not primitive: a class **String**
- “hello” works like a string constant
- It is actually an object (instance of the **String** class), containing the given string of characters
- The **+** operator has special overloading and coercion behavior for the class **String**:

Java Expression	Value
<code>"123"+"456"</code>	<code>"123456"</code>
<code>"The answer is " + 4</code>	<code>"The answer is 4"</code>
<code>" " + (1.0/3.0)</code>	<code>"0.3333333333333333"</code>

- Other useful methods:
 - `length()`, `charAt(i)`, `toUpperCase()`, ...

Enjoy...

```
private static int stringSize(String s){  
    int size = 0;  
    for (int i = 0; i < s.length(); i++) {  
        size++;  
    }  
    return size;  
}
```


Object Creation and Destruction

- To create a new object that is an instance of a given class – use **new**:

```
Point p;                // just the declaration  
p = new Point; // now it is allocated
```

- May also have parameters, if an appropriate constructor has been defined:

```
p = new Point(3,8);
```

- To deallocate an object – do nothing
 - **Garbage collection**

Arrays

- Always allocated dynamically:

```
int a [];           // just the declaration
a = new int[4];     // now it is allocated
```

```
double m [][];
m = new double [5][100];
```

Arrays are also objects, not just a contiguous collection of elements:

after allocation, can obtain the number of elements by using the field **length**:

```
int x = a.length;    // 4 elements
int x = m.length;    // 5 elements
int x = m[0].length; // 100 elements
```

Classes

- Visibility:
 - a class may be **public** or not (no label)
 - each field and method may be **public**, **private** or **protected**
 - additional labels:
 - **final**: a constant field, or a method that cannot be redefined in derived classes
 - **static**: a field or method shared by all instances (belongs to the class, not to each instance)
- Files:
 - no more header (.h) files
 - each class (MyClass) written in a separate file (MyClass.java)
 - or, several classes in a single file, but only one of them must be public
 - to run a program – the public class must contain a **main** function

An Example

- **ConsCell** - a class for building linked lists of integers

```
/**
 * A ConsCell is an element in a linked list of
 * ints.
 */
public class ConsCell {
    private int head; // the first item in the list
    private ConsCell tail; // rest of the list, or null

    /**
     * Construct a new ConsCell given its head and tail.
     * @param h the int contents of this cell
     * @param t the next ConsCell in the list, or null
     */
    public ConsCell(int h, ConsCell t) {
        head = h;
        tail = t;
    }
}
```

An Example

```
/**
 * Accessor for the head of this ConsCell.
 * @return the int contents of this cell
 */
public int getHead() {
    return head;
}

/**
 * Accessor for the tail of this ConsCell.
 * @return the next ConsCell in the list, or null
 */
public ConsCell getTail() {
    return tail;
}
}
```

An Example

IntList – the list of integers

```
/**
 * An IntList is a list of ints.
 */
public class IntList {
    private ConsCell start; // list head, or null

    /**
     * Construct a new IntList given its first ConsCell.
     * @param s the first ConsCell in the list, or null
     */
    public IntList(ConsCell s) {
        start = s;
    }
}
```

An Example

```
/**
 * Cons the given element h onto us and return the
 * resulting IntList.
 * @param h the head int for the new list
 * @return the IntList with head h, and us as tail
 */
public IntList cons (int h) {
    return new IntList(new ConsCell(h,start));
}
```

An **IntList** knows how to cons things onto itself. It does not change, but it returns a new **IntList** with the new element at the front.

An Example

```
/**
 * Get our length.
 * @return our int length
 */
public int length() {
    int len = 0;
    ConsCell cell = start;
    while (cell != null) { // while not at end of list
        len++;
        cell = cell.getTail();
    }
    return len;
}
```

An **IntList** knows how to compute its length

An Example

```
/**
 * Print ourselves to System.out.
 */
public void print() {
    System.out.print("[");           // print to standard output
    ConsCell a = start;
    while (a != null) {
        System.out.print(a.getHead());
        a = a.getTail();
        if (a != null) System.out.print(",");
    }
    System.out.println("]");
}
}
```

An **IntList** knows how to print itself

An Example

- The “main” class:

```
public class Driver {  
    public static void main(String[] args) {  
        IntList a = new IntList(null);  
        IntList b = a.cons(2);  
        IntList c = b.cons(1);  
        int x = a.length() + b.length() + c.length();  
        a.print();  
        b.print();  
        c.print();  
        System.out.println(x);  
    }  
}
```

Compiling the Program

- Three classes to compile, in three files:
 - `ConsCell.java`
 - `IntList.java`
 - `Driver.java`
- Watch capitalization!
- Compile with the command **javac**:
 - They can be done one at a time
 - Or, `javac Driver.java` gets them all
- The compiler produces **.class** files (contain byte code)

Running The Program

- Use the **java** command to run the **main** method in a **.class** file:

```
C:\demo>java Driver  
[ ]  
[2]  
[1,2]  
3
```

Generating Documentation

- Use the **javadoc** command to automatically generate documentation (in HTML format) for your classes
- Documentation is based on “special” comments:
 - between `/**` and `*/`
 - “keywords” that introduce specific descriptions (`@param`, `@return`, etc)

```
/**
 * Cons the given element h onto us and return the
 * resulting IntList.
 * @param h the head int for the new list
 * @return the IntList with head h, and us as tail
 */
public IntList cons (int h) {
    return new IntList(new ConsCell(h,start));
}
```

Generating Documentation

- Example (partial view):

Method Summary	
<code>IntList</code>	<code>cons</code> (<code>int h</code>) Cons the given element <code>h</code> onto us and return the resulting <code>IntList</code> .
<code>int</code>	<code>length</code> () Get our length.
<code>void</code>	<code>print</code> () Print ourself to <code>System.out</code> .

Methods inherited from class <code>java.lang.Object</code>
<code>clone</code> , <code>equals</code> , <code>finalize</code> , <code>getClass</code> , <code>hashCode</code> , <code>notify</code> , <code>notifyAll</code> , <code>toString</code> , <code>wait</code> , <code>wait</code> , <code>wait</code>

Constructor Detail

`IntList`

```
public IntList(ConsCell s)
```

Construct a new `IntList` given its first `ConsCell`.

Parameters:

`s` - the first `ConsCell` in the list, or null

Method Detail

`cons`

```
public IntList cons(int h)
```

Cons the given element `h` onto us and return the resulting `IntList`.

Parameters:

`h` - the head `int` for the new list

Returns:

the `IntList` with head `h`, and us as tail

Interfaces

- An **interface** in Java is a collection of method prototypes (just header, no body)

```
public interface Drawable {  
    void show(int xPos, int yPos);  
    void hide();  
}
```

- A class can declare that it **implements** a particular interface
- Then it must provide **public** method definitions that match those in the interface

Interfaces

```
public class Icon implements Drawable {  
    public void show(int x, int y) {  
        ... method body ...  
    }  
    public void hide() {  
        ... method body ...  
    }  
    ...more methods and fields...  
}
```

```
public class Square implements Drawable, Scalable {  
    ... all required methods of all interfaces implemented ...  
}
```


Interfaces

- An interface can be implemented by many classes:

```
public class Window implements Drawable ...  
public class MousePointer implements Drawable ...  
public class Oval implements Drawable ...
```

- An interface name can be used as a (polymorphic) type:

```
Drawable d;  
d = new Icon("i1.gif");  
d.show(0,0);  
d = new Oval(20,30);  
d.show(0,0);
```

Polymorphism with Interfaces

```
static void flashoff(Drawable d, int k) {  
    for (int i = 0; i < k; i++) {  
        d.show(0,0);  
        d.hide();  
    }  
}
```

- Class of object referred to by **d** is not known at compile time
- It is some class that **implements Drawable**, so it has **show** and **hide** methods that can be called

A More Complete Example

- A **Worklist** interface for a collection of **String** objects
- Can be added to, removed from, and tested for emptiness

```
public interface Worklist {  
    /**  
     * Add one String to the worklist.  
     * @param item the String to add  
     */  
    void add(String item);  
  
    /**  
     * Test whether there are more elements in the  
     * worklist: that is, test whether more elements  
     * have been added than have been removed.  
     * @return true iff there are more elements  
     */  
    boolean hasMore();  
}
```

A More Complete Example

```
/**
 * Remove one String from the worklist and return
 * it.  There must be at least one element in the
 * worklist.
 * @return the String item removed
 */
String remove();
}
```

- **Worklist** interface does not specify ordering: could be a stack, a queue, or something else
- We will do an implementation as a **stack**, implemented using linked lists

A More Complete Example

- The Node class:

```
/**
 * A Node is an object that holds a String and a link
 * to the next Node. It can be used to build linked
 * lists of Strings.
 */
public class Node {
    private String data; // Each node has a String...
    private Node link;   // and a link to the next Node

    /**
     * Node constructor.
     * @param theData the String to store in this Node
     * @param theLink a link to the next Node
     */
    public Node(String theData, Node theLink) {
        data = theData;
        link = theLink;
    }
}
```

A More Complete Example

```
/**
 * Accessor for the String data stored in this Node.
 * @return our String item
 */
public String getData() {
    return data;
}

/**
 * Accessor for the link to the next Node.
 * @return the next Node
 */
public Node getLink() {
    return link;
}
}
```

A More Complete Example

- The Stack class:

```
/**
 * A Stack is an object that holds a collection of
 * Strings.
 */
public class Stack implements Worklist {
    protected Node top = null; // top Node in the stack

    /**
     * Push a String on top of this stack.
     * @param data the String to add
     */
    public void add(String data) {
        top = new Node(data, top);
    }
}
```

A More Complete Example

```
/**
 * Test whether this stack has more elements.
 * @return true if this stack is not empty
 */
public boolean hasMore() {
    return (top!=null);
}

/**
 * Pop the top String from this stack and return it.
 * This should be called only if the stack is
 * not empty.
 * @return the popped String
 */
public String remove() {
    Node n = top;
    top = n.getLink();
    return n.getData();
}
}
```


A Test

```
Worklist w;  
w = new Stack();  
w.add("- repeat");  
w.add("- sleep");  
w.add("- goto school ");  
w.add("- wake up ");  
System.out.print(w.remove());  
System.out.print(w.remove());  
System.out.print(w.remove());  
System.out.println(w.remove());
```

- Output:
 - wake up - goto school - sleep - repeat
- Other implementations of **Worklist** are possible: **Queue**, **PriorityQueue**, etc.

Derived Classes

- One class can be derived from another, using the keyword **extends**
- The class **PeekableStack** – just like **Stack**, but also has a method **peek** to examine the top element without removing it

```
public class PeekableStack extends Stack {  
  
    public String peek() {  
        return top.getData();  
    }  
}
```

- What is the difference between **extend** and **implement**?
 - **Extend** a class – inherit all its fields and methods
 - **Implement** an interface – not inherit anything, just get an obligation to implement its methods

Inheritance Chains

- If you do not give an **extends** clause, Java supplies one:
extends Object
- All classes are derived, directly or indirectly, from the predefined class **Object** (except **Object** itself)
- All classes inherit methods from **Object**:
 - **toString**, for converting to a **String**
 - **equals**, for comparing with other objects
 - etc.
- No multiple inheritance
- However, can implement multiple interfaces

Object Oriented Design

- How should we establish inheritance relations?
 - usually, inheritance is one useful class extending another
 - what about this:

```
public class Label {
    private int x,y;
    private int width;
    private int height;
    private String text;
    public void move
        (int newX, int newY)
    {
        x = newX;
        y = newY;
    }
    public String getText()
    {
        return text;
    }
}
```

```
public class Icon {
    private int x,y;
    private int width;
    private int height;
    private Gif image;
    public void move
        (int newX, int newY)
    {
        x = newX;
        y = newY;
    }
    public Gif getImage()
    {
        return image;
    }
}
```

Object Oriented Design

```
public class Graphic {  
    protected int x,y;  
    protected int width,height;  
    public void move(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
}
```

```
public class Label  
    extends Graphic {  
    private String text;  
    public String getText()  
    {  
        return text;  
    }  
}
```

```
public class Icon  
    extends Graphic {  
    private Gif image;  
    public Gif getImage()  
    {  
        return image;  
    }  
}
```

- Here - factor out common code from different classes into a shared base class

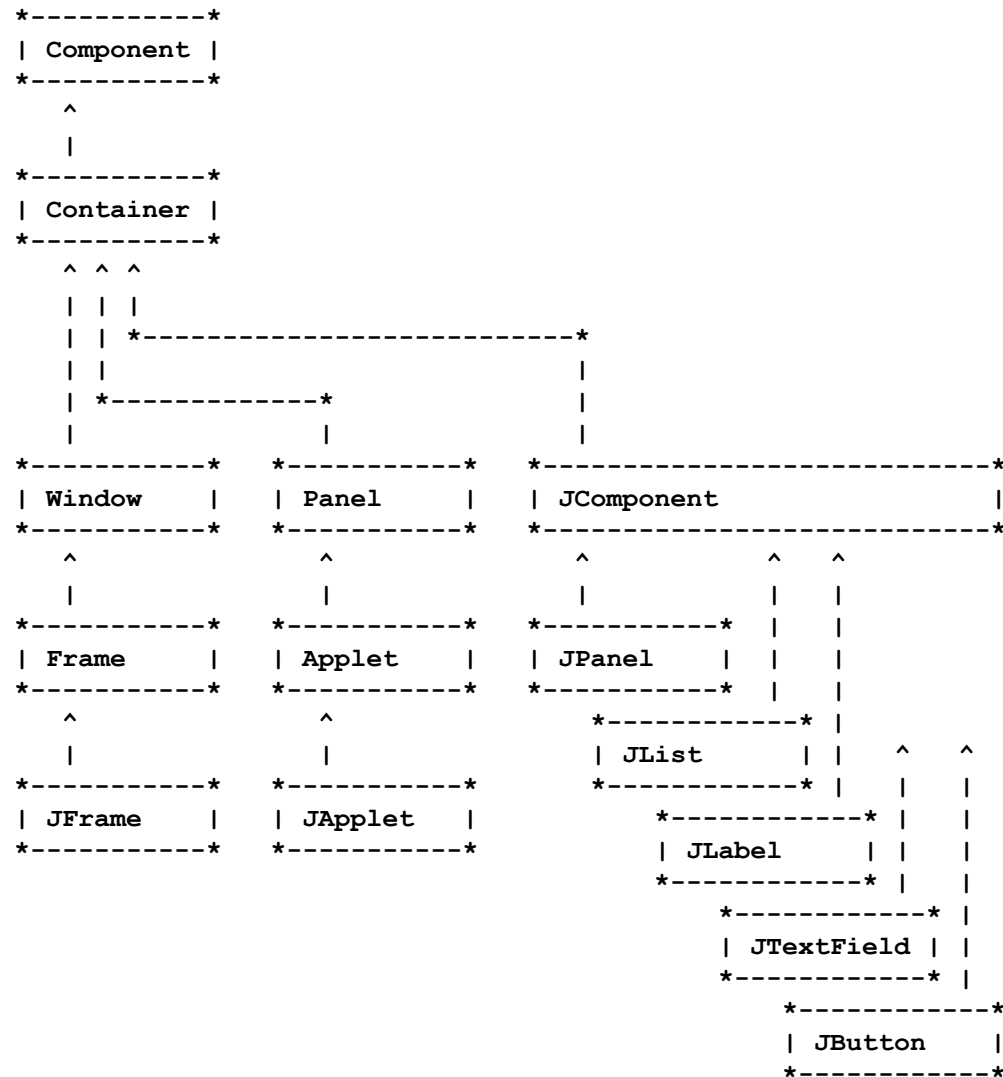
Object Oriented Design

- In general:
- When you write the same statements repeatedly
→ that should be a method
- When you write the same methods repeatedly
→ that should be a common base class
- Difficulty – see the need for a shared base class **early in the design**, before writing a lot of code that needs to be reorganized

GUIs

- General mechanisms for GUI development:
- **Components** – classes that have a graphical representation
 - buttons, text fields, labels, choice lists, ...
- **Containers** – components that allow other components to nest within their boundaries
 - windows, frames, panels, ...
- Two GUI packages:
 - **Abstract Window Toolkit (AWT)**: `java.awt`
 - **Classes**: `Frame`, `Button`, `TextField`, `Label`
 - **Swing**: `javax.swing` – newer than AWT, improved
 - **Classes**: `JFrame`, `JButton`, `JTextField`, `JLabel`

Class Hierarchy



Windows

- Create a window:

```
import javax.swing.*;
```

```
public class WindowApplication
```

```
{
```

```
    public static void main (String argv [])
```

```
    {
```

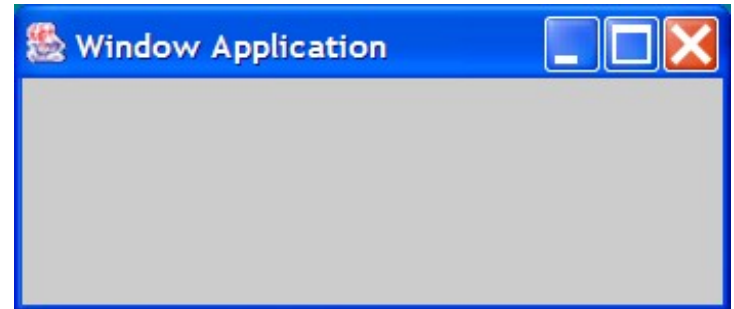
```
        JFrame frame = new JFrame("Window Application");
```

```
        frame.setSize(350, 150);
```

```
        frame.setVisible(true);
```

```
    }
```

```
}
```



Windows

- Anything wrong?
 - when closing the window, the program will not finish (need to kill it)
- Must explicitly end the program upon closing the window
- **Event-driven programming**
 - the language implementation generates **events** – user actions
 - the programmer can define **event handlers** – methods that specify what to do when a particular event occurs
- In Java – must implement **listeners**, for various types of events:
 - WindowListener
 - ActionListener
 - MouseListener
 - ListSelectionListener

Windows

- The **window listener** (implement an interface):

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

public class WindowDestroyer implements WindowListener
{
    public void windowClosing(WindowEvent e)
    {    System.exit(0);    }
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
}
```

Windows

- The new application:

```
import javax.swing.*;

public class WindowApplication
{
    public static void main (String argv [])
    {
        JFrame frame = new JFrame("Window Application");
        frame.setSize(350, 150);
        frame.addWindowListener(new WindowDestroyer());
        frame.setVisible(true);
    }
}
```

Windows

- Can do better – use a **window adapter** (extend a class)
- Other changes:
 - have the application itself be the window
 - use an inner class (defined within another class)

```
import javax.swing.*;
import java.awt.event.*;

public class WindowApplication extends JFrame
{
    public static void main (String argv [])
    {
        new WindowApplication("Window Application");
    }
}
```

Windows

```
public WindowApplication(String title)
{
    super(title);    // call constructor of base class
    setSize(350, 150);
    addWindowListener(new WindowDestroyer());
    setVisible(true);
}

// Define window adapter
private class WindowDestroyer extends WindowAdapter
{
    // implement only the function that you want
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
}
```

Components

- Components
 - labels, buttons, text fields, selection lists, ...
 - added to the **content pane** (the client area of a window)
- How will they be arranged?
 - defined by the **layout manager** – their size/position automatically change with window size
 - defined by explicitly specifying their size/position (with **setBounds**)
- Types of layout managers:
 - BorderLayout
 - GridLayout
 - FlowLayout
 - GridBagLayout
- GridLayout
 - usually best tradeoff between simplicity and flexibility
 - specifies the number of rows and columns
 - components are added in order (left-right, top-bottom)
 - all components have the same size

Labels

- Add six **labels**:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class WindowApplication extends JFrame
{
    public static void main(String argv [])
    {
        new WindowApplication("Window Application");
    }
}
```


Labels

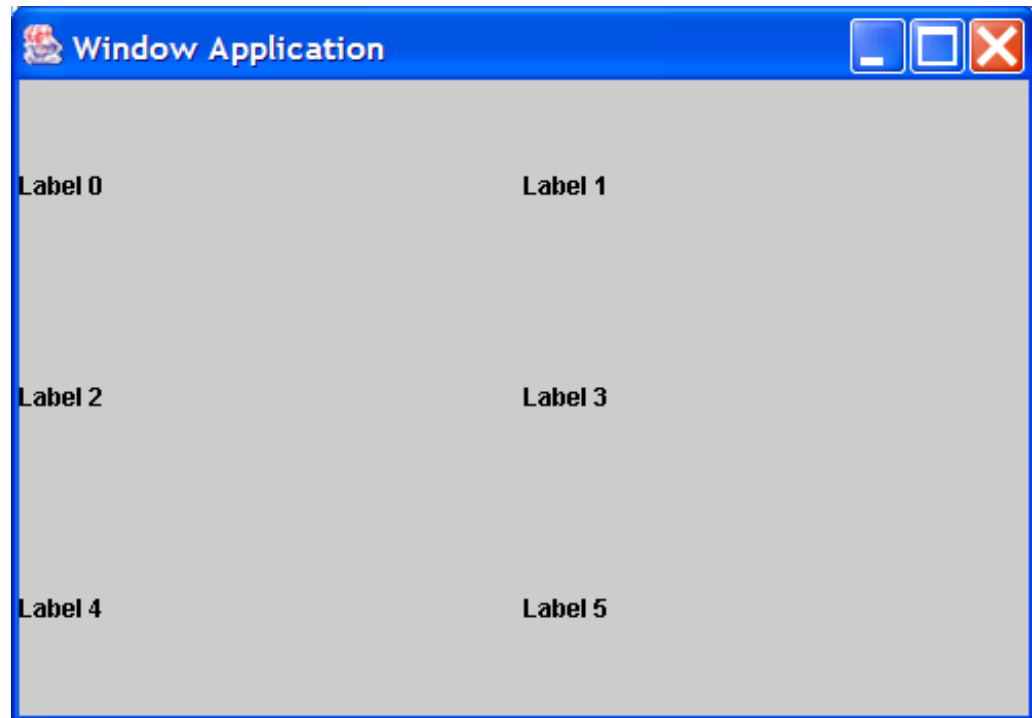
```
public WindowApplication(String title)
{
    super(title);
    setBounds(100, 100, 500, 350);
    addWindowListener(new WindowDestroyer());

    getContentPane().setLayout(new GridLayout(3, 2));
    int i;
    for (i = 0; i < 6; i++)
    {
        getContentPane().add(new JLabel("Label " + i));
    }

    setVisible(true);
}
```

Labels

```
// Define window adapter
private class WindowDestroyer extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
}
```



Buttons and Text Fields

- Add **buttons** and **text fields**:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class WindowApplication extends JFrame
{
    public static void main(String argv [])
    {
        new WindowApplication("Window Application");
    }
}
```

Buttons and Text Fields

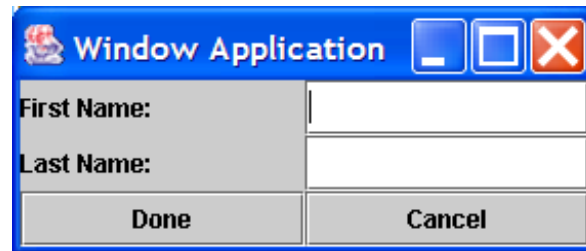
```
public WindowApplication(String title)
{
    super(title);
    setBounds(100, 100, 250, 100);
    addWindowListener(new WindowDestroyer());

    getContentPane().setLayout(new GridLayout(3, 2));
    getContentPane().add(new JLabel("First Name:"));
    getContentPane().add(new JTextField(""));
    getContentPane().add(new JLabel("Last Name:"));
    getContentPane().add(new JTextField(""));
    getContentPane().add(new JButton("Done"));
    getContentPane().add(new JButton("Cancel"));

    setVisible(true);
}
```

Buttons and Text Fields

```
// Define window adapter
private class WindowDestroyer extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
}
```



Buttons and Text Fields

- Handle user input
 - implement an **action listener**
- Also – **get** and **set** text in the text fields

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class WindowApplication extends JFrame
{
    protected JButton buttonDone;
    protected JButton buttonCancel;
    protected JTextField tfFirstName;
    protected JTextField tfLastName;

    public static void main(String argv [])
    {
        new WindowApplication("Window Application");
    }
}
```

Buttons and Text Fields

```
public WindowApplication(String title)
{
    super(title);
    setBounds(100, 100, 250, 100);
    addWindowListener(new WindowDestroyer());

    buttonDone = new JButton("Done");
    buttonCancel = new JButton("Cancel");
    buttonDone.addActionListener(new ActionListener());
    buttonCancel.addActionListener(new ActionListener());

    tfFirstName = new JTextField("");
    tfLastName = new JTextField("");
}
```

Buttons and Text Fields

```
getContentPane().setLayout(new GridLayout(3, 2));
```

```
getContentPane().add(new JLabel("First Name:"));
```

```
getContentPane().add(tfFirstName);
```

```
getContentPane().add(new JLabel("Last Name:"));
```

```
getContentPane().add(tfLastName);
```

```
getContentPane().add(buttonDone);
```

```
getContentPane().add(buttonCancel);
```

```
setVisible(true);
```

```
tfFirstName.setText("John");
```

```
tfLastName.setText("Doe");
```

```
}
```


Buttons and Text Fields

```
// Define action listener
private class ActionHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        if ( e.getSource() == buttonDone )
        {
            String s1 = tfFirstName.getText();
            String s2 = tfLastName.getText();
            System.out.println("Full name: " + s1 + " "
                               + s2);
        }
        else if ( e.getSource() == buttonCancel )
            System.out.println("You pressed the Cancel
                               button.");
    }
}
```

Buttons and Text Fields

```
// Define window adapter
private class WindowDestroyer extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
```

Selection Lists

- Add a **selection list** (with a scroll-bar)
- Also – implement a **list selection listener**

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

public class WindowApplication extends JFrame
{
    protected JList listMovies;

    public static void main(String argv [])
    {
        new WindowApplication("Window Application");
    }
}
```

Selection Lists

```
public WindowApplication(String title)
{
    super(title);
    setBounds(100, 100, 200, 100);
    addWindowListener(new WindowDestroyer());

    listMovies = new JList();
    listMovies.addListSelectionListener(new ListHandler());

    getContentPane().setLayout(new GridLayout(1, 1));

    // getContentPane().add(listMovies);      // no scroll
    getContentPane().add(new JScrollPane(listMovies));

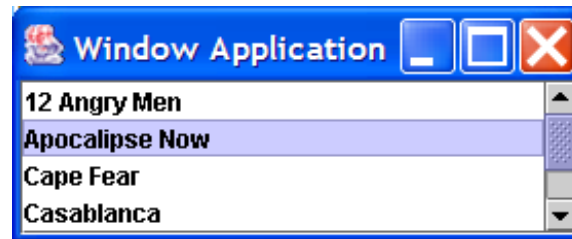
    setVisible(true);
}
```

Selection Lists

```
String movies[] = {"12 Angry Men", "Apocalypse Now",  
                  "Cape Fear", "Casablanca", "Fargo",  
                  "Solaris"};  
listMovies.setListData(movies);  
  
}  
  
// Define window adapter  
private class WindowDestroyer extends WindowAdapter  
{  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(0);  
    }  
}
```

Selection Lists

```
// Define list listener
private class ListHandler implements ListSelectionListener
{
    public void valueChanged(ListSelectionEvent e)
    {
        if ( e.getSource() == listMovies )
            if ( !e.getValueIsAdjusting() )
            {
                int i = listMovies.getSelectedIndex();
                String s = (String) listMovies.getSelectedValue();
                System.out.println("Position " + i + " selected: "
+s);
            }
    }
}
```



Menus

- Add a **menu**
- How can we handle user input?
 - menu selection is handled by an **action listener**

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class WindowApplication extends JFrame
{
    protected JButton buttonDone;
    protected JButton buttonCancel;
    protected JTextField tfFirstName;
    protected JTextField tfLastName;
    protected JMenuBar mb;
    protected Jmenu m;
    protected JMenuItem mi[];
```

Menus

```
public static void main(String argv [])
{
    new WindowApplication("Window Application");
}

public WindowApplication(String title)
{
    super(title);
    addWindowListener(new WindowDestroyer());

    buttonDone = new JButton("Done");
    buttonCancel = new JButton("Cancel");
    buttonDone.addActionListener(new ActionListener());
    buttonCancel.addActionListener(new ActionListener());

    tfFirstName = new JTextField("");
    tfLastName = new JTextField("");
```


Menus

```
mb = new JMenuBar();
m = new JMenu("Actions");
mi = new JMenuItem[2];
mi[0] = new JMenuItem("Clear");
mi[0].addActionListener(new ActionListener());
mi[1] = new JMenuItem("Exit");
mi[1].addActionListener(new ActionListener());
m.add(mi[0]);
m.add(new JSeparator());
m.add(mi[1]);
mb.add(m);
setJMenuBar(mb);

getContentPane().setLayout(new GridLayout(3, 2));
```

Menus

```
getContentPane().add(new JLabel("First Name:"));
getContentPane().add(tfFirstName);
getContentPane().add(new JLabel("Last Name:"));
getContentPane().add(tfLastName);
getContentPane().add(buttonDone);
getContentPane().add(buttonCancel);

setBounds(100, 100, 250, 150);
setVisible(true);

tfFirstName.setText("John");
tfLastName.setText("Doe");
}

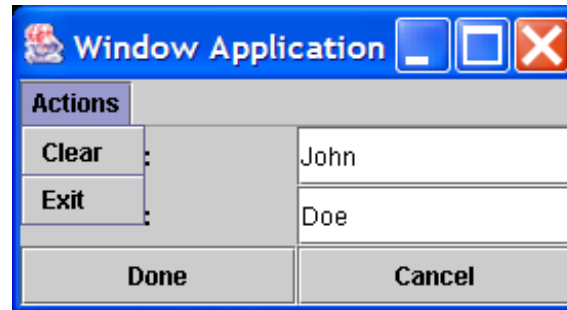
// Define window adapter
private class WindowDestroyer extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
```

Menus

```
// Define action listener
private class ActionHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        if ( e.getSource() == buttonDone )
        {
            String s1 = tfFirstName.getText();
            String s2 = tfLastName.getText();
            System.out.println("Full name: " + s1 + " " + s2);
        }
        else if ( e.getSource() == buttonCancel )
            System.out.println("You pressed the Cancel button.");
    }
}
```

Menus

```
else if ( e.getSource() == mi[0] )           // clear
{
    tfFirstName.setText("");
    tfLastName.setText("");
}
else if ( e.getSource() == mi[1] )           // exit
    System.exit(0);
}
}
```



Drawing

- Draw lines, rectangles, arcs, ovals, ...
- Need to redefine the `paint` method for the component where we draw
 - `paint` is automatically called every time the component needs to be (re)displayed
 - its only parameter is a `Graphics` object – contains current information needed for rendering, such as:
 - clip rectangle (the part of the component that needs to be painted)
 - color
 - font
 - logical pixel operation function (XOR or Paint)
 - for drawing - inside `paint`, call methods of the `Graphics` object such as:
 - `drawRect`
 - `drawLine`
 - `drawOval`
 - ...

Drawing

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class DrawingTester extends JComponent
{
    public void paint(Graphics g)
    {
        Dimension d = getSize();

        g.setColor(Color.yellow);
        g.fillRect(1, 1, d.width-2, d.height-2);

        g.setColor(Color.black);
        g.drawRect(1, 1, d.width-2, d.height-2);
        g.drawLine(1, d.height-1, d.width-1, 1);
        g.drawOval(d.width/2 - 30, d.height/2 - 30, 60, 60);
    }
}
```

Drawing

```
public class WindowApplication extends JFrame
{
    protected DrawingTester drawTest;
    protected JLabel labelX;
    protected JLabel labelY;
    protected JTextField tfX;
    protected JTextField tfY;

    public static void main(String argv [])
    {
        new WindowApplication("Window Application");
    }
    public WindowApplication(String title)
    {
        super(title);
        setBounds(100, 100, 300, 300);
        addWindowListener(new WindowDestroyer());
    }
}
```

Drawing

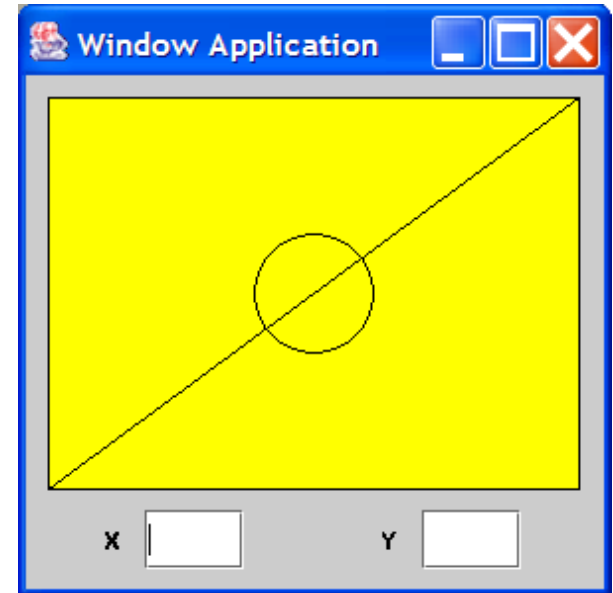
```
drawTest = new DrawingTester();  
labelX = new JLabel("X");  
labelY = new JLabel("Y");  
tfX = new JTextField("");  
tfY = new JTextField("");  
  
// let's also specify the arrangement of components by hand  
getContentPane().setLayout(null);  
  
getContentPane().add(drawTest);  
getContentPane().add(labelX);  
getContentPane().add(labelY);  
getContentPane().add(tfX);  
getContentPane().add(tfY);
```


Drawing

```
drawTest.setBounds(10, 10, 270, 200);  
labelX.setBounds(40, 220, 20, 30);  
tfX.setBounds(60, 220, 50, 30);  
labelY.setBounds(180, 220, 20, 30);  
tfY.setBounds(200, 220, 50, 30);
```

```
setVisible(true);  
}
```

```
// Define window adapter  
private class WindowDestroyer extends WindowAdapter  
{  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(0);  
    }  
}
```



Mouse Events

- Perform actions upon mouse events
 - implement a [mouse listener](#)
- Draw a small circle where the user clicks
 - need to make the component redraw itself – call [repaint](#)

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.event.*;
```

```
class DrawingTester extends JComponent implements MouseListener  
{  
    WindowApplication app;  
    int    x = -1;  
    int    y = -1;
```

Mouse Events

```
public DrawingTester(WindowApplication a)
{
    app = a;
}

public void paint(Graphics g)
{
    Dimension d = getSize();

    g.setColor(Color.yellow);
    g.fillRect(1, 1, d.width-2, d.height-2);

    g.setColor(Color.black);
    g.drawRect(1, 1, d.width-2, d.height-2);
    g.drawLine(1, d.height-1, d.width-1, 1);
    g.drawOval(d.width/2 - 30, d.height/2 - 30, 60, 60);
    if ( x >= 0 && y >= 0 )
        g.drawOval(x-10, y-10, 20, 20);
}
```

Mouse Events

```
public void mouseClicked(MouseEvent e)
{
    x = e.getX();
    y = e.getY();
    repaint();

    // also display click coordinates in the two text fields
    app.tfX.setText(String.valueOf(x));
    app.tfY.setText(String.valueOf(y));
}

public void mouseEntered(MouseEvent e)        {}
public void mouseExited(MouseEvent e) {}
public void mousePressed(MouseEvent e)        {}
public void mouseReleased(MouseEvent e)       {}
}
```

Mouse Events

```
public class WindowApplication extends JFrame
{
    protected DrawingTester drawTest;
    protected JLabel labelX;
    protected JLabel labelY;
    protected JTextField tfX;
    protected JTextField tfY;

    public static void main(String argv [])
    {
        new WindowApplication("Window Application");
    }
    public WindowApplication(String title)
    {
        super(title);
        setBounds(100, 100, 300, 300);
        addWindowListener(new WindowDestroyer());
    }
}
```

Mouse Events

```
drawTest = new DrawingTester(this);
labelX = new JLabel("X");
labelY = new JLabel("Y");
tfX = new JTextField("");
tfY = new JTextField("");

getContentPane().setLayout(null);
getContentPane().add(drawTest);
getContentPane().add(labelX);
getContentPane().add(labelY);
getContentPane().add(tfX);
getContentPane().add(tfY);

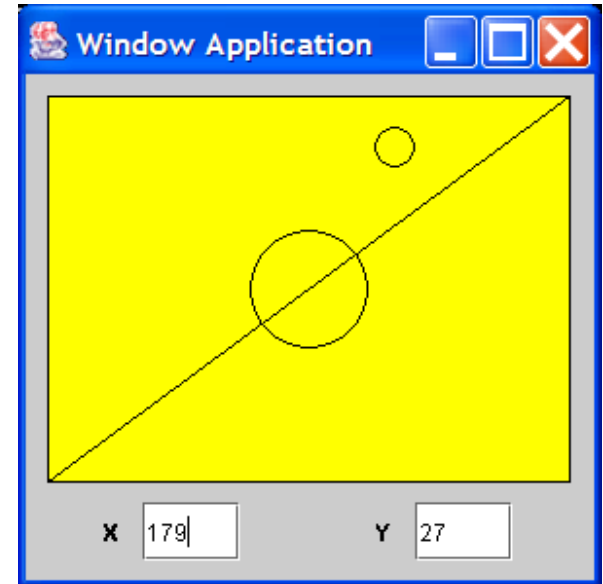
drawTest.setBounds(10, 10, 270, 200);
labelX.setBounds(40, 220, 20, 30);
tfX.setBounds(60, 220, 50, 30);
labelY.setBounds(180, 220, 20, 30);
tfY.setBounds(200, 220, 50, 30);
```

Mouse Events

```
drawTest.addMouseListener(drawTest);
```

```
setVisible(true);  
}
```

```
// Define window adapter  
private class WindowDestroyer extends WindowAdapter  
{  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(0);  
    }  
}
```



File I/O

- **Reading** from a **text file** – two main approaches:

File input stream

|

| Bytes

v

Input stream reader

|

| Characters

v

Buffered reader

|

| Lines

v

Your program

OR

\

\ Characters

v

Stream tokenizer

|

| Tokens (numbers, strings)

v

Your program

File I/O

- Read and print (to console) the data in the following file:

Jack	18	3.7
Mary	15	3.8
Jim	12	2.5

```
import java.io.*;
```

```
public class FileIODemo  
{
```

```
    public static void main(String argv[]) throws IOException  
    {
```

```
        FileInputStream stream = new FileInputStream("input.txt");
```

```
        InputStreamReader reader = new InputStreamReader(stream);
```

```
        StreamTokenizer tokens = new StreamTokenizer(reader);
```

File I/O

```
String s;
int     n;
float   f;
while (tokens.nextToken() != tokens.TT_EOF)
{
    s = (String) tokens.sval;
    tokens.nextToken();
    n = (int) tokens.nval;
    tokens.nextToken();
    f = (float) tokens.nval;

    System.out.println(s + " " + n + " " + f);
}

stream.close();
}
}
```

File I/O

- Writing to a text file – use a `FileOutputStream` and a `PrintWriter`

```
import java.io.*;

public class FileIODemo
{
    public static void main(String argv[]) throws IOException
    {
        FileInputStream istream = new FileInputStream("input.txt");
        InputStreamReader reader = new InputStreamReader(istream);
        StreamTokenizer tokens = new StreamTokenizer(reader);

        FileOutputStream ostream = new FileOutputStream("output.txt");
        PrintWriter writer = new PrintWriter(ostream);
```

File I/O

```
String s;
int n;
float f;
while (tokens.nextToken() != tokens.TT_EOF)
{
    s = (String) tokens.sval;
    tokens.nextToken();
    n = (int) tokens.nval;
    tokens.nextToken();
    f = (float) tokens.nval;

    writer.println(s + " " + n + " " + f);
}
writer.flush();
istream.close();
ostream.close();
System.out.println("File written.");
}
}
```

Announcements

- Readings
 - Java resources