Comp 585 Graphical User Interfaces
Noteset #1

GUI Design Principles

Traditional Monolithic Flow of Control Programming

- User launches application
- Application obtains inputs
- Application runs to completion and outputs results with no interaction with user

CLI: Command-Line Interface (or Interpreter)

- User Launches Application
- Application Begins Command Interpreter Loop
- User Enters Command Plus Related Data
- Application Computes Result, Outputs, Requests Next Command
- Application Terminates When Next Command Equals "Quit"

Event-Driven Programming

- User Launches Application
- Application Creates GUI
- User Interaction with GUI (via Keyboard and Mouse) Generates Events
- Application Is Organized into Event Handlers (callbacks) Triggered by Events
  - Mouse Click on Button
  - Text Entered into Text Field
  - Scroll Bar Clicked/Dragged/Released
  - …
- Application Terminates when "Exit" Menu Item Selected or Main Window is Closed
  - Event Handler for "Quit" Executes the Exit Code

In a multithreaded GUI application, the event thread acts as an invisible "blinking cursor", waiting for the user to "do something" on the GUI.

GUIs and Software Engineering

- Design of Application Should Be Modular, Not Monolithic
- Conceptually, Both CLI and GUI are Interchangeable Layers
- User Commands Replaced with Events

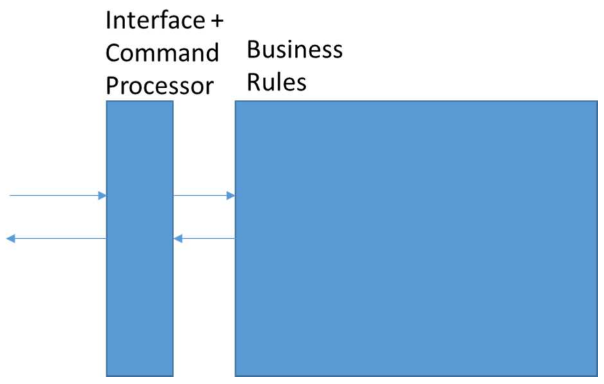
GUIs and OOP

- Both CLI and GUI Require a Modular Design of Underlying App
  - Consistent With OOD, although OOD not required
- GUI Is Naturally Described as a Collection of Interacting Objects
  - Windows, Menus, Buttons, Lists, …
- Smalltalk (Xerox PARC) introduced the term OOP to describe a collection of existing ideas
  - Message Passing from the Simula Programming Language
  - Class, object, method, message passing, encapsulation, inheritance, etc.
- Xerox PARC also introduced PUI (PARC User Interface)
  - Screen Widgets Plus Pointing Device ("x-y position indicator")
- Mass Marketed First By Apple, Then By Microsoft
- OOP has its critics, but it dramatically simplifies the presentation of GUI design
- Some Early GUI APIs Made Poor Use of OOP Concepts
  - Global Objects Configured by Huge Number of Global Functions
  - C++ and the MFC
- Later GUI APIs Follow a Much Improved Object Model
  - Java and Swing (JFC), C# and Windows Forms, WPF

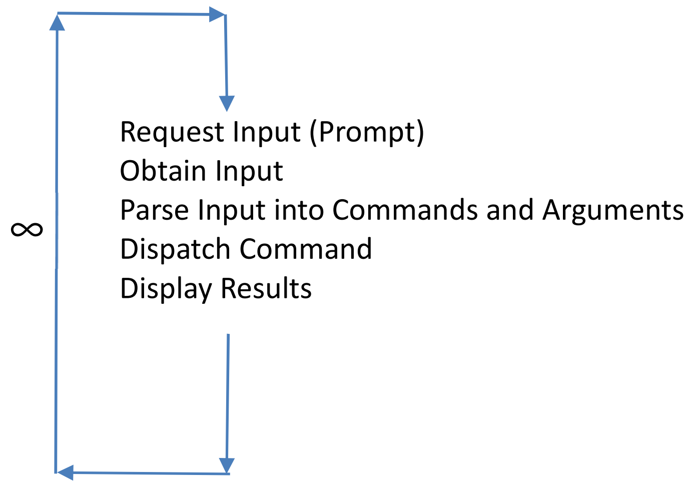
GUIs and Software Architecture
A software app performs a set of related operations on user data to convert inputs to outputs.

- Numbers + arithmetic operations ➔ calculator ➔ results
- Document + edit commands ➔ word processor ➔ Updated document

Every app needs an interface as part of the overall software architecture. The interface should be modular, in the sense that one interface can be substituted for another with little or no impact on the remainder of the app.

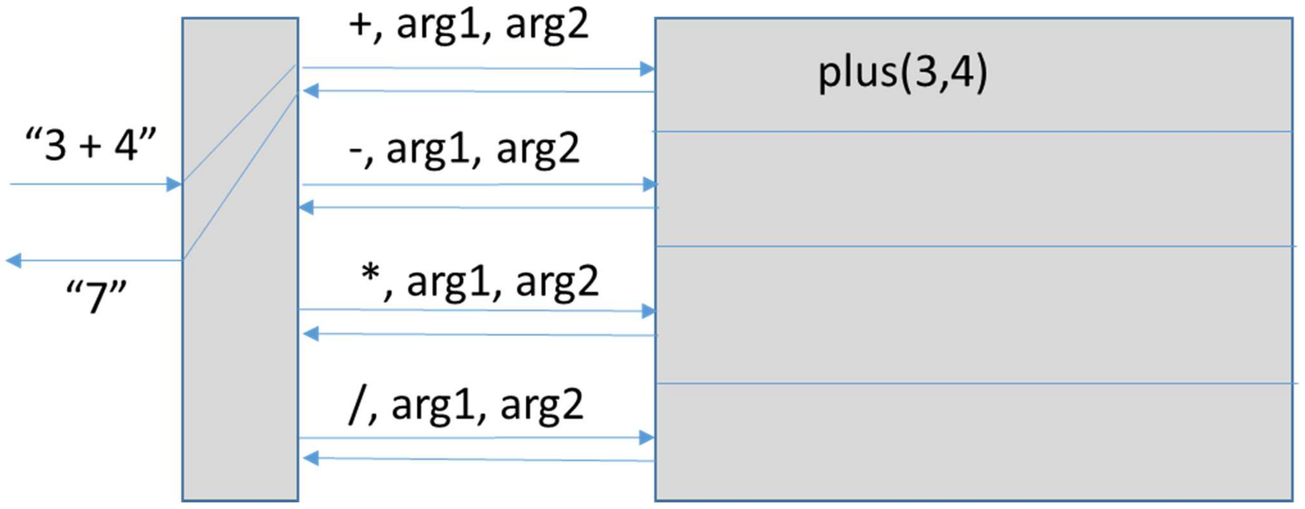


Complexity of interface depends on what services the app provides to the user. The simplest is a command line interface or command line processor:



There has to be a trigger to indicate that the input has been captured and is ready to be processed. On a command line interface, the trigger is usually pressing the "Enter" key. On a GUI, the trigger is some user interaction such as pointing and clicking a button.

Some apps break down into a collection of individual functions that don't interact:

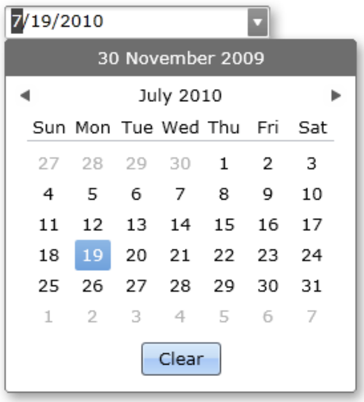


Other apps implement a state machine to walk the user through a complex sequence of inputs.

Book a flight …
- Enter your departure and destination airports …
- Enter the date you are traveling …
  - Searching …
- Order flights by (1) cheapest to most expensive, (2) least to most number of stops …
- Pick a flight …
  - Seat selection …
- Payment info …

Book another flight …

Inputting Structured/Constrained Data
Inputs like times/dates are highly constrained. User interface should constrain inputs to legal values rather than free form entry of text followed by validation. Imagine a text box for date entry that allows the user to type any text, compared to a "date picker" that displays an actual calendar, the user is constrained to pick only dates that exist by clicking on that date's position in the calendar



Or imagine a user interface that allows the user to enter a value that happens to have a valid range of 1 to 10, based on some physical constraint (for example, the input will drive a physical device with physical limits). Which is the better GUI approach:

- A: A text field that allows the user to type anything, after which the input is validated. If an illegal value was entered, an error box pops up and tells the user to try again.
- B: A slider with predefined hash marks corresponding to the values 1 through 10. The slider can only be set to one of the legal values. (http://www.embedded.com/2000/0012/0012ia1.htm)

There's some tension between the application developer's desire to constrain what the user inputs vs. the user's desire to not be constrained and enter whatever they feel like.

GUI Design, User Expectations, Industrial Psychology

Users will have better interaction with the GUI if certain visual cues are recognized and followed uniformly. Over time users have developed a set of mental expectations on what a user interface will look like. This impacts the software developer's choice of screen layout, menus, toolbars, icons, dialog boxes, etc. Microsoft has recently started experimenting with "ribbons" as a replacement for menus and toolbars. Touchscreens have expanded the vocabulary of input gestures that an app may need to respond to.

> "The idea of direct manipulation of objects on a screen is integral to the concept of a graphic interface. In fact, the idea of a GUI derives from cognitive psychology, the study of how the brain deals with communication. The idea is that the brain works much more efficiently with graphical icons and displays rather than with words – words add an extra layer of interpretation to the communication process. Imagine if all the road signs you saw were uniform white rectangles, with only the words themselves to differentiate the different commands, warnings, and informational displays. When the "Stop" signs hardly look different from the "Resume Highway Speed" signs, the processing of the signs' messages becomes a slower and more difficult process, and you'd have even more wrecks than you have now." http://www.sitepoint.com/real-history-gui/

Expectations regarding menus have become highly specific:
    File: New, Open, Save, Save As, Close, Exit
    Edit: Cut, Copy, Paste, Undo, Redo, …

For highly structured data like text, users now uniformly expect specific keyboard hotkey bindings regardless of the app:

Ctrl-A	select all
Ctrl-X	Cut to Clipboard
Ctrl-C	Copy to Clipboard
Ctrl-V	Paste from Clipboard
…	

Context sensitive popup menus are expected on a mouse right click, more generally, whenever the "popup trigger" for a specific platform is input

Touch screens support new gestures like swipe left, swipe right, pinch, expand, etc.

If the question is "why does the GUI need to be laid out in *this* way," the answer might be simply to meet user expectations. Most users don't need a user manual to learn basic interaction with the app.

Outline of Java Swing Tutorial
http://docs.oracle.com/javase/tutorial/uiswing/

Trail: Creating a GUI With JFC/Swing, aka The Swing Tutorial

This trail tells you how to create graphical user interfaces (GUIs) for applications and applets, using the Swing components. If you would like to incorporate JavaFX into your Swing application, please see Integrating JavaFX into Swing Applications.

Getting Started with Swing is a quick start lesson. First it gives you a bit of background about Swing. Then it tells you how to compile and run programs that use Swing components.

Learning Swing with the NetBeans IDE is the fastest and easiest way to begin working with Swing. This lesson explores the NetBeans IDE's GUI builder, a powerful feature that lets you visually construct your Graphical User Interfaces.

Using Swing Components tells you how to use each of the Swing components — buttons, tables, text components, and all the rest. It also tells you how to use borders and icons.

Concurrency in Swing discusses concurrency as it applies to Swing programming. Information on the event dispatch thread and the SwingWorker class are included.

Using Other Swing Features tells you how to use actions, timers, and the system tray; how to integrate with the desktop class, how to support assistive technologies, how to print tables and text, how to create a splash screen, and how to use modality in dialogs.

Laying Out Components Within a Container tells you how to choose a layout manager, how to use each of the layout manager classes the Java platform provides, how to use absolute positioning instead of a layout manager, and how to create your own layout manager.

Modifying the Look and Feel tells you how to specify the look and feel of Swing components.

Drag and Drop and Data Transfer tells you what you need to know to implement data transfer in your application.

Writing Event Listeners tells you how to handle events in your programs.
Performing Custom Painting gives you information on painting your own Swing components. It discusses painting issues specific to Swing components, provides an overview of painting concepts, and has examples of custom components that paint themselves.

A good place to start is "Getting Started" … Next take a look at "Using Swing Components", with subheadings "Visual Tour" and "How To"

For all examples, when the text points you to the source code, please take a moment and read the source code.

Exercise
Designing a GUI for an Existing Application

Imagine taking an existing application without a GUI and designing a GUI for it. The simplest GUI components for interaction are:

- Text fields for inputs and results
- Buttons for command triggers

Formula Evaluator

- One text field for each input
- One button for computation of the formula
- One text field for result

Simple Calculator

- Two text fields for each input
- Four buttons for each arithmetic operation
- One text field for result

Desktop Calculator (Integer)

- One text field to implement the display
- Ten buttons to represent each digit
- Four buttons to represent each operation
- One button to represent the equals operation
- One button to represent the clear operation

Telephone Number Database Lookup

- One text field to enter name to look up
- One button to trigger the search operation
- One text field to display the telephone number that holds the result of the search

For most of these applications, you can imagine an initial non-GUI application that is based on a CLI. In the CLI, each user choice may roughly correspond to a method. Similarly, in a GUI version of the application, each button may roughly correspond to a method.

CLI
A menu method or equivalent implements an “infinite” service loop. Application is idle until user input describing the next operation arrives from the keyboard.

The definition of the menu method is under programmer’s control. An if-else dispatcher determines which input has arrived via keyboard input from the user and then dispatches or invokes the corresponding method, then returns to await the next input

GUI
Application uses threads and events to remain idle until “something happens”. For example, a mouse click on a button triggers an event that causes the application to “wake up” and convert inputs to outputs.

The event processing procedure is inherent in the operating system and language runtime. Only the event handlers are under programmer control. In other words, the programmer defines a method to describe the application’s response to some event. The OS and application runtime are responsible for detecting that the event has occurred and invoking the method.

More GUI Design Exercises

Static Counter
Use a text field to display a counter, and an “increment” button to trigger one increment for each click.

Dynamic Counter
Use a text field to display a counter, “start” and “stop” buttons, and a separate thread to continuously increment the counter at a rate of one increment per second. The thread runs when the “start” button is clicked, and pauses when the “stop” button is clicked.

Design Guidelines

Effective GUI design is based on principles at the intersection of computer science, graphic arts, and industrial psychology. In this course, the focus is how to use existing APIs to create traditional GUIs that solve common interface problems. Briefly, here are some guidelines based on the psychology of human-computer interaction that produce user-friendly, satisfying GUIs.

Guidelines

- Expose features of application to the user in an intuitive way (in practice this means proper use of windows, dialogs, documents, files, menus, hot key)
- Let the user take control: don't disable too many functions.
- Limit the number of options at the top level
- Adopt commonly accepted conventions for menu systems whenever practical
- Adopt commonly accepted conventions for dialog boxes (modal and modeless) whenever practical
- Preserve the meaning of common GUI keywords such as "OK", "Cancel", "Cut", "Copy", "Paste", "Save", "Save As … " and don't invent your own quirky or whimsical interpretations.
- Use threads to offload time-consuming tasks to the background and maintain the perception of responsiveness. Use hourglass cursors, progress bars, etc. if time-consuming tasks must run in the foreground.
- Use commonly accepted guidelines for controls or widgets to limit complexity and default values.
- Provide default values and make them preselected so they can be easily modified.
- For GUIs that encompass multiple screens, provide consistency of design across all screens. As in multi-page websites, provide navigational controls and feedback to help the user maintain their bearings. A an equivalent of CSS for widgets.
- Keep in mind the difference in perspective between novice users and power users.
- Maintain GUI state between invocations of the application, so that the GUI returns to the state the user left it in (very common example that many apps screw up: file open/save dialogs which reset to default dir between invocations)
- Interaction with the mouse can be very intuitive, but power users doing "heads-down" data entry can operate faster if they keep their hands on the keyboard. Take advantage of keyboard equivalents -- mnemonics and accelerators. Also, preserve an intuitive tab order for controls on a single page.
- Support Undo/Redo as much as practical. Admittedly, some user actions are not undoable.
- Use GUI modes including enabling/disabling of widgets to guide the user into entering only valid input and avoiding invalid input/operations, but don't overdo it. Too many disabled functions are a negative.
- Use structured widgets for inputting highly structured values (e.g. calendar dates), don't just throw up 3 text boxes with no input validation.
- On the horizon: increasing attention must be given to ADA-compliant interfaces. Currently, compliance is voluntary for most vendors, but will become increasingly mandatory in the future. Example: don't use color alone to communicate important aspects of a GUI.

Overview of API-based GUI Programming

There exist many useful IDEs that allow a GUI developer to implement the GUI graphically, using a drag-and-drop approach, selecting components from a palette. Example: NetBeans for Java, Visual Studio for C++ and C#. In this course, the emphasis is on building GUIs programmatically, which means writing the code manually, referencing the API as necessary to call functions or methods to create components. IDEs are an essential part of any large GUI project, but the projects in this course must be built programmatically.

APIs for constructing GUIs typically use an OOD/OOP approach. Elements of the GUI appear as **objects** which are customized through **property manipulation**. A typical set of objects to build a simple GUI include:

Containers: usually rectangular regions on the screen or desktop that visually group together GUI elements. Also called windows, forms, frames, panels, dialogs, etc. Much of the organization of a complex GUI comes from careful arrangement of multiple containers via nesting, tiling, etc.

Components: individual elements that appear within some container, and which provide an I/O function. Also called widgets or controls. Examples include text boxes or fields, push buttons, radio buttons, check boxes, lists, scroll bars, menu systems (menu bar, menus, menu items, pop-up menus, walking menus, etc.), tables, trees.

Other Software "Agents": behind-the-scenes objects that help manage the look and behavior of the GUI. Examples include **layout managers** and **event listeners**.

After the design phase of the GUI is completed, the code to build the GUI is written, following common OOP guidelines. Objects are instantiated, customized and linked together. The early stages of this process are analogous to building a bookcase: insert tab A into slot 3, etc.

Layout Issues

A big source of potential complexity in GUI coding is how to handle layout issues. In a GUI with a **static layout**, GUI elements are positioned with respect to an x-y coordinate grid within the display area of some container (also called **absolute positioning**). The usual assumption is that once the components have been laid out within the container, their relative positioning will not change. Such an assumption usually also implies that the size of the container will also not change. This approach is typically applied by older environments such as Visual Basic.

A newer approach is to allow dynamic layouts, which require the use of a layout manager. Components are initially laid out programmatically, but positions are determined by a simple set of principles rather than by absolute x-y coordinates. The approach assumes that the dimensions of the container may be dynamically changed by the user. The layout manager is then charged with dynamically repositioning components to keep the overall look-and-feel and meaning of the GUI coherent. This is the approach used by Java Swing, and increasingly by .Net Windows Forms.

Events

The static look and feel of a GUI is largely accomplished by standard OOP programming that builds and connects objects. This will build a push button, for example, and make it appear on screen. To specify what should happen dynamically when the button is pressed requires additional code defining the GUIs response to events. In Java Swing, event responders are called event listeners. There are many kinds of events – keyboard events, mouse events, window events, etc. – and each event has its own event listener. A similar approach is used by .Net Windows Forms, where the event responder or handler is called a delegate. Each type of event is handled by its own event delegate.

Review of Java Features

Types of Java Class Definition
Look at the following program skeleton:

```
public class NestedClass {
      public class A { … }
      public static class B { … }

      public static void main(String[] args) {
            NestedClass nc = new NestedClass();
            A a = nc.new A();
            B b = new B();
      }
}

class Driver {
      public static void main(String[] args) {
            NestedClass nc = new NestedClass();
            NestedClass.A a = nc.new A();
            NestedClass.B b = new NestedClass.B();
      }
}
```

NestedClass and Driver are called **top level classes** because they are not nested inside other classes.
Top level classes are inherently static, even though the static keyword is not added (cannot be added) to their definition.

A and B are nested classes
  A is a nonstatic nested class (aka inner class)
  B is a static nested class
Static Nested Classes behave almost identically to top level classes.

Result after compiling is

```
Driver.class
NestedClass$A.class
NestedClass$B.class
NestedClass.class
```

See http://mindprod.com/jgloss/nestedclasses.html for a good discussion about the different kinds of Java classes.

Inner classes are inherently non-static. Objects of an inner class are guaranteed to have available a reference to an object of the enclosing outer class (a second reference named "this"). For static nested classes this is not the case.

```
public class Outer {
      public class Inner {
            Inner a = this;
            Outer b = Outer.this;
      }
}

class Driver {
      public static void main(String[] args) {
            Outer x = new Outer();
            Outer.Inner y = x.new Inner();
      }
}
```

For future reference, this is different from some other languages like, for example C#, which supports static nested classes but not inner (non-static) classes.

Anonymous Classes

Anonymous classes are used frequently in Java GUI programming to create simple one-time-use objects such as event listeners. There are several guidelines for when using an anonymous class makes sense:

- The class will be used to instantiate only one object (for example, an event listener).
- The class definition is short and simple (like the **ActionListener** interface, which requires only one method – **actionPerformed()**).
- The class interacts closely with one other class. The information coupling is so strong that making the first class an inner class of the second results in a simplified software architecture (for example, an outer class that represents a GUI element such as a panel and an anonymous class that represents a listener associated with the panel class's buttons or other controls).

Note that these are only guidelines, not requirements.

How to Create Objects of Anonymous Class Types
When we talk about using anonymous classes in Java, what we really mean is that we're creating **objects** whose **type** is defined by **an anonymous class**. To create such an object we do the following:

- Invoke a constructor of the object's superclass (the class is anonymous, but the superclass isn't)
- Between the right parenthesis and the semicolon, place a block of code delimited by curly brackets.
- Inside the brackets, place the definition of the anonymous class. This code **implicitly extends** the superclass.
- What can be extended is very limited, usually just definitions of new methods or overrides of existing or inherited ones.
- Some very common uses of anonymous classes are for building a simple listener by implementing an interface, or by creating a thread by overriding the run method.

Example

```
public class Box {
      public int getSize() { return 3; }
}

class Demo {
      public static void main(String[] args) {
            Box b = new Box();
            Box c = new Box() {                          // <-----
                  public int getSize() { return 4; }
            };
            System.out.println(b.getSize());
            System.out.println(c.getSize());
            System.out.println(b.getClass().toString());
            System.out.println(c.getClass().toString());
      }
}
```

Output is

```
3
4
class Box
class Demo$1
```

Observations

- What is object b’s type? Box
- What is object c’s type? Anonymous subclass of class Box; this class is named “Demo$1” by the compiler.
- You cannot call the constructor for an anonymous class by name, because it’s anonymous. You can only call the constructor for its superclass, then override its methods on the fly, at the same time that the object is actually constructed.

Example
The outer class is named “Anon”, and the anonymous class is defined as an extension (subclass) of class “SomeClass”.

```
class SomeClass {
      int x;
      int y;

      public SomeClass(int xi, int yi) {
            x = xi;
            y = yi;
      }

      public void print() {
            System.out.print(x + " " + y);
      }
}
```

```
public class Anon {
   public static void main(String[] args) {

         System.out.println("Example 1 ---");

         SomeClass a = new SomeClass(2,6);
         a.print();
         System.out.println("");

         System.out.println("Example 2 ---");
         new SomeClass(5,3) {
               public void print() {
                     System.out.println(x + "," + y);
               }
         }.print();

         System.out.println("Example 3 ---");
         SomeClass b = new SomeClass(8,7) {
               public void print() {
                     System.out.println(x + "," + y);
               }
         };
         b.print();
         System.out.println("class = " + b.getClass());

         System.out.println("Example 4 ---");
         SomeClass c = new SomeClass(10,15) {
               public void print() {
                     System.out.println(x + "," + y);
               }
         };
         c.print();
         System.out.println("class = " + c.getClass());
   }
}
```

Output is

```
Example 1 ---
2 6
Example 2 ---
5,3
Example 3 ---
8,7
class = class Anon$2
Example 4 ---
10,15
class = class Anon$3
```

As you can see, objects that belong to an anonymous class are unique members of their class.

- Example 2 creates an anonymous class object, overrides the print() method, and immediately calls it, without saving a reference to the object.
- Examples 3 and 4 create two objects of what appears to be the same class, but the Java environment treats them differently, since it has no way to analyze and confirm that the classes are the same.

Interfaces and Abstract Classes
Used as an OOD strategy. Some features of software design can be communicated by method signatures inherited from interfaces and classes. There is a spectrum of possibilities from purely abstract to purely concrete classes.

Abstract Class
A partially implemented (or wholly unimplemented) class. An abstract class cannot be instantiated. It usually contains at least one abstract method, but it doesn't have to. A class definition can be explicitly abstract via the keyword **abstract.** By this mechanism, a class may be abstract even though it contains no abstract methods, and no unimplemented methods. A class may be implicitly abstract by virtue of only partially implementing an interface, or partially extending another abstract class.

Abstract Method
A method that includes the keyword "abstract" in its declaration and which doesn't include a body, similar to a function prototype or external function declaration in C and C++. The definition must include the return and parameter types.

Interface
Java interfaces are commonly used to define event listeners. An interface defines a set of method signatures without specifying the implementation of the methods. Using a type of inheritance called implementation, a class that implements an interface is obligated to provide a body (block of code) for each method named by the interface, even if the body provided is empty. The methods named by an interface are implicitly **abstract**, **public**, and **non-static**. Interfaces less commonly also contain data declarations. These are implicitly **static** and **final**. An interface is analogous to a pure abstract class, that is, a class in which no method has been implemented.

Interface (Pure Abstract Class) ←→ Abstract Class ←→ (Concrete) Class

Java Inheritance

"Traditional" Inheritance via "extends"
- A class may only directly extend one other class
- "subclass": the class being defined (the extender)
- "superclass": the class being extended (the extendee)
- Any class that **doesn't explicitly** extend a class **implicitly** extends class **Object**

"Multiple" Inheritance via "implements"
- A class may implement an arbitrary number of "interfaces"
- If any interface methods are unimplemented, the resulting class is abstract.

Other Java Features to Be Reviewed Later

- Polymorphism
- Class Cast Exception

Interfaces
An interface is a special “pure abstract” class. All methods are implicitly abstract and non static. Interfaces can be partially implemented by an abstract class or fully implemented by a non-abstract class.

Example 1

```
      interface Iface {
            public void a();
            public void b();
            public void c();
            public void d();
      }

      class C1 implements Iface {
            public void a() { System.out.println("a"); }
            public void b() { System.out.println("b"); }
            public void c() { System.out.println("c"); }
            public void d() { System.out.println("d"); }
      }

      class Driver {
            public static void main(String[] args) {
                  C1 x = new C1();
                  x.a();
                  x.b();
                  x.c();
                  x.d();
            }
      }
```

Example 2

```
      class C2 implements Iface {
            public void a() { System.out.println("a"); }
            public void b() { System.out.println("b"); }
            public void c() { System.out.println("c"); }
      }
```

```
> javac C2.java
```

Output is

```
      C2.java:1: C2 is not abstract and does not override abstract method d() in
      Iface

      public class C2 implements Iface {
             ^
```

Example 2 (corrected)

```
      abstract class C2 implements Iface {
            public void a() { System.out.println("a"); }
            public void b() { System.out.println("b"); }
            public void c() { System.out.println("c"); }
      }
```

Example 3
Multi-level full implementation of the original interface:

```
interface Iface {
      public void a();
      public void b();
      public void c();
      public void d();
}

abstract class C2 implements Iface {
      public void a() { System.out.println("a"); }
      public void b() { System.out.println("b"); }
      public void c() { System.out.println("c"); }
}

class C3 extends C2 {
      public void d() { System.out.println("d"); }
}

class Driver {
      public static void main(String[] args) {
            C3 x = new C3();
            x.a();
            x.b();
            x.c();
            x.d();
      }
}
```

Design Example
Suppose we design a class to demonstrate sorting algorithms, as in COMP 182. We want to test more than one sorting algorithm, but the sort operation should be accessible from the interface with a common name sort().

One Design
```
      class BubbleSorter {
            public int[] data;
            public void generate(int n, int range)  { }
            public void bubblesort() { }
      }
      class QuickSorter {
            public int[] data;
            public void generate(int n, int range)  { }
            public void quicksort() { }
      }
      …
```

In C#, the delegate feature allows the programmer to define a method placeholder or pointer that can be plugged in with a specific method at runtime. Java doesn't really have an equivalent. The "Java Way" to solve this design problem is to use an abstract method.

```
      // abstract superclass with abstract method sort()

      public abstract class Sorter {
            public int[] data;
            public void generate(int n, int range) { … }
            public abstract void sort();  // semicolon required here
      }

      // each subclass overrides the abstract method

      class BubbleSorter extends Sorter {
            public void sort() { System.out.println("bubble"); }
      }

      class QuickSorter extends Sorter {
            public void sort() { System.out.println("quick"); }
      }

      …

      class Driver {
            public static void main(String[] args) {
                  Sorter sorter;
                  sorter = new BubbleSorter();
                  sorter.sort();
                  sorter = new QuickSorter();
                  sorter.sort();
            }
      }
```

So defining a method to be abstract in the superclass, then providing a concrete implementation in a subclass that overrides the abstract definition is one Java equivalent of the delegate feature (function pointer) in C#, to be discussed later.

One limitation of this approach is that constructors are not inherited. Each subclass would need to redefine constructors even if they are all similar.

What OOP feature is demonstrated by the sorter.sort() calls in the driver above?

Other Java Features

- Generics

As of Java 1.5, collections classes (in the java.util package) can be customized for specific base element classes, using the new generics framework. The use of generics eliminates much tedious typecasting code, and results in safer code, since the old style typecasting is vulnerable to throwing ClassCastException.

Example: Vector of Strings

Old Style:
```
        Vector v = new Vector();
        String[] data = {“abc”, “def”, “ghi”};
        for (int i=0; i<data.length; i++)
              v.add(data[i]);               // insert Strings into Vector
        String s = null;
        for (int i=0; i<data.length; i++)
              s = (String)(v.elementAt(i)); // access String in Vector
```

Generics Style:
```
        Vector<String> v = new Vector<String>();
        for (int i=0; i<data.length; i++)
              v.add(data[i]);               // insert Strings into Vector
        String s = null;
        for (int i=0; i<data.length; i++)
              s = v.elementAt(i);           // access String in Vector
```

- Autoboxing and Autounboxing

Java requires that the programmer treat primitive data with value semantics and reference data with reference semantics. Some contexts, such as most java.util collections classes, require references to objects and cannot work with direct primitive values. For this reason, each primitive data type has its own “wrapper” class type, used to create an object to “wrap” a primitive value into a simple object when reference syntax is required.

Creating a wrapper object and inserting a primitive value is called “boxing”, and extracting a primitive value from a previously boxed wrapper object is called “unboxing”. As of Java 1.5, the compiler will perform autoboxing and autounboxing, which means that conversions between primitive values and wrapper objects will be performed implicitly and automatically.

Old Style
```
        Vector v = new Vector();
        int x = 5;
        Integer y = new Integer(x);
        v.add(y);
```

New Style
```
        Vector v = new Vector();
        int x = 5;
        v.add(x);                     // autoboxing of x occurs here
```

There is a similar auto-unboxing effect on the other end, when removing a wrapper object and assigning to a primitive.

- New “for” Syntax (the so-called “foreach” loop)

Old Style
```
      int[] data = {3,5,7,9,11};
      for (int i=0; i<data.length; i++) System.out.println(data[i]);
```

New Style
```
      int[] data = {3,5,7,9,11};
      for (int x:data) System.out.println(x);
```

In the new style, the explicit index variable disappears. The declared variable x acts as an **iterator**, taking on in turn each value in the array data. The old style is still useful and not going away! The new style is sometimes calle the “foreach” loop, although there is no “foreach” keyword in Java.

New in Java 7

Swing
- JLayer Class for decorating components and responding to events
- Nimbus Look and Feel
- Better Mixing of Heavyweight and Lightweight Components
- Shaped and Translucent Windows
- Hue-Saturation-Luminance (HSL) Color Selection in JColorChooser Class

Java Programming Language

Binary Literals	[int x = 0b10101;]
Underscores in Numeric Literals	[int y = 123_456_789;]

- Strings in switch statements
- Try-with-resources statement
- Catching multiple exception types, rethrowing exceptions with improved type checking
- Type inference for generic instance creation
- Improved Compiler Warnings/Errors with Non-Reifiable Formal Params with Varargs Methods
  - Non-reifiable type: not completely available at runtime

Other
- Networking
- Security
- Concurrency Utilities
- Java 2D
- Java XML

New in Java 8

- Lambda Expressions
- Package java.util.stream

Java Packages of Interest

General-interest packages

java.lang (System,String,wrapper classes)
java.util (ArrayList,Vector,Stack, other collections)
java.io (files, streams, readers, writers)
...

awt: abstract windowing toolkit

java.awt
java.awt.event
java.awt.image
...

swing: aka JFC or Java Foundation Classes

javax.swing
javax.swing.event
...

The Abstract Windowing Toolkit (AWT)

- java.awt
- java.awt.event
- others

Original set of class definitions for building GUIs
Color, Point, Dimension, Font
Button, TextField, Panel, Canvas, Frame

AWT widgets are "heavyweight" components, which means that the AWT widget is associated with a **peer object** written in native code (not Java) that actually handles the drawing of that widget on the screen. Heavyweight components have good performance, but their "look-and-feel" is more closely bound to the style used by the operating system that the application is executing on. More about that later.

The Java Foundation Classes (JFC)

- javax.swing
- javax.swing.event
- others

JFC was code-named "swing", and the name stuck. Does not replace or supersede AWT, the two must be used carefully in tandem. Certain AWT classes are not replaced: Color, Point, Dimension, Font

Most widgets are reimplemented with a new class beginning with "J": JButton, JTextField, JPanel, JFrame

A few top-level Swing components, such as JFrame, are still "heavyweight", but most others are "lightweight". Use of lightweight components makes possible a "pluggable look-and-feel" feature for the visual style of components. Care must be taken when using heavyweight components in Swing to correctly allocate and deallocate the resources they use, in order to avoid memory leaks.

Outline of Java Swing GUI Building Blocks

Top-Level Containers

- JFrame
- JDialog
- JApplet

Other Containers

- JPanel
- JScrollPane
- JSplitPane
- JTabbedPane

Basic Components

- JLabel
- JTextField
- JButton
- JRadioButton (see also class ButtonGroup)
- JCheckBox
- JComboBox

Complex Components

- JList
- JTree
- JTable

Special Purpose Components

- JSlider
- JSpinner

Text Components

- JTextArea
- JTextPane
- JEditorPane

Special Purpose Dialogs

- JOptionPane
- JFileChooser
- JColorChooser

2D Graphics Classes from java.awt used with javax.swing Classes

- Graphics
- Color
- Dimension
- Font
- Image
- Point
- Polygon
- Rectangle

"Model View Controller" Framework for GUI Elements

The GUI is the layer of the software application that gives the end user intuitive access to the application's features.

A common framework for discussing how GUIs work is called **model-view-controller**.

- model: a container for data in storage (array, vector, linked list, other data structure)
- view: a visual representation of the data inside the model (often a subset of the complete data set)
- controller: a software agent that maintains consistency between model and view.

In Java Swing, the view and controller are merged into a single element called a delegate (which has nothing to do with C# delegates discussed later).

Heavyweight vs. lightweight GUI components
A good article/white paper on the difference between awt and swing
http://www.oracle.com/technetwork/java/painting-140037.html

Painting in AWT and Swing

AWT components are **heavyweight**. This means that the drawing or painting of the component to the display depends on a code provided by the local platform (not the Java packages). For every heavyweight component, there is associated with it a "peer" that knows how to draw a native window according to the software provided by the local platform.

In contrast, most (but not all) Swing components are **lightweight**. This means that they will always be displayed within the boundary of some heavyweight component in the visual hierarchy, but that within their sub-region of the screen, the component is drawn completely by code provided by the java packages. In short, a GUI built from Swing components has far fewer native windows running around cluttering up the desktop.

Painting is via callbacks.
    Programmer places all painting code in the callback.
    System calls the callback at the appropriate time.
System-triggered painting
    paint(Graphics g) [note: AWT only, do not override paint() in Swing]
    paintComponent(Graphics g)
App-triggered painting
    repaint()
    overridden versions provide coordinate defining the subregion to repaint

There is a third method update() which is subtly different from paint()/repaint(). The bottom line for lightweight components is that update() and repaint() are essentially the same, so don't mess with update().

Bottom line is that any custom painting defined for a component by the programmer goes into the callback **paintComponent().** The system decides when it needs to be called. In order for the app to trigger a repaint, call the **repaint()** method, which usually occurs from the event thread.

Inheritance Hierarchy
This is critical in defining GUI elements that have a consistent API and behavior.

Heavyweight Examples:
- JFrame
  - Object(lang)/Component(awt)/Container(awt)/Window(awt)/Frame(awt)
- JDialog
  - O/C/C/W/Dialog
- JApplet
  - O/C/C/Panel/Applet

Lightweight Examples:
- JButton (lightweight)
  - O/C/C/JComponent(swing)/AbstractButton(swing)
- JTextField
  - O/C/C/JC/JTextComponent(swing)
- JLabel
  - O/C/C/JC
- JMenuItem
  - O/C/C/JC/AbstractButton
- JMenu
  - O/C/C/JC/AbstractButton/JMenuItem
- JPanel
  - O/C/C/JC

At each step in the hierarchy, certain operations are introduced that are then available to all subclasses.

Object
- clone()
- toString()
- equals()
- wait()/notify()

Component
- "visual" properties that allow the object to be drawn/painted to the display
- location and size properties
- font
- foreground/background color
- graphics context
- keyboard focus
- paint callback method

Container
- list of contained components
- layout manager

JComponent
- pluggable Look and Feel (L&F)
- input and action "maps" to provide convenient keystroke handling
- tool tips
- painting that supports double buffering and borders

Visual Containment Hierarchy
Related to the heavyweight vs. lightweight discussion earlier every Swing GUI is built with a minimum of one "top-level" container. This is usually a JFrame. A JFrame is a heavyweight component that provides a window in which the rest of the application's visual information is displayed. Any other element that is a part of the same application is either:

another independent top-level container (2nd JFrame, JDialog, etc.)

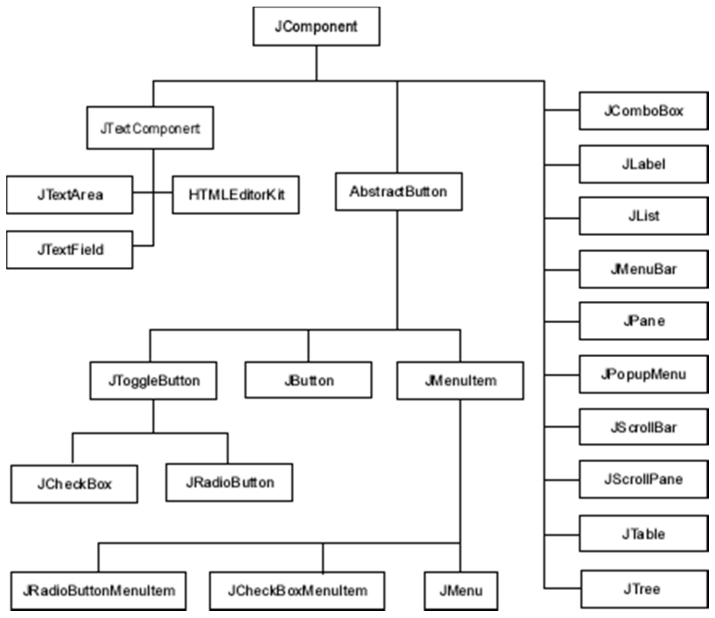
a lightweight component attached to the visual hierarchy rooted by this JFrame.

Visual Hierarchy
Not all Java objects are **visual** objects. A String is a Java object, but it doesn't possess the requisite properties to allow it to be displayed directly on the desktop. The GUI building blocks in the AWT and Swing packages are **visual** components that can be displayed.

In addition to being visual components, the elements in a Java Swing GUI must be arranged in a **visual hierarchy** for a specific application, rooted by a top-level container (for example, JFrame). An internal container like JPanel is attached to the JFrame. Other components (JButton, JTextField, JLabel) are attached to the JPanel. Once the elements of the GUI are assembled, the complete visual hierarchy is displayed by making the JFrame visible.

The way in which the visual hierarchy is built programmatically implies properties of the components such as stacking or Z-order. This will impact how the visual hierarchy is displayed at any one time (which components are displayed on top of or behind other visual components, for example).

Inheritance Hierarchy for Swing and AWT Classes
See http://www.comp.nus.edu.sg/~cs3283/ftp/Java/swingConnect/archive/what_is_arch/tool_set/tool_set.html



Typo: JPane should be JPanel
Components that descend from JComponent are lightweight, but not all Swing classes are descendants of JComponent.

Object ➔ Component ➔ Container ➔ JComponent

Container	➔	JComponent	➔	… (lw)		
Container	➔	Panel	➔	Applet	➔	JApplet (hw)
Container	➔	Window	➔	Dialog	➔	JDialog (hw)
		Window	➔	Frame	➔	JFrame (hw)
		Window	➔	JWindow (hw)		

- Descendants of JComponent are lightweight components
- Descendants of Container (but not JComponent) are heavyweight (JFrame, JWindow, JDialog, JApplet)
- Note there is no JContainer Class.

Briefly:
Lightweight components are rendered in Java
Heavyweight components have a companion object called a peer
The peer is implemented in native (non-Java) code
The peer renders the heavyweight object
The visual containment hierarchy of any Java GUI begins with a heavyweight component
For starters, we will assume JFrame, but others are possible.
Rules for mixing heavyweight and lightweight components in one app are complex.
But Java 7 has just introduced some improvements for this situation.

Important constraint: cannot add a Window to a Container
Throws IllegalArgumentException at runtime

The inheritance hierarchy simplifies the management of common properties such as
Size
Position
Array of Subcomponents
…

Framework for a Widget-Based Swing Application

From the main method

- Create a JFrame object (a heavyweight top-level container)
- Create a JPanel object (the internal container or content pane of the frame)
- Attach the JPanel to the JFrame.
- Create desired widgets and attach them to the JPanel (typically, this simply uses the **add** method).
- Give the JFrame a size.
- Set the JFrame's "visible" property to "true" (should be the last step of the GUI building process).

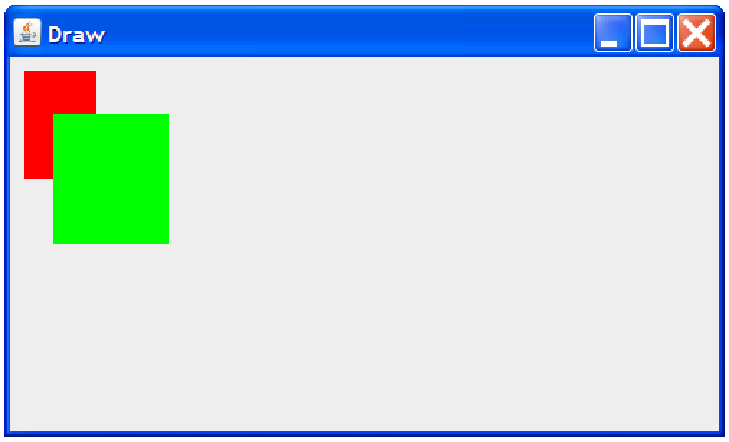
Code Example: SimpleForm.java

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

public class SimpleForm {

      public static void main(String[] args) {
            JFrame frame = new JFrame("Simple Form Demo");
            JPanel panel = new JPanel();
            JLabel label = new JLabel("Enter Value:");
            JTextField textfield = new JTextField(10);
            JButton button = new JButton("Submit");

            panel.add(label);
            panel.add(textfield);
            panel.add(button);

            frame.setContentPane(panel);

            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(400,100);
            frame.setVisible(true);
      }
}
```

Framework for a Drawing-Based Swing Application

- Similar to above framework.
- Instead of attaching widgets to the panel, obtain the Graphics object for the panel, and apply the appropriate drawing method to the graphics object.
- Best approach is to subclass the JPanel class and override the a paintComponent() method.
- This method is automatically called by the JVM whenver necessary to redraw the JPanel.

Simple Drawing Example: Panel with Simple Geometric Drawing Instructions

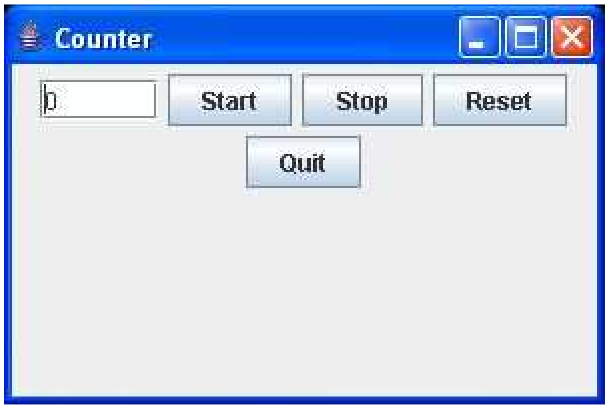


Note: programming style is not recommended, presented only for simplicity

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

public class Drawing {
      public static void main(String[] args) {
            JFrame f = new JFrame("Draw");
            final JPanel p = new JPanel();
            f.setContentPane(p);

            Thread t = new Thread()  {
                  public void run() {
                        Graphics g = p.getGraphics();
                        g.setColor(Color.RED);
                        g.fillRect(10,10,50,75);
                        g.setColor(Color.GREEN);
                        g.fillRect(30,40,80,90);
                  }
            };

            f.setSize(500,300);
            f.setVisible(true);
            SwingUtilities.invokeLater(t);
      }
}
```

This example demonstrates a simple relationship between frames, panels, graphics, and drawing methods. Style is poor, however, and does not accommodate necessary refresh of the application.

A Simple But Complete Widget Example: A Threaded Integer Counter with Button Control



Counter 1.0

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

public class Counter1 {
      public static void main(String[] args) {

            JFrame frame = new JFrame("Counter"); // create objects
            JPanel panel = new JPanel();
            JTextField tf = new JTextField(5);

            tf.setText("0");                      // property values
            panel.add(tf);
            panel.add(new JButton("Start"));
            panel.add(new JButton("Stop"));
            panel.add(new JButton("Reset"));
            panel.add(new JButton("Quit"));
            frame.setContentPane(panel);

            frame.setSize(300,200);               // set size
            frame.setVisible(true);               // make visible
      }
}
```

Version 1.0 will draw the GUI, and it's a good starting point for simply demonstrating some basic features of the API. But there are many problems. For example:

- Poor use of OOP
- No encapsulation of GUI features into an object.
- Reference variables for GUI components defined as local variables inside main method.
- Main method should be reserved for client code. It should not handle internal detail concerned with creation and management of the GUI itself.

GUI Building and OOP/OOD
As with every programming problem category, GUI programming supports many different software architecture “styles”. Every programmer will want to develop their own style. Here, I suggest a very popular style for organizing the code required by a GUI. But every programmer should feel free to use this as a starting point, and experiment. Also, in some cases different problems will be best served by different styles.

- The GUI should be represented by a top-level class that extends either **JFrame** or **JPanel**.
- The important components – components that need to be referenced during the lifetime of the GUI – should be defined as **instance variables** positioned as instance variables of the top-level class.
- The GUI constructor should do the majority of the GUI building.
- Define other methods as needed to manipulate the GUI from client code, in traditional OOP style.
- The main method is not normally a member of the GUI class. Instead, provide a static method – **buildGUI()** or something similar – that can be called from the main or other method in the client.
- Event listeners are best defined as inner classes (frequently, anonymous classes) of the top-level class. This arrangement simplifies access to object state between closely related objects.
- If the top-level class extends JFrame, then most build code is in the constructor, and the buildGUI() method only needs to call the constructor. If the top-level class extends JPanel, then the constructor only builds the JPanel, and the buildGUI() method must finish the details involving the application’s JFrame.

Version 2
Here's a variation that uses a top-level class that extends JFrame, with an inner class that extends JPanel.

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

class CounterFrame extends JFrame {

      class CounterPanel extends JPanel {
            JTextField tf;
            JButton btnStart,btnStop,btnReset,btnQuit;

            public CounterPanel() {
                  tf = new JTextField(5);
                  tf.setText("0");
                  this.add(tf);

                  btnStart = new JButton("Start");
                  this.add(btnStart);

                  btnStop = new JButton("Stop");
                  this.add(btnStop);

                  btnReset = new JButton("Reset");
                  this.add(btnReset);

                  btnQuit = new JButton("Quit");
                  this.add(btnQuit);
            }
      }

      public CounterFrame(String caption) {
            super(caption);
            CounterPanel panel = new CounterPanel();
            this.setContentPane(panel);
      }

      public static void createGUI() {
            CounterFrame frame = new CounterFrame("Counter");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(300,200);
            frame.setVisible(true);
      }
}

public class Counter2 {
      public static void main(String[] args) {
            CounterFrame.createGUI();
      }
}
```

Improvements

- Details are encapsulated into CounterFrame and CounterPanel classes, hidden from client.
- Widgets are instance variables of CounterPanel class, so they can be referred to later by event handlers.
- Frame is set to exit on close.

Events and Listeners
Counter Version 3 should make the buttons do something when clicked. Before we try to implement Counter Version 3, we need to look at events, listeners, and adapters.

Event-Driven Program Flow of Control
Imagine a generic software application for which the user interface details have been abstracted away. What is left is essentially a series of services that the software stands ready to perform for the user when requested. The user interface simply provides the request for the specific service.

If a GUI is not available, a standard undergraduate approach to receiving user requests from the input and dispatching the request to the correct software service is a command-line interface (CLI) or command-line prompt. In its simplest form, it is easily implemented as an input statement to receive the request from the keyboard input, an if-else ladder to interpret the choice represented by the request, then a call to the function or method that implements the requested service.

Command-Line Prompt
Event = User Command Choice, Event Dispatch Under **Client** Control

```
main() {
      while (true) {
            prompt();
            command = input();

            if (command == command1) {
                  command1Handler();
            }
            else if (command == command2) {
                  command2Handler();
            }
            …
            else if (command == “quit”) {
                  break;
            }
      }
}
```

How does this change with a GUI?
Input is now expanded to include the mouse.
Rather than typing textual responses to prompts, most end users expect to be able to point and click, interacting with standardized GUI elements: buttons, check boxes, scroll bars, menus, minimize/maximize/close window controls, etc.

In the examples above, the building of the elements that make up the visual hierarchy uses standard structured and OOP programming techniques. What's missing is the functions or methods that represent the services provided by the software application. How are these to be connected to the code that builds the GUI? Clearly the functions are not meant to be invoked during app initialization while the GUI is being built. They are intended to be "attached" to a GUI element such as a button, and invoked at runtime dynamically at the time when the user points and clicks the button.

Java handles this kind of interaction by defining events and listeners. Events are created by the system automatically when the user interacts with the GUI: mouse events, window events, component events, key events, etc. Listeners are special program blocks that can be bound or registered with a GUI element and activated when the user causes that element to fire an event.

Event = User action with mouse or keyboard, Event Dispatch Under **System** Control

```
main() {
      buildGui();
            button #1 ←→ command1Handler()
            button #2 ←→ command2Handler()
}

void command1Handler() {
      // here when button #1 is pressed
      …
}

void command2Handler() {
      // here when text entered in text field 3
      …
}

…
```

User provides definitions or implementations of special methods called event handlers. User doesn't typically have any control over when they are invoked. Invocation happens automatically when the user of the application triggers the relevant event.

Events in Java

ActionEvent and ActionListener for Handling Button Clicks

- Event: ActionEvent
- Interface: ActionListener
- Action: clicking on a Button with the mouse, pressing <return> while cursor is in a TextField, others.

The ActionEvent is a useful general-purpose or catchall event for common GUI operations. Corresponding to the ActionEvent is the ActionListener interface. The interface describes in outline form what an object who responds to an ActionEvent should look like. Here's the interface:

```
interface ActionEvent {
      public void actionPerformed(ActionEvent e);
}
```

Steps During Instantiation of GUI

- Instantiate JButton object as usual.
- New: instantiate some object that implements the ActionListener interface.
- New: attach ("register") the listener to the button using the addActionListener() method.

Steps During Operation of GUI

- User uses the mouse to click on the Button
- JVM instantiates an ActionEvent object which stores information about the action. This happens whether there are any listeners for the event or not.
- JVM checks the Button to see if any ActionListeners have been registered with the Button.
- If so, the actionPerformed() method of each ActionListener is called. The ActionEvent object is passed to this method as a parameter. The method may examine the event object if it needs to.
- If no listeners have been registered with the button, the ActionEvent object is discarded.

There are many other types of event and listener. The next most important are

- MouseEvent and MouseListener
- MouseEvent and MouseMotionListener (there is no MouseMotionEvent)
- WindowEvent and WindowListener

For future reference, it will be interesting to see how other plaforms resolve the same problem. For example, the .NET platform uses delegates rather than listeners. Delegates are similar to function pointers that are encapsulated as an object, attached to a GUI element, and invoked dynamically at a later time.

An alternative would be to solve the custom component behavior problem purely by subclassing:

```
class MyButton extends JButton {
      public void whenClicked() {
            …                        // this method overrides the default method
      }
}
```

Java GUIs have chosen not to take this approach. Interestingly, .NET provides a similar mechanism, in addition to the delegate approach mentioned earlier.