
Java

OOP Overview

OOP

Object oriented

- ✓ The way to organize a program as a set of component called object.
- ✓ It is a power full way of organizing and developing a software.
- ✓ one characteristics of java.

Object and class:

Class

A class is a logical framework or blue print or template of an object.

A class is a programmer defined data type.

A class has :-

- ❖ state (field) and
- ❖ behavior (method)

Fields: say what a class is (Things an object knows about itself).

Hold data values.

Methods: say what a class does (Things an object can do).

Operates on those values.

Definition and declaration of a class

Syntax:

```
class ClassName
{
    //member attributes (instance /classvariables)
    //member Methods
}
```

Example:

```
public class Student //consider the naming
                    //convention of a class
{

}
```

Member methods : instant method, main method...

Instance variable: a variable which all object of a class can use

The structure of a method includes a method signature and a code body

[access modifier] return type method name (list of arguments)

{

statements, including local variable declarations

}

main() method Inside the class, outside other instance methods.

Example

```
class Circle
{
    float rad;
    int xCoord, yCoord;
    void showArea(){
        float area = Math.PI*rad*rad;
        System.out.println("The area is:" + area);
    }
    void showCircumference()
    {
        float circum=2* Math.PI*rad;
        System.out.println("The circumference is:"+circum);
    }
}
```

Object ?

Object

- ❖ An object is an instance of a class.
- ❖ Creating an object is a two-step process.

Creating a reference variable

Syntax:

`<class idn><ref. idn>;`

eg. `Circle c1;`

Setting or assigning the reference with the newly created object.

Syntax:

`<ref. idn> = new <class idn>(...);`

`c1 = new Circle();`

The two steps can be done in a single statement

`Circle c2 = new Circle();`

Accessing members of an Object

To access members of an object we use ‘.’ (*dot*) operator together with the reference to an object.

`<ref. idn>.<member>;`

Example

```
Circle c1 = new Circle();  
c1.rad = 2.3;  
c1.showArea();  
c1.showCircumference();
```

object and class

Class is a type.

Object is instance of a type (class).

Static Members?

Static Variables

- **Member data** - Same data is used for all the instances (objects) of some Class.

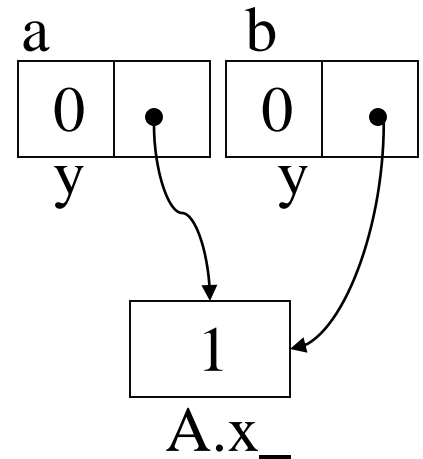
```
Class A {  
    public int y = 0;  
    public static int x_  
1;  
};
```

*Assignment performed
on the first access to the
Class.
Only one instance of 'x'
exists in memory*

```
A a = new A();  
A b = new A();  
System.out.println(b.x_);  
a.x_ = 5;  
System.out.println(b.x_);  
A.x_ = 10;  
System.out.println(b.x_);
```

Output:

1
5
10



Static Method

- **Static member method/function**

- Static member function can access only static members
- Static member function can be called without an instance.

```
Class TeaPot {  
    private static int numOfTP = 0;  
    private Color myColor_  
    public TeaPot(Color c) {  
        myColor_ = c;  
        numOfTP++;  
    }  
    public static int howManyTeaPots()  
    { return numOfTP; }  
  
    // error :  
    public static Color getColor()  
    { return myColor_; }  
}
```

Constructor ?

Constructor

- Special type of method invoked when you instantiate a class.
- Constructors are used to initialize the instance variables (fields) of an object.
- Its called when the object is created.
- Constructors are similar to methods, but with some important differences.
 - ❖ Constructor name is class name.
 - ❖ Default constructor is created by compiler If you don't define a constructor for a class.
 - ❖ no return type there fore no return statement even no void.

Syntax: `public ClassName(arguments)`
`{`

`}`

Inheritance

- A feature that allow us to drive class from another class.
- It allows programmers to customize a class for a specific purpose, without actually modifying the original class (the superclass)
- The derived class (subclass) is allowed to add methods or redefine them
- The subclass can add variables, but cannot redefine them

Inheritance

Two classes involved in inheritance:

- 1.Subclass : Inherits.
- 2.Superclass:Allows another class to inherit.

Inheritance Example

Class C is a subclass of class B (its superclass) if its declaration has the form:

```
class C extends B {  
  ...  
}
```

The subclass is a *specialization* of the superclass

The superclass is a *generalization* of the subclass

Inheritance and Messages

- When C is a subclass of B
 - C objects can respond to all messages that B objects can respond to
- In general C objects can be used whenever B objects can be used
- It is possible the a subclass of B may have methods and variables that have not been defined in B
- It is the case B objects may not always be used in place of C objects

Inheritance Hierarchy

- A class may have several subclasses and each subclass may have subclasses of its own
- The collection of all subclasses descended from a common ancestor is called an *inheritance hierarchy*
- The classes that appear below a given class in the inheritance hierarchy are its *descendants*
- The classes that appear above a given class in the inheritance hierarchy are its *ancestors*

Inheritance and Visibility Rules

Private variables and methods are not visible to subclasses or clients. (accessible only from within this class)

Public variables and methods are visible to all subclasses and clients. (accessible from anywhere)

Variables and methods with **package** visibility are only visible to subclasses and clients defined in the same package as the class. (accessible from the same package)

A variable or method declared with the **protected** visibility modifier can only be referenced by subclasses of the class and no other classes. (accessible from any descendants in the same package)

Constructors and inheritance

- The general rule is that when a subclass is created Java will call the superclass constructor first and then call the subclass constructors in the order determined by the inheritance hierarchy
- If a superclass does not have a default constructor with no arguments, the **subclass** must explicitly call the superclass constructor with the appropriate arguments

Using super() Call Constructor

The call to super must be the first statement in the subclass constructor

Example:

```
class C extends B {  
    ...  
    public C ( ... ) {  
        super( B's constructor arguments );  
        ...  
    }  
    ...  
}
```


Abstraction

Abstraction is the thought process where in **ideas are separated from objects.**

We form concepts of everyday objects and events by a process of abstraction where we remove unimportant details and concentrate on the essential attributes of the thing.

Roughly speaking, abstraction can be either that of ***control or data.***

Control abstraction is the abstraction of actions while data abstraction is that of data

Abstraction is just a fancy name for **encapsulating the data and method.**

Abstract Classes

- Abstract classes are only used as super classes
- Classes are declared as abstract classes only if they will never be instantiated
- Abstract classes contain usually **one** or **more** abstract methods

Example:

```
public abstract class Mouse implements Direction {  
    ...  
    abstract void makeMove( );  
}
```

Abstract Methods

Abstract methods have no body at all and just have their headers declared

The only way to use an abstract class is to create a subclass that implements each abstract method

Concrete classes are classes that implement each abstract method in their superclasses

Example:

```
abstract void makeMove( );
```

Method is abstract if :

- 1/Declared as abstract.

- 2/Declared inside an interface.

Interface

- An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference.
- The difference is that interfaces define only abstract methods and final fields.

Creating an interface

```
interface InterfaceName
{
    Variable declaration;
    Method declaration;
}
```

...interface

Example:

//shape.java

```
interface Shape
{
    final double PI = 3.14;
    public abstract double Volume();
    public abstract double getName();
}
```

...Interface

- All the methods are defined as abstract (method with no implementation).
- Interface shall also be saved in Java file using .java extension.
- If there are attributes in an interface they should be static final.

...Interface

- Use **implements** keyword to implement the abstract method of the interface.

```
public class Point extends Object implements Shape
{
```

```
    // body of point class
```

```
    //class implements an interface if it supplies code for all  
    method of that interface
```

```
}
```

An interface can be sub interfaced from another interface.

Class Vs Interface

Similarities

- **Class**

Provides secondary data type for the object of the subclass

If the class is abstract class, it cannot be instantiated

A sub class of abstract class must implement all the inherited abstract method.

Class extend another class

- **Interface**

Provides secondary data type for the object of the class that implements it

An interface cannot be instantiated.

A class that implements an interface must implement all method of the interface

Interface extend another interface

Difference

- Class

Class can extend only one class

Class defines its own
constructor

If class is abstract one/more
method is abstract

- Interface

Class can implement any number
of interface

An interface has no constructor

All methods are abstract.

Interface

Interface can extend another interface.
use extends keyword.

Syntax:

```
interface SubInterface extends SuperInterface
{
    //body of sub interface

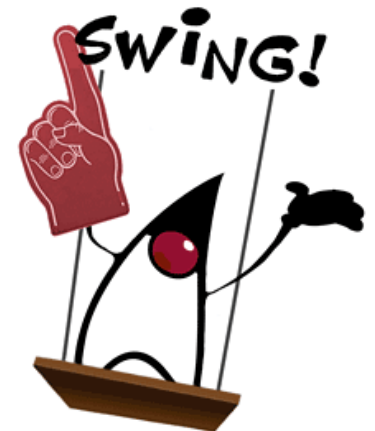
}
```

Chapter One

Graphical User Interfaces (GUIs) with Swing/AWT

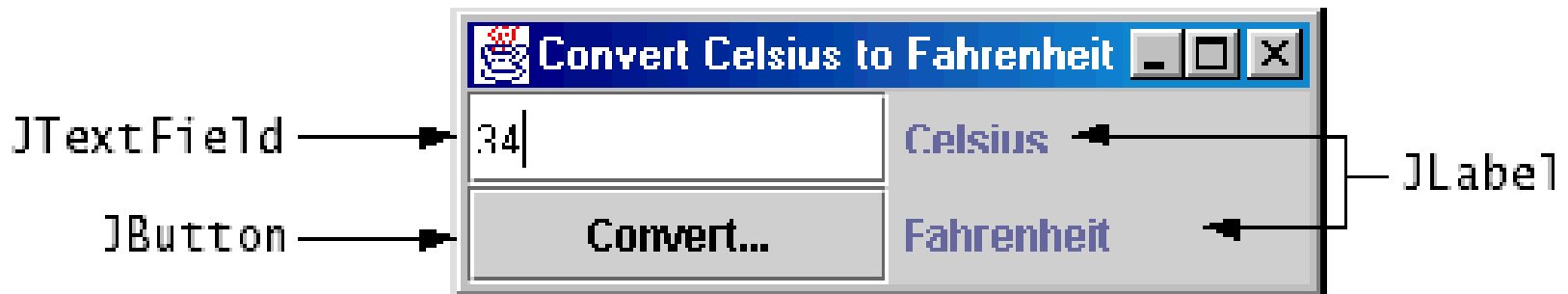
Java GUI History

- **Abstract Windowing Toolkit (AWT):** Sun's initial effort to create a set of cross-platform GUI classes. (*JDK 1.0 - 1.1*)
 - Maps general Java code to each operating system's real GUI system.
 - *Problems:* Limited to lowest common denominator; clunky to use.
- **Swing:** A newer GUI library written from the ground up that allows much more powerful graphics and GUI construction. (*JDK 1.2+*)
 - Paints GUI controls itself pixel-by-pixel rather than handing off to OS.
 - *Benefits:* Features; compatibility; OO design.
 - *Problem:* Both exist in Java now; easy to get them mixed up; still have to use both in various places.

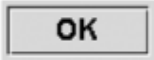




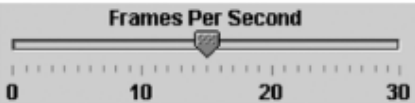

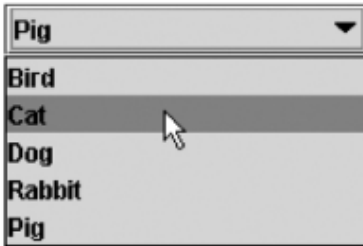

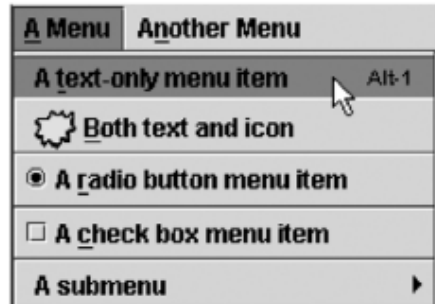
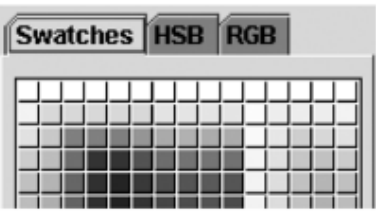








GUI terminology

- **window:** A first-class citizen of the graphical desktop.
 - Also called a *top-level container*.
 - examples: frame, dialog box, applet
- **component:** A GUI widget that resides in a window.
 - Also called *controls* in many other languages.
 - examples: button, text box, label
- **container:** A logical grouping for storing components.
 - examples: panel, box



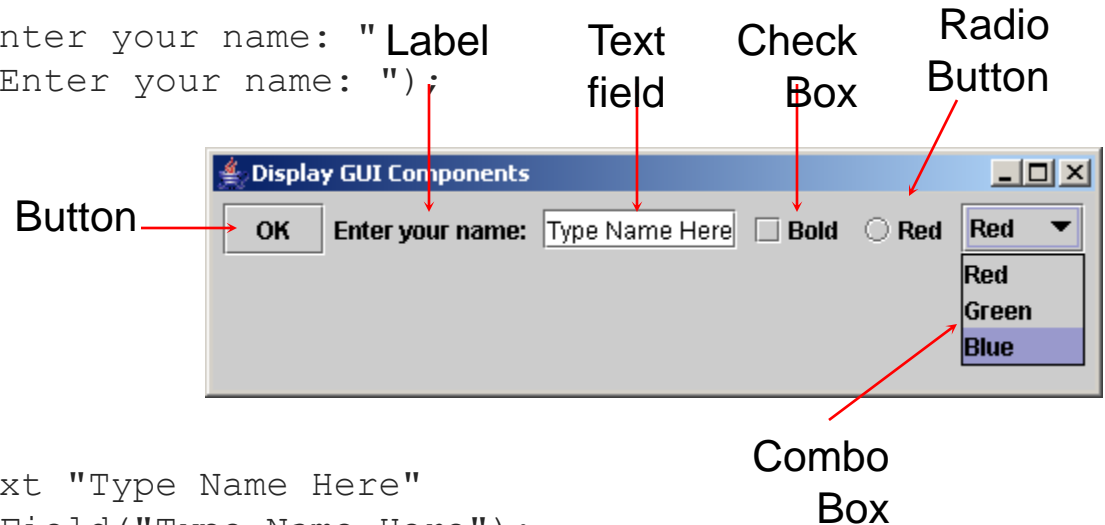
Components

JButton 	JCheckBox 	JRadioButton 	JLabel  Text-Only Label														
JTextField 	JSlider 	JToolBar 															
JComboBox 	JList 	JMenuBar, JMenu, JMenuItem 															
JColorChooser 	JFileChooser 	JTable  <table><thead><tr><th>First Name</th><th>Last Name</th><th>Favorite F</th></tr></thead><tbody><tr><td>Jeff</td><td>Dinkins</td><td rowspan="5"></td></tr><tr><td>Ewan</td><td>Dinkins</td></tr><tr><td>Amy</td><td>Fowler</td></tr><tr><td>Hania</td><td>Gajewska</td></tr><tr><td>David</td><td>Gearv</td></tr></tbody></table>	First Name	Last Name	Favorite F	Jeff	Dinkins		Ewan	Dinkins	Amy	Fowler	Hania	Gajewska	David	Gearv	JTree  <ul style="list-style-type: none">Music<ul style="list-style-type: none">Classical<ul style="list-style-type: none">BeethovenBrahmsMozartJazzRock
First Name	Last Name	Favorite F															
Jeff	Dinkins																
Ewan	Dinkins																
Amy	Fowler																
Hania	Gajewska																
David	Gearv																

Creating GUI Objects

```
// Create a button with text OK
JButton jbtOK = new JButton("OK");
```

```
// Create a label with text "Enter your name: "
JLabel jlblName = new JLabel("Enter your name: ");
```



```
// Create a text field with text "Type Name Here"
JTextField jtfName = new JTextField("Type Name Here");
```

```
// Create a check box with text bold
JCheckBox jchkBold = new JCheckBox("Bold");
```

```
// Create a radio button with text red
JRadioButton jrbRed = new JRadioButton("Red");
```

```
// Create a combo box with choices red, green, and blue
JComboBox jcbColor = new JComboBox(new String[]{"Red",
    "Green", "Blue"});
```


Swing inheritance hierarchy

- Component (AWT)

- Window

- Frame

- **JFrame** (Swing)

- **JDialog**

- Container

- JComponent (Swing)

- **JButton**

JColorChooser

JFileChooser

- **JComboBox**

JLabel

JList

- **JMenuBar**

JOptionPane

JPanel

- **JPopupMenu**

JProgressBar

JScrollbar

- **JScrollPane**

JSlider

JSpinner

- **JSplitPane**

JTabbedPane

JTable

- **JToolBar**

JTree

JTextArea

- **JTextField**

...

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

Component properties

- Each has a `get` (or `is`) accessor and a `set` modifier method.
- examples: `getColor`, `setFont`, `setEnabled`, `isVisible`

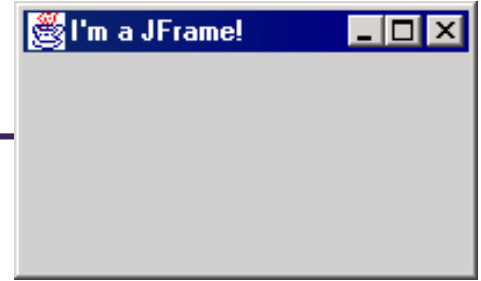
name	type	description
background	Color	background color behind component
border	Border	border line around component
enabled	boolean	whether it can be interacted with
focusable	boolean	whether key text can be typed on it
font	Font	font used for text in component
foreground	Color	foreground color of component
height, width	int	component's current size in pixels
visible	boolean	whether component can be seen
tooltip text	String	text shown when hovering mouse
size, minimum / maximum / preferred size	Dimension	various sizes, size limits, or desired sizes that the component may take

Frames

- Frame is a window that is not contained inside another window. Frame is the basis to contain other user interface components in Java GUI applications.
- The JFrame class can be used to create windows.
- For Swing GUI programs, use JFrame class to create windows.

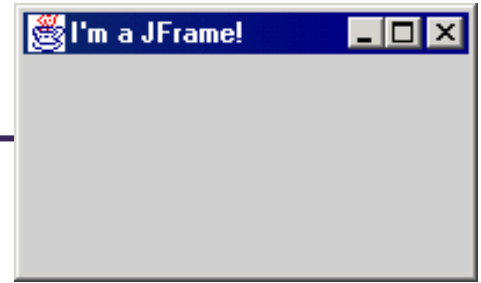
JFrame

a graphical window to hold other components



- `public JFrame()`
`public JFrame(String title)`
Creates a frame with an optional title.
 - Call `setVisible(true)` to make a frame appear on the screen after creating it.
- `public void add(Component comp)`
Places the given component or container inside the frame.

JFrame



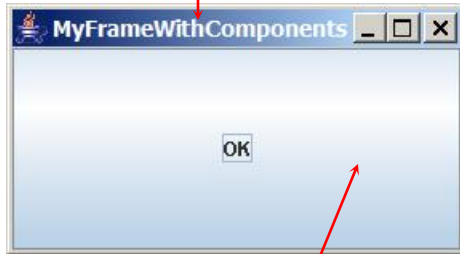
- `public void setDefaultCloseOperation(int op)`
Makes the frame perform the given action when it closes.
 - Common value passed: `JFrame.EXIT_ON_CLOSE`
 - If not set, the program will never exit even if the frame is closed.
- `public void setSize(int width, int height)`
Gives the frame a fixed size in pixels.
- `public void pack()`
Resizes the frame to fit the components inside it snugly.

Creating Frames

```
import javax.swing.*;
public class MyFrame {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Test Frame");
        frame.setSize(400, 300);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Adding Components into a Frame

Title bar



Content pane

```
// Add a button into the frame  
frame.getContentPane().add(  
    new JButton("OK"));
```

```
// Add a button into the frame  
frame.add(new JButton("OK"));
```

JFrame Class

javax.swing.JFrame

+JFrame()

Creates a default frame with no title.

+JFrame(title: String)

Creates a frame with the specified title.

+setSize(width: int, height: int): void

Specifies the size of the frame.

+setLocation(x: int, y: int): void

Specifies the upper-left corner location of the frame.

+setVisible(visible: boolean): void

Sets true to display the frame.

+setDefaultCloseOperation(mode: int): void

Specifies the operation when the frame is closed.

+setLocationRelativeTo(c: Component):
void

Sets the location of the frame relative to the specified component.
If the component is null, the frame is centered on the screen.

+pack(): void

Automatically sets the frame size to hold the components in the frame.

JButton



Button 1

a clickable region for causing actions to occur

- `public JButton(String text)`
Creates a new button with the given string as its text.
- `public String getText()`
Returns the text showing on the button.
- `public void setText(String text)`
Sets button's text to be the given string.

GUI example

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

public class GuiExample1 {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 100));
        frame.setTitle("A frame");

        JButton button2 = new JButton();
        button2.setText("Click me!");
        button2.setBackground(Color.RED);
        frame.add(button2);

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

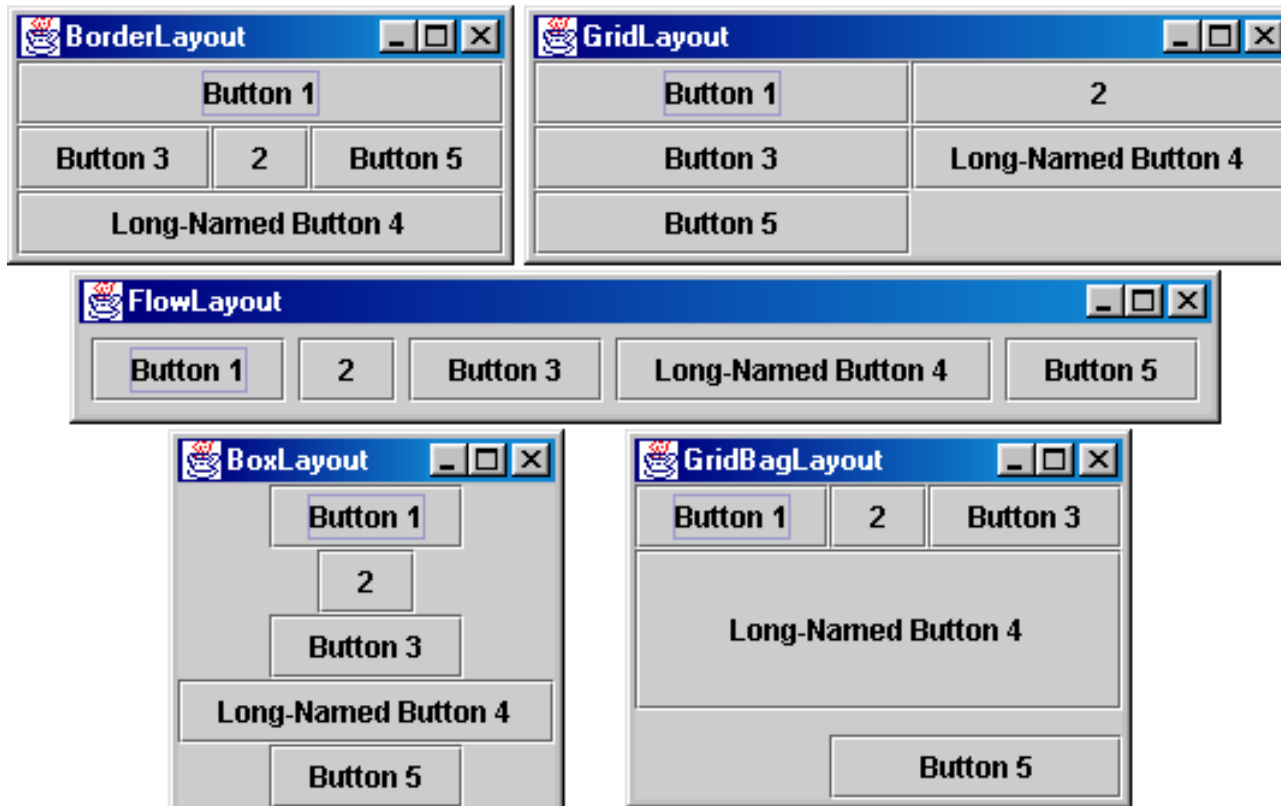
Sizing and positioning

How does the programmer specify where each component appears, how big each component should be, and what the component should do if the window is resized / moved / maximized / etc.?

- **Absolute positioning** (C++, C#, others):
Programmer specifies exact pixel coordinates of every component.
 - "Put this button at (x=15, y=75) and make it 70x31 px in size."
- **Layout managers** (Java):
Objects that decide where to position each component based on some general rules or criteria.
 - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the frame."

Containers and layout

- Place components in a *container*; add the container to a frame.
 - **container**: An object that stores components and governs their positions, sizes, and resizing behavior.



JFrame as container

A JFrame is a container. Containers have these methods:

- `public void add(Component comp)`
`public void add(Component comp, Object info)`
Adds a component to the container, possibly giving extra information about where to place it.
- `public void remove(Component comp)`
- `public void setLayout(LayoutManager mgr)`
Uses the given layout manager to position components.
- `public void validate()`
Refreshes the layout (if it changes after the container is onscreen).

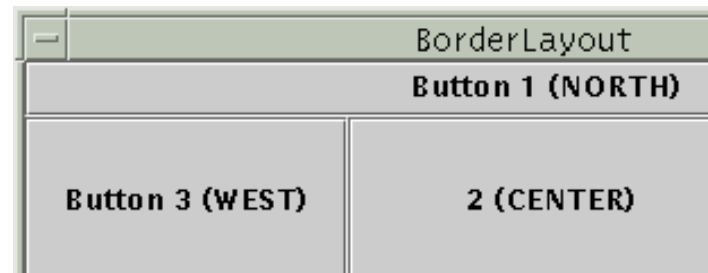
Preferred sizes

- Swing component objects each have a certain size they would "like" to be: Just large enough to fit their contents (text, icons, etc.).
 - This is called the *preferred size* of the component.
 - Some types of layout managers (e.g. `FlowLayout`) choose to size the components inside them to the preferred size.
 - Others (e.g. `BorderLayout`, `GridLayout`) disregard the preferred size and use some other scheme to size the components.

Buttons at preferred size:



Not preferred size:



FlowLayout

```
public FlowLayout()
```

- treats container as a left-to-right, top-to-bottom "paragraph".
 - Components are given preferred size, horizontally and vertically.
 - Components are positioned in the order added.
 - If too long, components wrap around to the next line.

```
myFrame.setLayout(new FlowLayout());  
myFrame.add(new JButton("Button 1"));
```



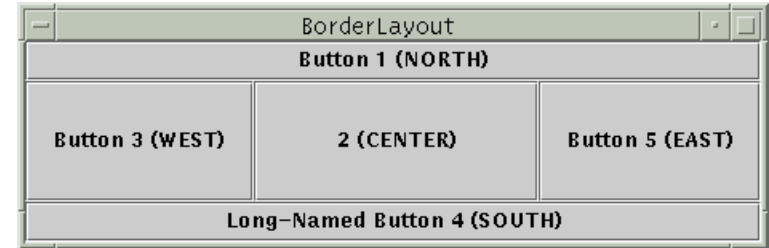
- The default layout for containers other than `JFrame` (seen later).

BorderLayout

```
public BorderLayout ()
```

- Divides container into five regions:

- NORTH and SOUTH regions expand to fill region horizontally, and use the component's preferred size vertically.
- WEST and EAST regions expand to fill region vertically, and use the component's preferred size horizontally.
- CENTER uses all space not occupied by others.



```
myFrame.setLayout(new BorderLayout());  
myFrame.add(new JButton("Button 1"), BorderLayout.NORTH);
```

- This is the default layout for a JFrame.

GridLayout

```
public GridLayout(int rows, int columns)
```

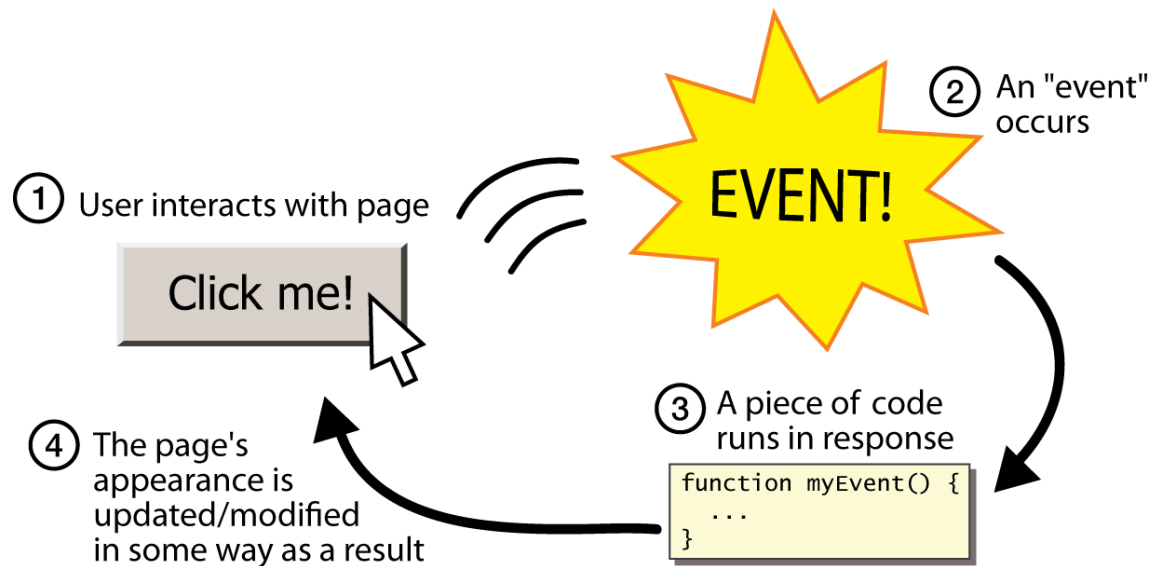
- Treats container as a grid of equally-sized rows and columns.
- Components are given equal horizontal / vertical size, disregarding preferred size.
- Can specify 0 rows or columns to indicate expansion in that direction as needed.



Event Listeners

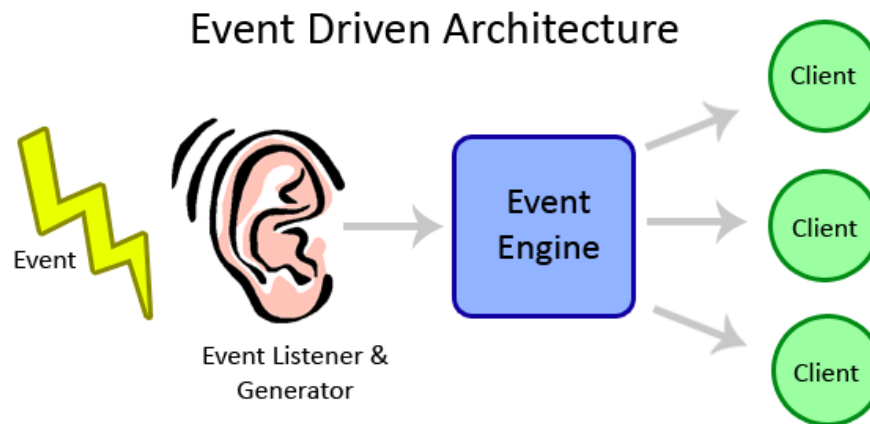
Graphical events

- **event:** An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.
- **listener:** An object that waits for events and responds to them.
 - To handle an event, attach a *listener* to a component.
 - The listener will be notified when the event occurs (e.g. button click).



Event-driven programming

- **event-driven programming:** A style of coding where a program's overall flow of execution is dictated by events.
 - Rather than a central "main" method that drives execution, the program loads and waits for user input events.
 - As each event occurs, the program runs particular code to respond.
 - The overall flow of what code is executed is determined by the series of events that occur, not a pre-determined order.



Event hierarchy

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

- EventObject
 - AWTEvent (AWT)
 - **ActionEvent**
 - TextEvent
 - ComponentEvent
 - FocusEvent
 - WindowEvent
 - InputEvent
 - KeyEvent
 - MouseEvent
- EventListener
 - AWTEventListener
 - **ActionListener**
 - TextListener
 - ComponentListener
 - FocusListener
 - WindowListener
 - KeyListener
 - MouseListener

Action events

- **action event:** An action that has occurred on a GUI component.
 - The most common, general event type in Swing. Caused by:
 - button or menu clicks,
 - check box checking / unchecking,
 - pressing Enter in a text field, ...
 - Represented by a class named `ActionEvent`
 - Handled by objects that implement interface `ActionListener`



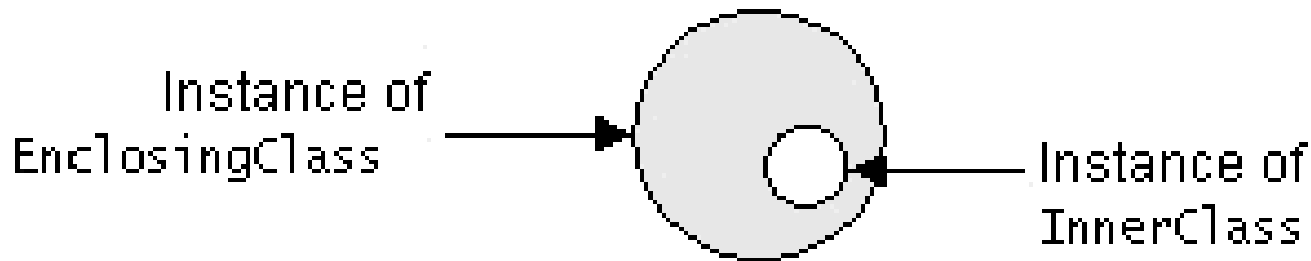
Implementing a listener

```
public class name implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        code to handle the event;  
    }  
}
```

- JButton and other graphical components have this method:
 - `public void addActionListener(ActionListener al)`
Attaches the given listener to be notified of clicks and events that occur on this component.

Nested classes

- **nested class:** A class defined inside of another class.
- Usefulness:
 - Nested classes are hidden from other classes (encapsulated).
 - Nested objects can access/modify the fields of their outer object.
- Event listeners are often defined as nested classes inside a GUI.



Nested class syntax

```
// enclosing outer class
public class name {
    ...

    // nested inner class
    private class name {
        ...
    }
}
```

- Only the outer class can see the nested class or make objects of it.
- Each nested object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
 - If necessary, can refer to outer object as **OuterClassName**.`this`

Static inner classes

```
// enclosing outer class
public class name {
    ...

    // non-nested static inner class
    public static class name {
        ...
    }
}
```

- Static inner classes are *not* associated with a particular outer object.
- They cannot see the fields of the enclosing class.
- *Usefulness:* Clients can refer to and instantiate static inner classes:
`Outer.Inner name = new Outer.Inner (params) ;`

GUI event example

```
public class MyGUI {
    private JFrame frame;
    private JButton btn;
    private JTextField textfield;

    public MyGUI() {
        ...
        btn.addActionListener(new BtnListener() );
    }
    ...

    // When button is clicked, doubles the field's text.
    private class BtnListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            String text = textfield.getText();
            textfield.setText(text + text);
        }
    }
}
```