# Topic 5

## Intro to GUI Programming (AWT/Swing)

## Learning Outcomes

- **Describe** the concept of ***usability*** of a computer interface and how it measured
- **Explain** the pros and cons of a Graphical User Interface (GUI) versus a command line interface
- **Describe and Document Diagrammatically** the OO design of the Java AWT/Swing APIs and **apply** these APIs to create graphical user interface (GUI) code
- **Describe and Document Diagrammatically** the relationship between **Containers** and **Components** in AWT/Swing and **apply** these as building blocks in GUI code
- **Describe** the use of **LayoutManager** classes in AWT/Swing and **apply** the `FlowLayout`, `BorderLayout` and `GridLayout` classes to size/layout a basic UI
- **Design and Code** with Windows, Dialogs and Frames and **Document Diagrammatically** the API composition and application of the `JFrame` and `JDialog` classes
- **Design and Code** with Menus and **Document Diagrammatically** the API composition and application of the `Jmenu`, `JMenuItem` and `JMenuBar` classes
- **Explain** the distinction between Lightweight and Heavyweight components and how this influences the use of AWT/Swing

## Graphical User Interfaces - Introduction

Most (modern) application programs (applications or apps) do not communicate with their users solely via text input and output. They use a **Graphical User Interface (GUI)**.

Most users find a **graphical user interface more convenient and easier to use** than a **text-mode console** system (recognition versus recall).

A **Graphic User Interface** (GUI, pronounced 'gooey') allows interaction via windows containing buttons, menus, control panels, dialog boxes, etc.

However, it is important to note that a **good user interface cannot compensate for a poor program.**

Likewise, an otherwise excellent program **can be rendered unusable or useless** by a poor user interface (see following slide, usability).

Therefore, it is important to design an **effective means** for the **user to interact** with the program.

## Designing for Usability

Usability is a combination of the following user-oriented characteristics:

(Designing the User Interface 3rd Ed., Shneiderman, B., 1998)

- High speed of user task performance.
- Low user error rate.
- Ease of learning.
- User retention over time.
- Subjective user satisfaction.

The user is the final judge of the usability and appropriateness of the interface.

Achieving usability requires attention to two main components:

- The product (interactive software system)
- The process by which the product is developed.

*Is primarily the domain of usability specialists and graphic designers. This course emphasis quality of the user interface code*

* Eclipse is a large Java project with a complex GUI, how many lines of code do you think the latest version has?
https://www.eclipse.org/org/press-release/20160622_neon.php

## Java’s GUI

Java's Abstract Window Toolkit (AWT) and Swing are standard Java libraries of packages and classes designed to provide a framework for programmers to create GUIs. AWT provides the basic GUI classes. Swing extends AWT and provides more advanced GUI components. They share the same concepts and event model. More on Swing in the next slide.

Where available Swing is preferred over AWT however AWT is still used for 2D graphics, fonts, color, layout management and event handling (since Swing does not replace or supersede these AWT features).

This course:

• Covers general software design principles

• Introduces AWT (especially the core event handling and layout management model)

• Introduces basic Swing components for designing the graphical user interface

*NOTE: IBM provides an alternative GUI framework called SWT (Standard Widget Toolkit ) which is not covered in this course (Eclipse is a good example of an SWT project).*

## JavaFX

Java provides a newer GUI library called JavaFX that has a number of new features compared to AWT/Swing:

Included with Java SE 8 (but the API documentation is separate)

Based on presentation metaphor of *stages* containing *scenes* containing *nodes* compared with Containers/Components/LayoutManagers in AWT/Swing

Declarative UI generation using FXML

Declarative UI styles using Cascading Style Sheets (CSS)

Automated Scene Builder tool which graphically (drag & drop) produces FXML

Dependency Injection using `@annotations` for injecting event handlers etc.

Richer support for multimedia, animation, 3D graphics etc.

The ability to be integrated into Swing applications using the `javafx.embed.swing.JFXPanel` class which extends `javax.swing.JComponent`

***For simplicity and to follow a more OO based development approach we use AWT/Swing in this course for GUI development***

***AWT/Swing will remain part of Java SE for the foreseeable future (http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6)***

## Java Applets

Java provides a web based technology called applets that use the same AWT/Swing components as Java desktop applications

However this technology is no longer commonly used for new applications as some web browsers such as Chrome (as of version 42 in 2015) no longer support applets (due to security concerns)

In this course we will write AWT/Swing based desktop application not web based applets and applet behaviour is not examined

However some of the code examples use applets since it provides a smaller code footprint (less starting up code for windows etc.) when demonstrating specific ideas such as event handling

These applets can be run in eclipse by right clicking the appropriate .java file and selecting Run As -> Java Applet

## Swing Basics

Swing is an additional graphical interface toolkit (distributed as part of Java SE since Java 1.2) that provides many improvements over AWT that was introduced with Java 1.0 and refined with Java 1.1.

Swing is built on top of the AWT component set, and relies on it for its basic functionality.

Swing and AWT components, however, should not be used together in the same interface (due to z-order incompatibility - see slide 31).

These libraries are **programmatic** in nature and the components and other classes related to AWT are in the series of packages named *java.awt.**

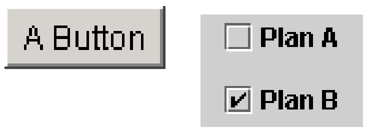
The components and other classes related to Swing are in the series of packages named *javax.swing.**

Swing UI components begin with 'J' to differentiate them from their AWT counterparts; e.g. the Swing button object is JButton, the AWT equivalent is Button.

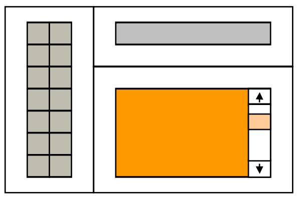
## Developing a GUI

There are **three basic things** that are necessary to develop a graphical user interface using a component (i.e. not java.awt.Graphics) approach:

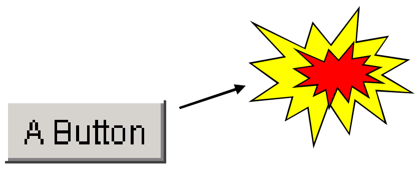
**The GUI components:** these are the buttons, labels, check boxes, etc. which form the interface.



**Component arrangement (Layout Management):** this is the scheme whereby the UI components are arranged to create the interface.



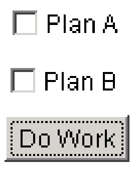
**Response to user requests:** this is the act of associating actions to user requests (known as 'events'.)



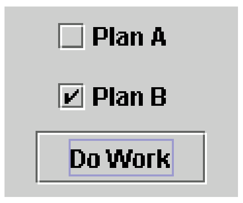
## AWT and Swing Components

There are two sets of components in Java: the **AWT** (Abstract Windowing Toolkit), and the **Swing component set**.

The **AWT component set** is a very **basic set** of user interface items.

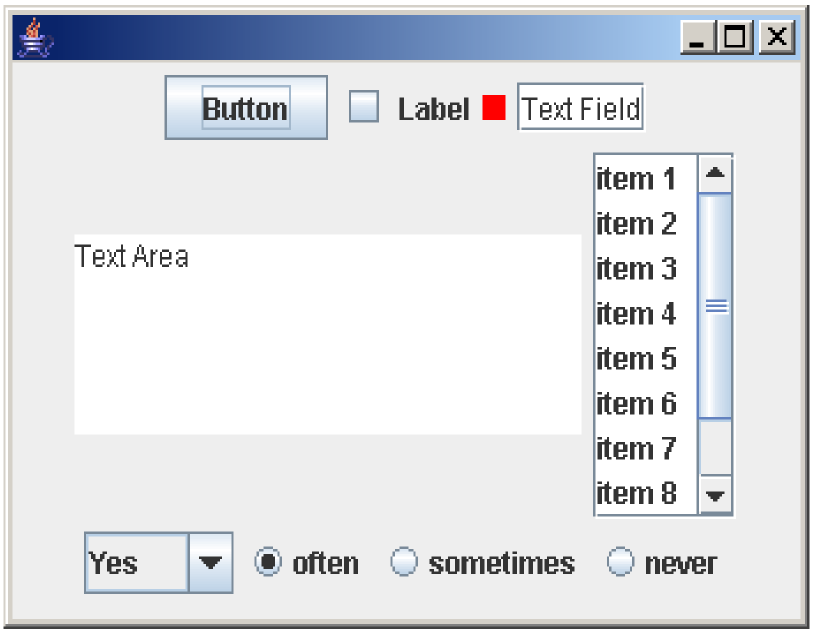


The **Swing component set** (which is an extension of the AWT set) are **far richer** in presentation and functionality, and are preferred over AWT unless you have an older/legacy platform that does not support Swing.



**Mixing** AWT and Swing components **is not recommended**

## GUI Component Example



A ButtonGroup ensures only one item in the group can be selected.

## Containers, Components & Layout Management

- The relationship between containers, their contained components and their layout manager is fundamental to the AWT (and Swing)
- A container is an AWT component that can contain other components (added using the `Container.add(Component)` method)
- The abstract class `java.awt.Container` extends `Component` and thus Containers may also contain other Containers (composite pattern)
- Containers group related components and treat them as a unit, which helps arrange components on the display
- Many AWT components and all Swing components subclass container (e.g. (J)Frame, (J)Window, (J)Panel etc.)

## The Swing Containment Hierarchy

A Swing containment hierarchy typically has at least:

a top-level container: the root container (JRootPane) that holds everything together. e.g. JFrame, JDialog

one or more intermediate containers to simplify the positioning of atomic components. e.g. JPanel, JScrollPane

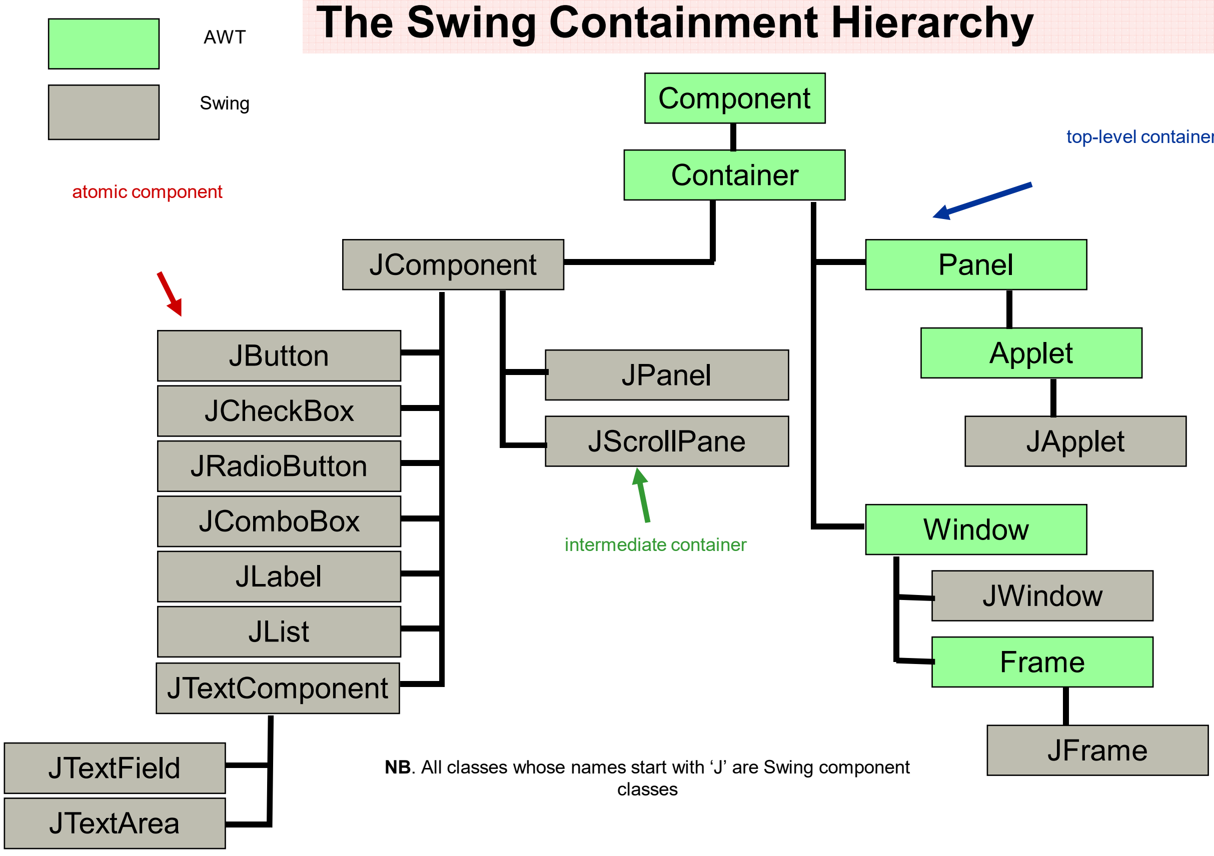
an atomic component: a self-sufficient component not holding other components. eg. JButton, JLabel

***In a Swing application:***

The top-level container must be Swing. e.g, JFrame (not Frame)

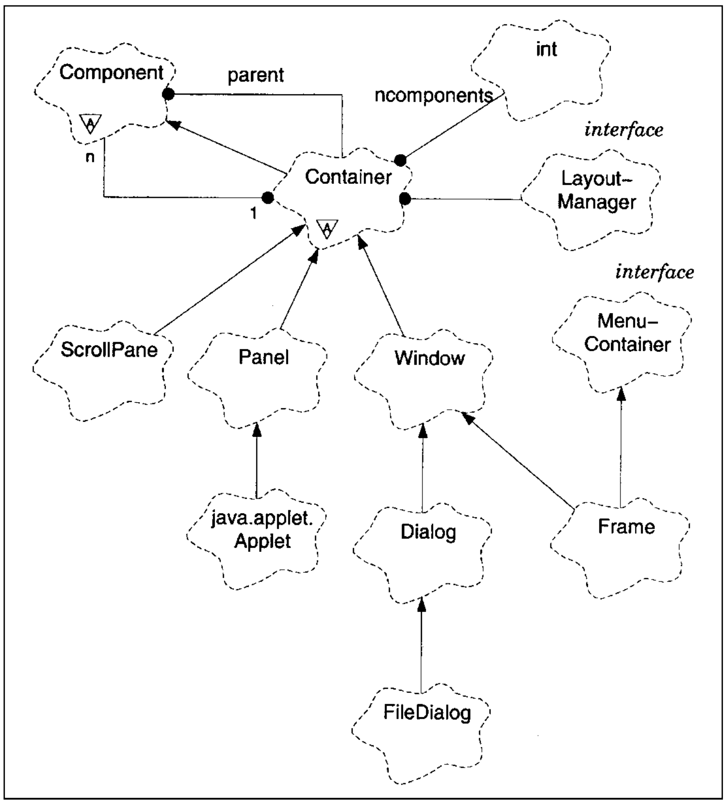
A component cannot be added directly to the top-level container, but must be added to an intermediate container contained in the top-level container, called the content pane. This is done by calling the method public Container. getContentPane() that returns a reference to its content pane (which is actually a JPanel).

## The Swing Containment Hierarchy



**NB**. All classes whose names start with 'J' are Swing component classes

## Class Diagram – Component / Container Relationships



## Layout Managers

- Containers are not responsible for laying out (sizing, positioning etc.) their own components => all containers have an instance of a layout manager
- A Layout Manager is a class that implements the `java.awt.LayoutManager` or `java.awt.LayoutManager2` interface
  - the LayoutManager2 interface allows a component to provide constraints that assist with resizing
  - uses overloaded `Container.add(Component comp, Object constraints)` method
- AWT provides 5 pre-defined layout managers
  - `FlowLayout*`, `BorderLayout*`, `GridLayout*`, `CardLayout` and `GridbagLayout` (* covered in this lecture)
  - Swing adds the `BoxLayout` as well `SpringLayout`, `OverlayLayout` and `GroupLayout` (low level for use with GUI Builders)
- Call Container's `void setLayout(LayoutManager mgr)` method to set a containers layout manager

## Why use Layout Managers?

- Layout managers are used in Java in preference to fixed x,y,width,height positioning:
  - fixed positioning can be achieved by setting the layout manager to 'null'
  - poor solution for windows that need to be resized because of font metrics, placement etc. => must be manually calculated (difficult!)
  - cross platform programming is tedious (because of differences in available fonts, component sizes etc.)
  - layout managers provide solutions for common layout scenarios (i.e. reuse instead of coding from scratch)

## Component Preferred/Minimum Sizes

- components can specify *preferred* and *minimum* sizes that are used by the layout manager

  *default values automatically set by the component (or peer)

- can be changed in AWT but requires subclassing (prior to version 1.5) in order to override two Component methods:
  - `public Dimension getPreferredSize()`
  - `public Dimension getMinimumSize()`
  - Swing (and AWT in Java 1.5+) provides setter methods
- not all layout managers always abide by the components preferred and minimum sizes
- preferred and minimum sizes are usually set by the peer so generally don't need to be set manually
- there is also a *maximum* size but is not used by standard AWT layout managers

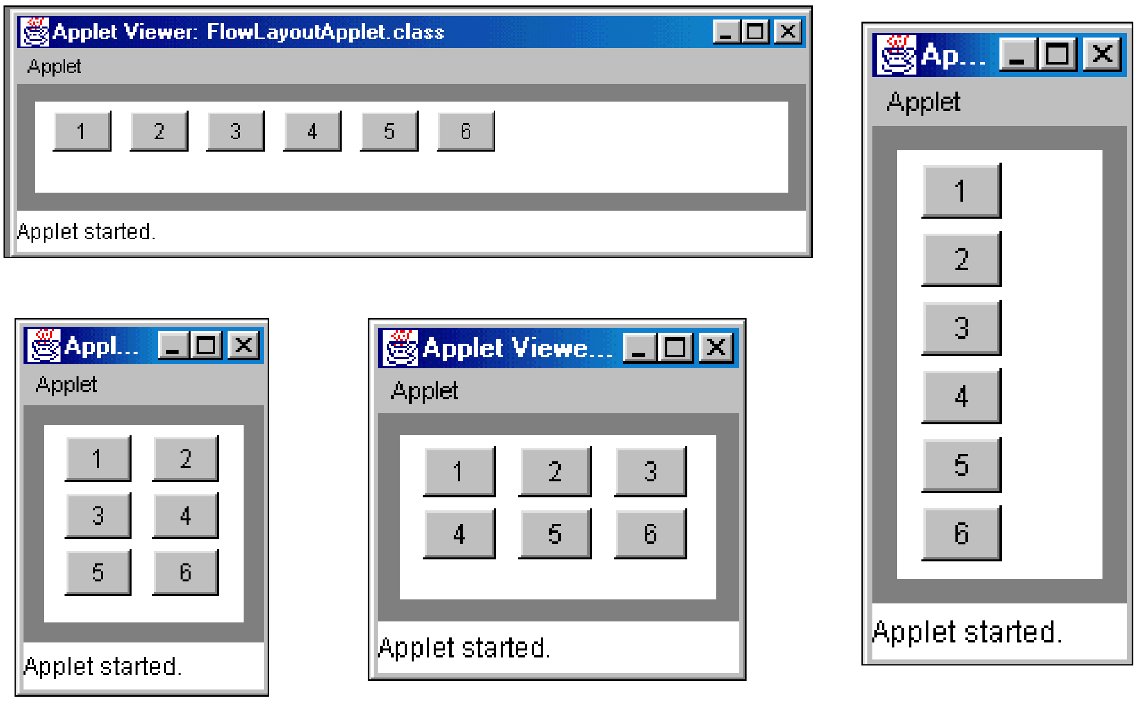
## Flow Layout

- displays components left to right, top to bottom.
- default layout manager for panels
- respects preferred size and height if component has not been explicitly sized with `setSize()`
- can set the spacing between components during construction or with `setHgap(int hgap)` or `setVgap(int vgap)`
- can set the alignment of components during construction or with `setAlignment(int align)`
  - `FlowLayout.CENTER, FlowLayout.LEFT, FlowLayout.RIGHT`

Contructors

- `FlowLayout()`
- `FlowLayout(int align)`
- `FlowLayout(int align, int hgap, int vgap)`

## Flow Layout Resizing



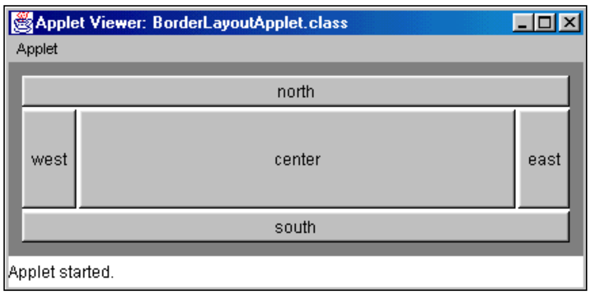
## Border Layout

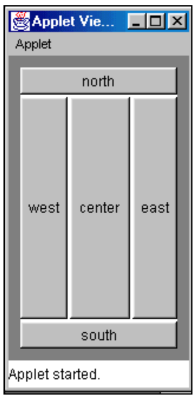
- Lays out at most five components into geometric regions
- Implements `LayoutManager2` interface so requires constraints which are string constants :
  - `BorderLayout.NORTH , BorderLayout.SOUTH , BorderLayout.WEST, BorderLayout.EAST & BorderLayout.CENTER`
- *north* and *south* components are stretched horizontally but preferred height is respected
- *west* and *east* components are stretched vertically but preferred width is respected
- *center* layout gets whatever space is left (preferred size is ignored)
- vertical and horizontal gap set in same manner as FlowLayout

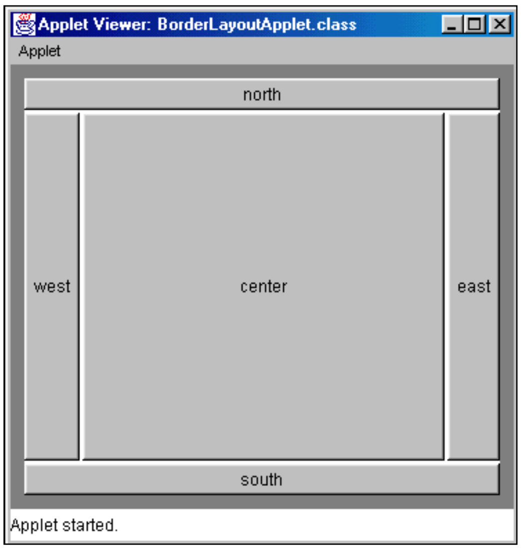
Constructors

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`

## Border Layout Resizing





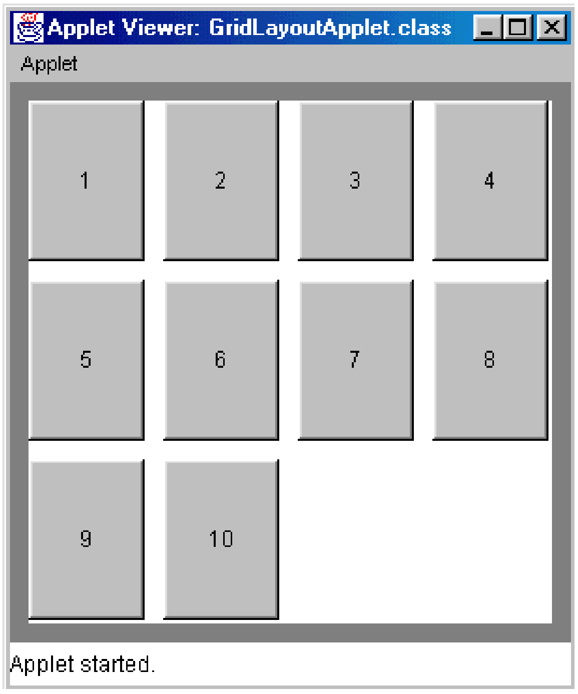


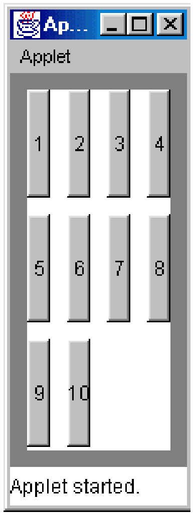
## Grid Layout

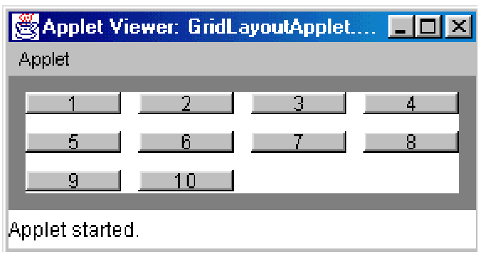
- Lays out components in a grid
- Implements `LayoutManager` interface (i.e. there are no constraints)
- Number of cells are specified during construction
  - if a row or column is specified as 0 then size is automatically computed depending upon number of contained components
  - row and column cannot both be zero (throws exception)
- All cells are the same size (components grow or shrink to fill the display area)

  => *preferred* and *minimum* sizes of contained components are ignored
- Can set vertical gaps and horizontal gaps as with `BorderLayout` and `FlowLayout`
- Constructors:
  - `GridLayout()`
  - `GridLayout(int rows, int cols)`
  - `GridLayout(int rows, int cols, int hgap, int vgap)`

## Grid Layout







## GridBagLayout

- Most flexible but most complicated* of the layout managers
- Implements `LayoutManager2` interface
  - constraint is an instance of `GridbagConstraints` class
- Lays out components in a grid
  - components may span more than one cell
  - number of cells is not specified at construction but rather determined by the constraints which are set for each component
  - size of cells is not specified but is determined in relation to the size of the component it contains
- Constructor
  - `GridBagLayout()`

## Forcing Layout Management

- Sometimes it is necessary to relayout a container
  - e.g. added or removed components from the container, or layout-related information changed
- This is done by
  - Calling `Component.invalidate()` on a component in the container .. Since optimisation can take place
  - Calling `Container.validate()` on the container
  - In Swing this can also be done by calling `JComponent.revalidate()` on a component which also forces relayout
- As a last result if repainting still not complete call `Component.repaint()`
  - this does a full redraw so is not efficient
  - there is also a version with bounds parameters

## Windows

- `java.awt.Window` provides a superclass with common functionality that is extended by both `Frame` and `Dialog`
- it subclasses `Container` allowing components to be added with the `add()` method
- generally not instantiated directly
- provides a number of general purpose window methods
  - inherited by `Frame` and `Dialog`
  - recall `WindowListener` and `WindowEvent` classes that are used with Windows

## java.awt.Frame

- `java.awt.Frame` is a subclass of `java.awt.Window`
  - provides a border, title and optional menubar
  - may be resizable (depends on implementation)
  - can be minimized to an icon (depends on implementation)
  - can have an associated icon image: `setIconImage(Image image)`
- constructor: `Frame(String Title)`
  - *Title* represents the text to be displayed in the title bar
- initially invisible and remains so until `setVisible(true)` method is called
- Swing `JFrame` extends `java.awt.Frame` and has same basic behaviour
  - main difference is components are added to the contentPane [retrieved via `getContentPane()`] not directly to `JFrame`
  - usually add a container (`JPanel`) to contentPane and then manage components in the `JPanel`

## java.awt.Dialog

- `java.awt.Dialog`
  - similar to frame (provides a border and title and is resizable)
  - however it cannot have a menu or be iconized
  - must be anchored to a frame (same as Window)
- Dialogs can be modal => when `setVisble(true)` is called:
  - input to the Dialog's ancestors (usually a `Frame`) is blocked
  - the thread that launched the dialog is suspended until the dialog is closed
  - non-modal behaviour is default if not specified with constructor
- AWT provides limited Dialog support
  - `FileDialog` is the only custom dialog
  - message, yes/no and question dialogs must be hand coded
  - **Swing provides JOptionPane class with static methods for creating basic dialogs**
  - e.g. `JOptionPane.showMessageDialog(...)`
- Relationship between Swing `JDialog` and AWT `Dialog` is same as `Frame/JFrame`

## Sizing and Moving Windows

- All windows (i.e. frames and dialogs) inherit from `Component` => use the methods of component class for sizing etc.

```
void setLocation(int x, int y)
void setSize(int width, int height)
void setBounds(int x, int y, int width, int height)
e.g. Frame f=new Frame("Test");
           f.setSize(100,100);
           f.setLocation(0,0);
           f.show();
```

- Usually only size top level containers (Frame, Dialog etc.) otherwise use layout management for sizing

## Adding a Menu Bar to a JFrame

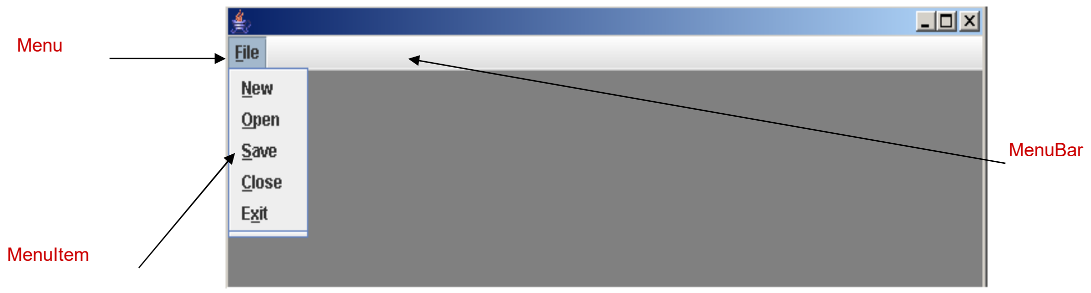
A Menu bar consists of three different parts – the J/MenuBar component itself, one or more J/Menu components (which are added to the J/MenuBar) and a set of J/MenuItem components for each Menu component which represent the menu options.

To add a menu bar to a JFrame as shown in the example below you need to create the J/MenuBar, create one or more Menu components, fill each of the Menu components with their corresponding J/MenuItem components and add each of the (now filled) Menu components to the J/MenuBar.

JMenu extends JMenuItem to facilitate submenus (Composite pattern again!)

Once the MenuBar has been set up it can be added to the JFrame using the method:

void setJMenuBar(JMenuBar menubar);



## Lightweight and Heavyweight Components

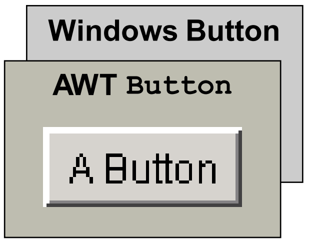
Most Swing components are “lightweight” components, compared to the AWT (where they are “heavyweight”.)

Lightweight components are those that are written completely in Java, and do not rely at all on the host operating system.

Heavyweight components depend on the related ‘peer’ component of the native operating system.

This is why AWT-based programs look slightly different on each platform, but Swing programs maintain the same look (and also why these component sets should not be mixed together.)

The GUIs we develop in this course will be Swing based.



Heavyweights rely on their native peers.