UI architecture evolution and object-oriented design

Week 2 Lecture 2

16.01.2008

Lyn Bartram

lyn@sfu.ca

Some slides adapted from K. Edwards, Georgia tech

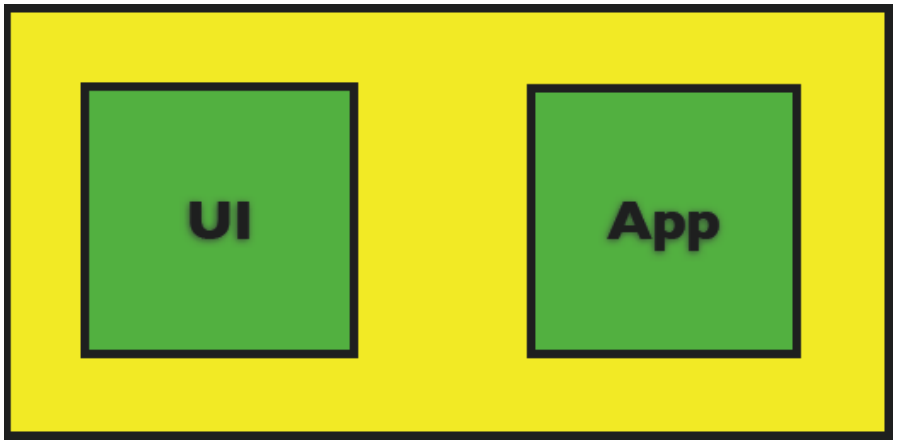
SCHOOL OF INTERACTIVE ARTS + TECHNOLOGY

Architectural goals

1. Separation of concerns
   - Traditionally think of the “UI” as only one component of the system
2. Multiplicity of presentation options
   - Pluggable, quasi independent views
3. Coordination for interaction
   - Coherent framework for mapping and controlling input to logic to output

Programmer's perspective

- The "UI" was/is typically viewed as one component of the overall system
  - The part that "deals with the user"
  - Separate from the "functional core" (the application)



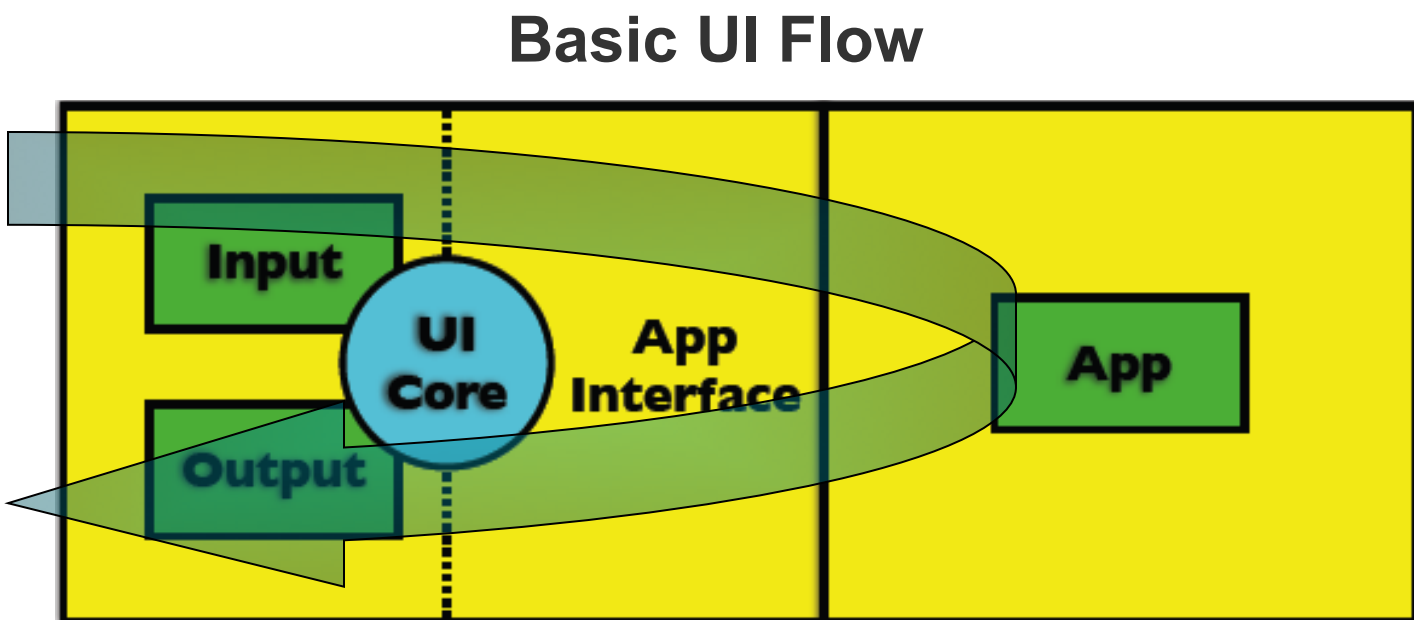
Hmm, in practice

This is very hard to do in real-world application environments

- More and more interactive programs are “tightly coupled” to the UI
  - Programs are structured around the UI concepts and the flow of behaviour
  - Lower level support needs to be present to enable higher-level behaviour

- At the same time, more and more interactive applications are a combination of distributed services
  - UI has to know something about the back end architecture to function

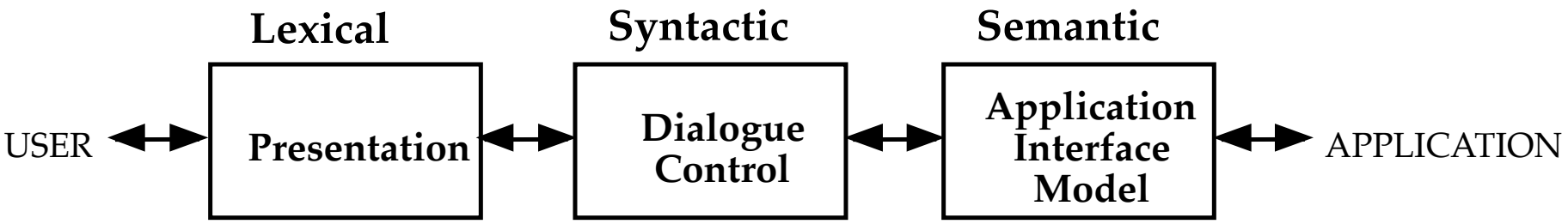
Conceptual overview of the UI: recall



How would you architect this?

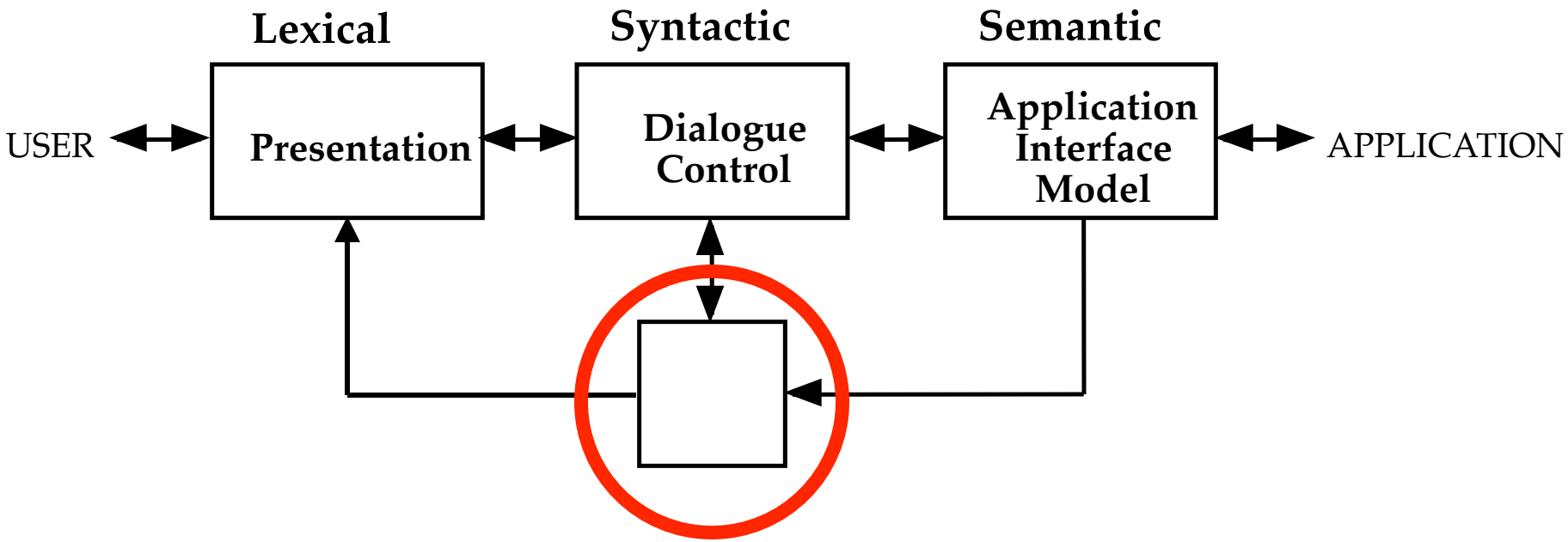
- Tempting to architect the system around these boxes
  - One module for input, one for output, etc
  - Has been tried (the “Seeheim model”)
  - Didn’t work well

Seeheim model



Result of 1985 workshop in Seeheim, Germany
Basis of the UIMS approach

Problem:
Rapid Semantic Feedback



Big box architectures don't work well because...

- Modern interfaces: set of quasi-independent agents
  - Each "object of interest" is separate
  - e.g. a button
    - produces "button-like" output
    - acts on input in a "button-like" way
    - etc.
  - Each object does its tasks based on
    - What it is
    - What its current "state" is
      - Context from prior interaction or application

The philosophical shift

- Compiler mentality
  - Lexical/Syntactic/Semantic
  - Seeheim, ARCH
- Object design model
  - Interactive system as collection of *objects*

Object-oriented architecture

- *Interactor* objects (“object of interest”)
  - AKA components, controls, widgets
  - Example: an on-screen button

- Each object implements each aspect
  - Common methods for
    - Drawing output (button-like appearance)
    - Handling input (what happens when button is clicked)

- Objects organized hierarchically with *inheritance*
  - reflecting spatial containment relationships
  - Reflecting behaviour flow

Implementation support

1. windowing systems
   - Device independence
   - Multiple tasks (simultaneous, distinct user activity)
2. Dialogue control
3. interaction toolkits
4. user interface management systems (UIMS)

Implementation support

1. windowing systems

2. Interaction and control
   - Modal, tight “read-evaluate-act” loop
   - Notification or event-based
   - Paradigm for how application is controlled

3. interaction toolkits

4. user interface management systems (UIMS)

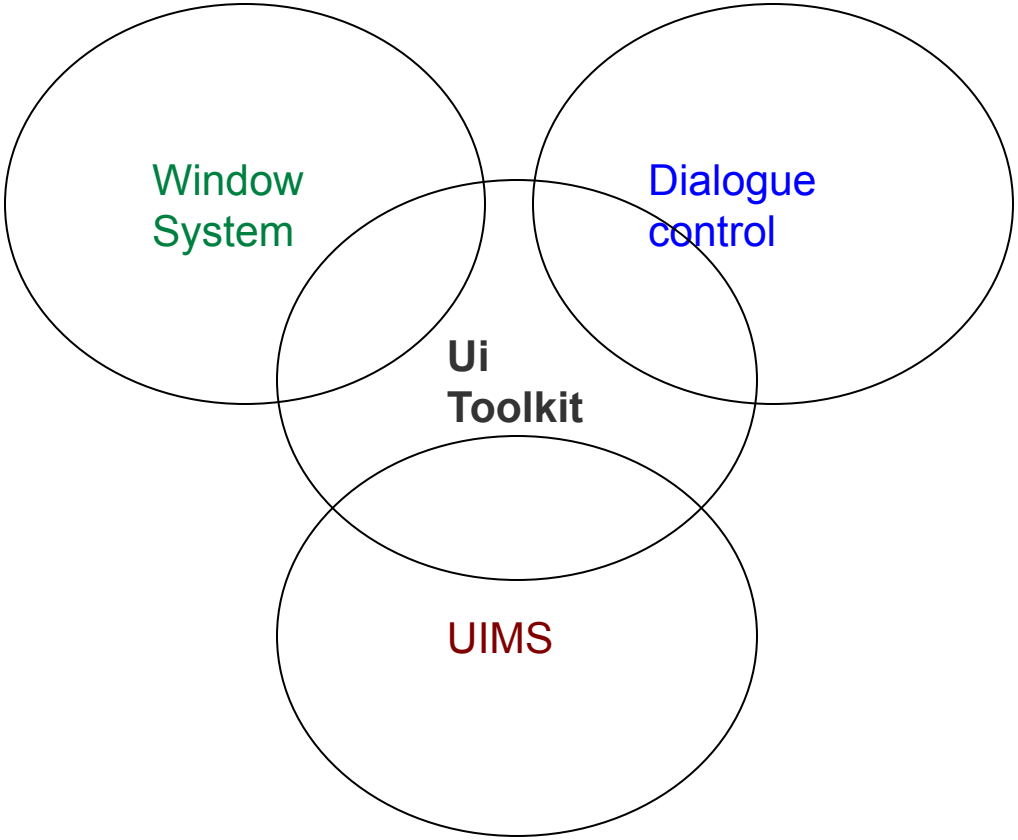
Implementation support

1. windowing systems
2. Dialogue control
3. Interface toolkits
   - Programming interaction objects and behaviours (UI toolkits)
   - Component-based systems
   - Libraries of widgets and services
   - UI “builders”
4. user interface management systems (UIMS)

Implementation support

1. windowing systems
2. Dialogue control
3. interaction toolkits
4. user interface management systems (UIMS)
   - Conceptual architectures for separation
   - Techniques for expressing dialogue

Modern Implementation support



UI Toolkits (GUI Toolkits)

- System to provide development-time and runtime support for UIs
  - Event- driven programming
  - Widgets/components
  - Interactor tree
- Specific interaction techniques
  - Libraries of interactors
  - Look and feel
- How the UI connects to the application (the API)
- Describes how most GUIs work
- We will be using SWING, the Java GUI toolkit
- We will not be using UI builders

Toolkit detail (roadmap)

- Input
  - Picking
    - Figuring out what interactors are active under a given screen point
  - Events
    - Dispatch
    - Translation
    - Handling and exceptions
    - This is where a LOT of the work goes

- **Abstractions**
  - **Separable architecture**
  - **Extensible constructs**

2D interface programming toolkits

- Tcl/Tk
- Motif/UIL
- IDEs (e.g. VB, MSVC++/MFC), Visual Studio
- Java JFC Swing (Java2 - JDK >= 1.2)
- JBuilder and other Java IDEs
- Unity™, game engines
- Scripters (ActionScript), sort of Processing
- Specialised tools

Toolkit detail (roadmap)

- Hierarchy management
  - Create, maintain and tear down the tree/graph of interactor objects
- Geometry management
  - Dealing with coordinate systems
  - Windows and graphics
- Interactor status
- Output/display
  - Layout
  - Drawing and redrawing (damage management)
  - Images and text

Before we Start…

- The Java GUI toolkit requires you to know about, and understand:
  - Classes / Objects
  - Method Overloading
  - Inheritance
  - Polymorphism
  - Interfaces
  - How to read the Java2 API Documents

Object-oriented design

- How to minimize complexity of individual objects?

- Three general approaches
  - Inheritance
  - Composition
  - Aggregation

OO terminology & objectives

- Encapsulation
  - don't touch my private data
  - get/set it using my public/package methods
- Inheritance
  - a parent provides for her/his children
  - children add to a parent's knowledge
- Polymorphism
  - Java figures out family relationships:
    when the parent knows best or
    when the child knows more than the parent

Inheritance in Java

- super class or parent, subclass or child
- adds functionality to an existing class
  - `class` Child `extends` Parent

- use of the keyword `super`
  - not `this` object's constructor/variable/method, but the one from the `super` class.

More generally, objects and classes

- Java is an object-oriented language
  - Logical translation of design objects
- *Objects* are built from *classes*
  - Classes are the blueprint, objects are built from the blueprint
  - Objects are called *instances* of classes
- Each element or component in your application should be a class
  - In fact, the `JButton` you add to your `JFrame` is a separate class

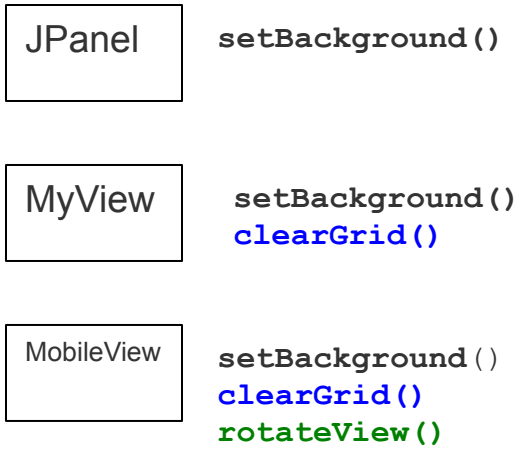
Inheritance

- Simplest ideal : All concerns in one object/class
  - inherit / override them separately
  - works best with multiple inheritance
  - example: draggable_icon
    - inherit appearance from “icon” (output aspects only)
    - inherit behavior from “draggable” (input aspects only)

- Java class has only one “parent”
  - `interface` and `abstract` class support multiple inheritance

Inheritance

- Practical implementation
  - Objects can be derived from other objects, inheriting all the functionality and access of the parent
  - `Class MyView extends JPanel()`
  - `Class MobileView extends MyView()`
- allows for increased specialisation
- Interfaces and abstract classes extend it



FoodProcessor.java example

```
interface Sliceable
{
   public void slice();
}
abstract class Fruit
{
  public void printName() {System.out.println( "Fruit" );}
   public void printCalories() { System.out.println( "No Fat" ); }
   public abstract void printShape();
}
```

FoodProcessor.java example

```
/* an implementation of the interface Sliceable */
class Apple extends Fruit implements Sliceable
{ public void slice() { System.out.println( "Slice the
public void printName() {System.out.println( "Apple" );}
 public void printShape() {System.out.println( "almost
like a sphere" );}
}

class Banana extends Fruit implements Sliceable
{public void slice() { System.out.println( "Slice the
banana into 7 pieces." ); }
public void printName() {System.out.println( "Banana" );}
   public void printShape() {System.out.println( "almost
like a crescent" );}
```

Composition

- Combine interactive objects at larger scale than interactors

- `Container` objects
  - e.g., row and column layout objects
- Containers can also add input & output behavior to things they contain

Aggregation

- Different concerns in separate objects
- Combine (aggregate) them into sets of objects
  - Treat collection as “the interactor”
  - General approach: *design patterns*

- Classic design pattern: “model-view-controller” (MVC)
  - Also presentation-abstraction-control (PAC)
  - Localise activity and data in separate classes

What is Swing?

- A part of The Java Foundation Classes
  - Swing
    - Look and feel
    - Accessibility
    - Java 2D (Java 2 onwards)
    - Drag and Drop
    - etc
- Can be used to build Standalone Apps as well as Servlets and Applets

Assignment 1

- Use Swing to create the skeleton of an energy journaling application

- Due Friday, Sept. 19, at midnight

- Monday's lecture :Swing introduction
- Monday workshop: Swing exercises

Getting started with Swing (1)

- Compiling & running programs
  - Swing is standard in Java 2 (JDK >= 1.2)
  - Use:
    - ‘javac <program.java>’ && ‘java <program>’
    - Or Eclipse

Getting started with Swing (2)

- Swing, like the rest of the Java API. is subdivided into *packages*:
  - **`javax.swing, javax.accessibility, javax.swing.border`** …
- At the start of your code - always
  - **`import javax.swing;`**
  - **`import javax.swing.event;`**
- Most Swing programs also need
  - **`import java.awt.*;`**
  - **`import java.awt.event.*;`**

Using Swing and AWT

- ***Do not*** mix Swing and AWT components
  - Lightweight and heavyweight components cause side effects
- If you know AWT, put ‘J’ in front of everything
  - AWT: **`Button`**
  - Swing: **`JButton`**
- Swing does all that AWT does, but better and there’s much more of it

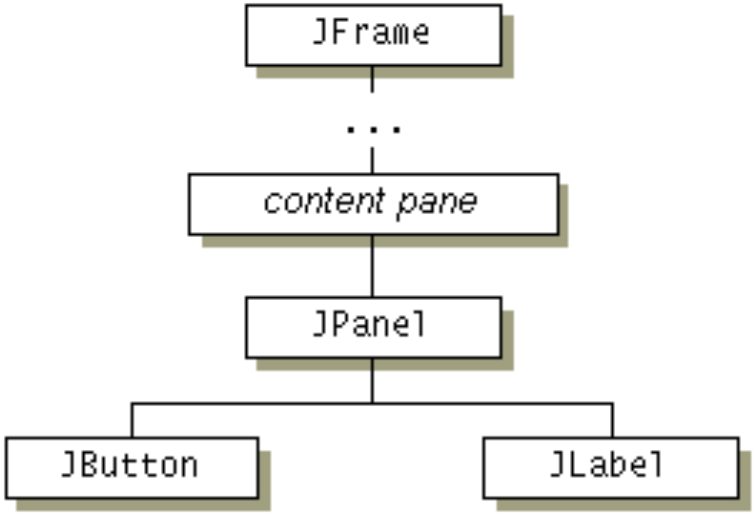
A typical Swing program

- Consists of multiple parts
  - Containers
  - Components
  - Events
  - Graphics
  - (Threads)

- We will look at each in turn

A simple Swing program - Containers

- Containers



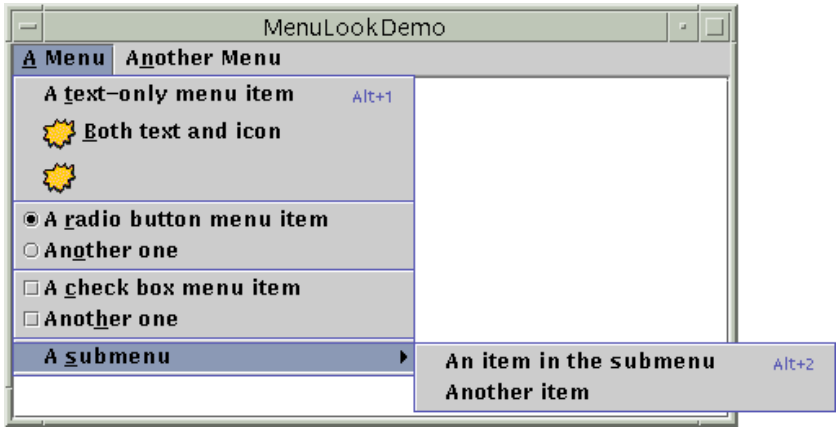
- JFrame, JDialog, JApplet

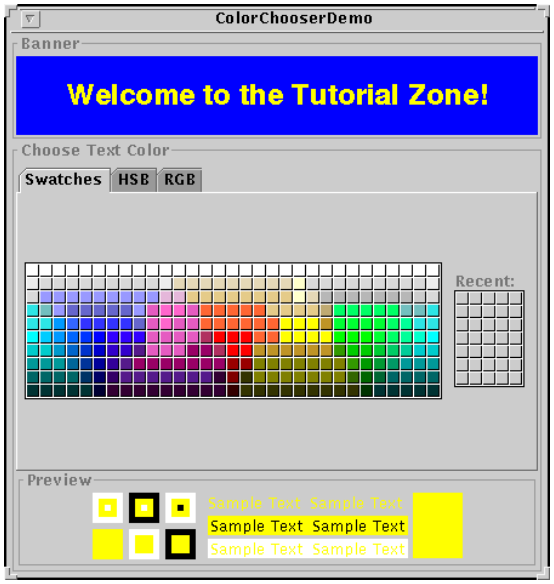
Remember this about Containers:

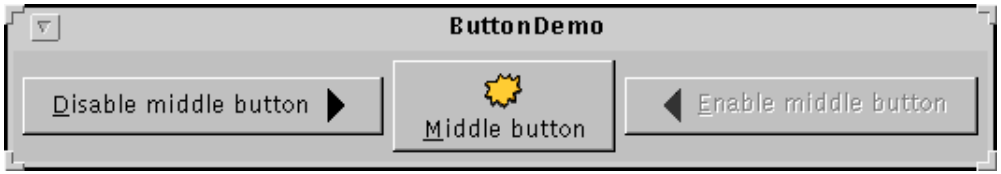
- The structure of containers is your design decision and should always be thought through in advance
  - particularly for managing components
  - nesting containers.
  - A component can only be in one container!
- Failure to do so usually either results in a messy interface, messy code or both.

A simple Swing program - Components

- Components







Components

- Components are added to Containers
- A Component can only live in one Container
- Components get added to the Container's *content pane*
  - In the case of `JFrame`, using the `setContentPane()` method.
  - Exception: we can add a menu bar to a Container

Remember this about Components:

- There are many components that make your job much easier.
- Often, you will be able to customise an existing Swing component to do a job for you, instead of having to start from scratch
  - Eg can extend (inherit from) the **`JButton`** class and 'paint' a new button over the top

The JComponent class

- All functions of interactors encapsulated in this base class

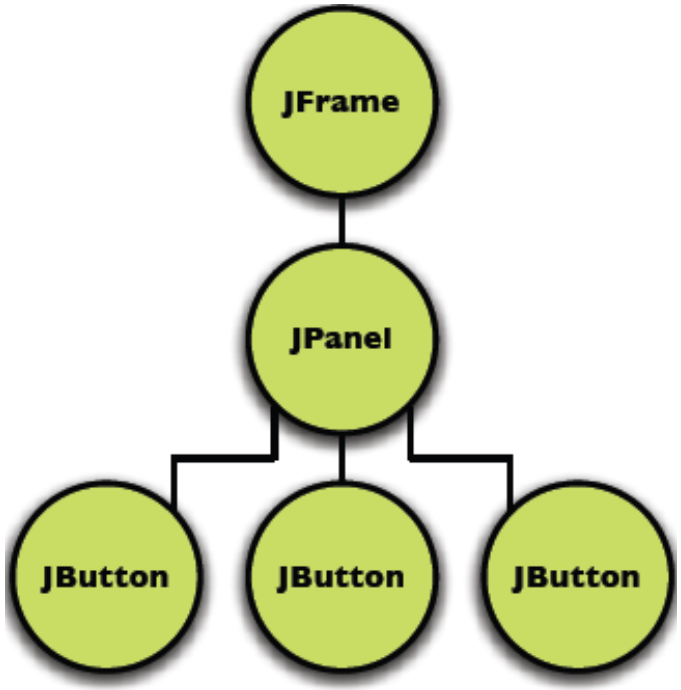
- Javax.swing.Jcomponent;
- Objects inherit from this class
- Methods for:
  - Hierarchy management
  - Geometry
  - Status
  - Layout
  - (re)drawing
  - picking

- In subclasses and other parts of the toolkit

- Input dispatch and handling
- Application interface
- Pluggable look and feel
- Undo support
- accessibility

Hierarchy Management

- Swing interfaces are trees of components
- To make something appear you must add it to the tree
  - In the right order
- Swing takes care of many of the details from there
  - Screen redraw
  - Input dispatch



Hierarchy Management

- Lots of methods for manipulating the tree
- `add(), remove(), getComponent(), isAncestorOf(), getChildCount()`

- Common mistake
  - If nothing shows up, make sure you have added it
  - `setVisible()` !

Geometry

- Every component maintains its own local geometry
- Bounding box:
  - getX(), getY(), getWidth(), getHeight()
  - 0,0 is at parent's upper left corner
  - setSize(), setLocation(), setBounds(), getSize(), getLocation(), getBounds()
- All drawing happens within the bounding box
  - Including output of children
- Drawing is relative to top-left corner
  - Each component has own coordinate system
  - Need to know dimensions of component

Object status

- Each component maintains information about its state
  - isEnabled(), setEnabled()
  - IsVisible(), setVisible()
- Lots of other methods of more limited importance

Each component handles

- Layout (coming later)
- Drawing
  - Component knows how to (re)create its appearance based on its current state
  - Understanding how the drawing methods are called is a little complicated
  - We'll return to this later, but for now

Each component handles

- Responsible for painting 3 items in order
    1. Component
    2. Borders
    3. Children
  - `paintComponent()`, `paintBorder()`, `paintChildren()`
  - These are the only places to draw on the screen! BUT
  - Automatically called by Jcomponent's `paint` method, itself called by the Swing repaintManager (figures out damaged regions)

  - So the key method (and the only one for now) that you call in each component is **`repaint()`**

Damage(Change) Management

- Damage: areas of a component that need to be redrawn
  - Generic term
- Sometimes computed automagically by RepaintManager
  - Window overlap, resize
- Other times: you need to flag changes or damage yourself to tell the system that something in the internal state has changed and the onscreen image needs to be updated
  - E.g. changing the colour of a label
- Managing damage yourself
  - `Repaint(Rectangle r)`
  - `<componentName>.repaint();`
  - Puts the indicated area or component on the the internal queue of regions to be redrawn

Assignment 1

- Goal: learn how to use basic Swing components
- Familiarise yourself with toolkit

- Application: a simple journaling tool

- Use `Jframe` (windows) , panes, buttons and labels to build simple windowed tool

- Base of assignments 2 and 3
- We will develop examples in the tutorial

How to Learn Swing

- Don’t even try.
- Learn general framework principles and design styles.
- Then use the API reference, and Swing Tutorials to discover detailed usage of each component.

How to read Java Docs (1)

- Java 2 (1.6) API Reference available at:
  - http://java.sun.com/javase/6/docs/api/
- Split into 3 Sections (html frames):
  - Top Left: Packages
  - Bottom Left: Classes in Packages
  - Main Frame: Information about selected Class

How to read Java Docs (2)

- General idea is find class, and examine main frame for information.

- Main frame pages split into sections:
  - Package hierarchy & implemented interfaces
  - Class Description, and links to more info
  - Nested Class Summary – Detail in separate page
  - Fields - 2 types Class (static) and instance, plus fields inherited from parent classes / interfaces
  - Constructor Summary
  - Method Summary & inherited methods from parents
  - Detailed info on all summary sections