
JavaOne℠
Sun's Worldwide Java Developer Conference

1.0 Libraries Technical Overview

Jonni Kanerva
Frank Yellin

JavaSoft

Outline

- Introduction
- Functionality in the 1.0 Libraries
- Distributed design of the Java™ platform
- Instructive oddities
- Design patterns
- Extensibility
- Wrapup

Approach

- Yes: explore the universe
  - General principles and overview
  - Instructive oddballs and treats
  - Larger-scale patterns
- No: even-handed skim survey

Background Reading

The Java™ Application Programming Interface, Vols. 1 & 2

James Gosling, Frank Yellin, The Java Team

Addison-Wesley, 1996

Central Themes

- Functionality
  - What can you do with 1.0 java.*?
- Design
  - The pieces
  - How they fit together and cooperate
- Extensibility
  - java.* provides a simple, ubiquitous, extensible base for your programs.

API Slogans

- Simplicity is power,
  accuracy is leverage
- Provide partial designs, ready to be
  woven together
- The simple should be simple,
  the complex should be possible

API Goals

- Function
  - Overall scope is good
  - Important problems are solvable
- Partition
  - Divide and conquer complexity
  - Pieces are understandable
  - Pieces are changeable

API Goals (cont.)

- Names
  - Simple, direct, systematic
  - Reduce chances for misunderstanding
- Extensible
  - Easy to specialize
  - Easy to connect
- Platform independent
- Implementable

Good API = ???

- Learnable
  - Easy to learn, easy to remember
- Usable
  - Easy to code to, easy to build tools on
- Flexible
  - Allow different solutions and solution styles

Next

- Introduction
- Functionality in the 1.0 Libraries
- Distributed design of the Java™ platform
- Instructive oddities
- Design patterns
- Extensibility
- Wrapup

Functionality in the 1.0 Libraries

- Java™compatible library = Java package
  - Classes
  - Interfaces
  - Subpackages
- Functionality spread through 8 packages

Packages in the 1.0 API

- java.lang: core language support
- java.io: input/output streams, data types
- java.net: networking support
- java.util: HashTable, StringTokenizer…
- java.awt: cross-platform window toolkit
- java.awt.peer: interfaces to native GUI
- java.awt.image: image processing
- java.applet: applets and applet contexts

Package java.lang

- 21 classes, 2 interfaces, 40 excep/errors
- Class wrappers for primitive data types:
  - Boolean, Integer, Double…
- Classes for core language concepts:
  - String, Thread, Object
- Access to system resources:
  - Process, Runtime, System, …
- Imported by every Java-powered program

Package java.io

- 23 classes, 3 interfaces, 5 sections
- Byte-oriented stream abstraction for input and output
- Mix and match filtering
- Cross-platform file abstraction
- Stream tokenizer

Package java.net

- 11 classes, 3 interfaces, 5 exceptions
- URLs
- URL connections
- Sockets
- Internet addresses

Package java.util

- 10 classes, 2 interfaces, 2 exceptions
- Generic utilities:
  - Vector, HashTable, Stack, Enumeration,
  - BitSet, StringTokenizer
  - Date
  - …

Package java.awt

- 42 classes, 2 interfaces, 1 exception, 1 error
- GUI elements:
  - Button, TextField, Window, Menu, …
- Event handling
- Fonts and font metrics
- Graphics
- Colors

Package java.awt.peer

- 0 classes, 22 interfaces
- Interfaces for communicating with native GUI elements:
  - ButtonPeer, TextFieldPeer, …
- Decouples AWT classes from platform-specific toolkit implementations

Package java.awt.image

- 9 classes, 3 interfaces
- Image creation
- Image filters
- Color mapping

Package java.applet

- 1 class, 3 interfaces
- The Applet class
- Applet audio and images
- Applet context:
  - Applet-browser relationship
  - Inter-applet communication

Distributed Design of the Java™ Platform

- Java platform = runtime + language + classes
- java.* standard API
  - On all Java-compatible platforms
  - Write once, compile once, run everywhere

Classes Complement the Language and Runtime

- java.* classes are integrated with core language mechanisms:
  - Primitive data types
  - Operators
  - Class and interface membership
  - Control flow
  - Runtime and environment

Primitive Data Types and java.* Classes

- boolean: Boolean
- char: Character
- int, long: Integer, Long
- float, double: Float, Double

```
Double d = new Double(3.14159);
double simplePi = d.doubleValue();
```

Operators and java.* Classes

- Language: +
- Classes: String, StringBuffer

```
String s = “a” + “b”;
String s = new
   StringBuffer().append(“a”).append(“b”).toString();
```

Class/Interface Types and java.* Classes

- Language:
  - class, interface, implements, extends, instanceof
- Classes: Class, Object

```
interface Fooable { public void fooIt(); }
class Foo implements Fooable { public void fooIt() {} }
Fooable obj = new Foo();
obj.getClass()    ==>    “class Foo”
obj.getClass().getSuperclass().isInterface()    ==>    false
Class.forName("Fooable").isInterface()    ==>    true
```

Multithreading and java.* classes

- Language:
  - synchronized (methods and blocks)
- Classes:
  - Thread, ThreadGroup,
  - Object: **wait** and **notify** methods

```
Thread thread1, thread2;  Runnable r;
thread1 = new Thread(r);
thread2 = new Thread(r);
thread1.start();  thread2.start();
```

Control Flow and java.* Classes

- Language:
  - try, catch, finally, throws
- Classes:
  - Throwable, Error, Exception

```
try {
    sleep(500);
} catch (InterruptedException e) {
    System.out.println(“e = “ + e);
}
```

Runtime/Environment and java.* Classes

- Runtime/Environment:
  - Object allocation, security, garbage collection
- Classes:
  - Class, ClassLoader, Object
  - Runtime, System, SecurityManager

```
String s1 = (String)”Hello”.getClass().newInstance();
String s2 = System.getProperty(“java.version”);
```

Recap

- Integrated Java-compatible platform includes java.* classes
  - Standard on all Java-compatible systems
- java.lang.* is the core of the core
  - Most tightly integrated
  - Imported automatically

Instructive Oddities

- Typical class in java.*:
  - Public, concrete, subclassable
  - Has instance methods, instance variables
- Variety is the spice of life:
  - A classy class
  - A very protected class
  - A very abstract class
  - An abstract but not abstract class
  - A half class, half language primitive

java.lang.Math – A Classy Class

- Two numerical constants:
  - Math.E, Math.PI
- Range of standard math functions
  - All as class methods:

    public static double tan(double a)
- Declared as final – no subclasses
- No constructor
- No instances

java.lang.ClassLoader – A Very Protected Class

- Key class for security:
  - Methods and constructor accessible only from your own subclass
  - Only one method can/must be overridden

protected ClassLoader()

protected final void resolveClass(Class c)

protected final Class findSystemClass (String name)

protected final Class defineClass(byte[] data, int offset, int length)

protected abstract Class loadClass (String name, boolean resolve)

java.lang.Number – A Very Abstract Class

- Abstract superclass for number objects:
  - Integer, Long, Float, Double

```
public abstract class Number {
    public abstract int intValue();
    public abstract long longValue();
    public abstract float floatValue();
    public abstract double doubleValue();
}
```

java.awt.Component – Abstract or Not?

- Contains no abstract methods
- Declared as an abstract class
- Cannot be instantiated

Array – Half Class

- Not in any package
- One final instance variable: length
- Cannot be extended (subclassed)
- Superclass is Object
- Inherits methods from Object

(new int[5]).getClass().getSuperclass() ==> java.lang.Object

Design Patterns

- Weaving partial designs together
- Interactions of classes, interfaces, and instances
- Design units within larger picture
- Tool for understanding

Recommended Reading

Design Patterns: Elements of Reusable Object-Oriented Software

Gamma, Helm, Johnson, Vlissides

Addison-Wesley, 1995

Decorator – The Pattern

- Extend functionality of an object
  - Not statically through subclassing
  - By wrapping it in another object, a decorator
- Decorator's interface is superset of decoratee's
- Decorator forwards some requests to decoratee

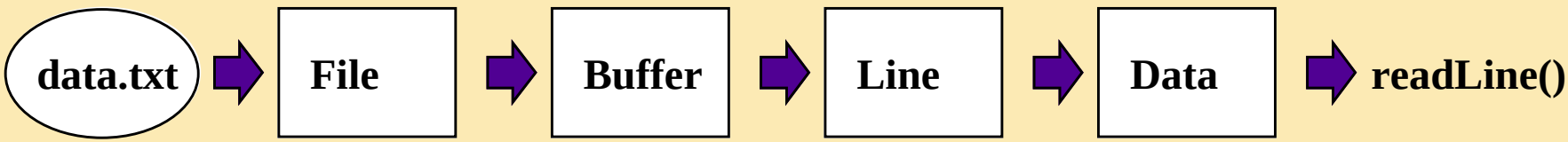
Decorator Example: java.io Input And Output

- Byte-oriented stream input and output
  - Base classes are InputStream, OutputStream
- Mix and match filtering:
  - Filter…, Buffered…, Data…, LineNumber...
- FilterInputStream.read():
  - Invokes read() on the decoratee InputStream:

```
return in.read();
```

Buffered, Numbered, Line Input from a File

```
FileInputStream in1 = new FileInputStream("data.txt");
BufferedInputStream in2 = new BufferedInputStream(in1);
LineNumberInputStream in3 = new LineNumberInputStream(in2);
DataInputStream in4 = new DataInputStream(in3);
String line;
while ((line = in4.readLine()) != null) {
    System.out.println(in3.getLineNumber() + ": " + line);
}
in4.close();
```



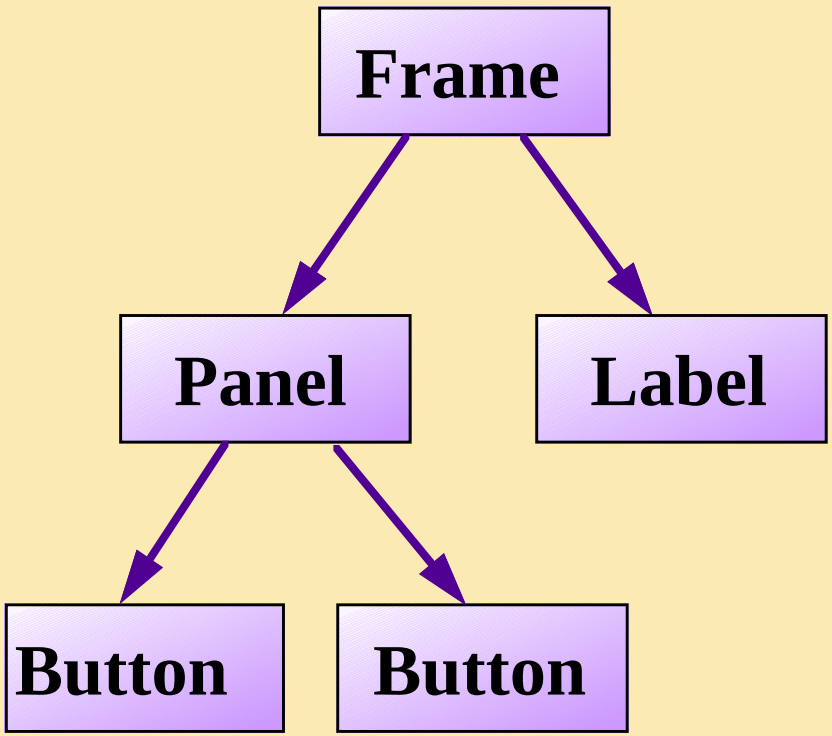
Composite – The Pattern

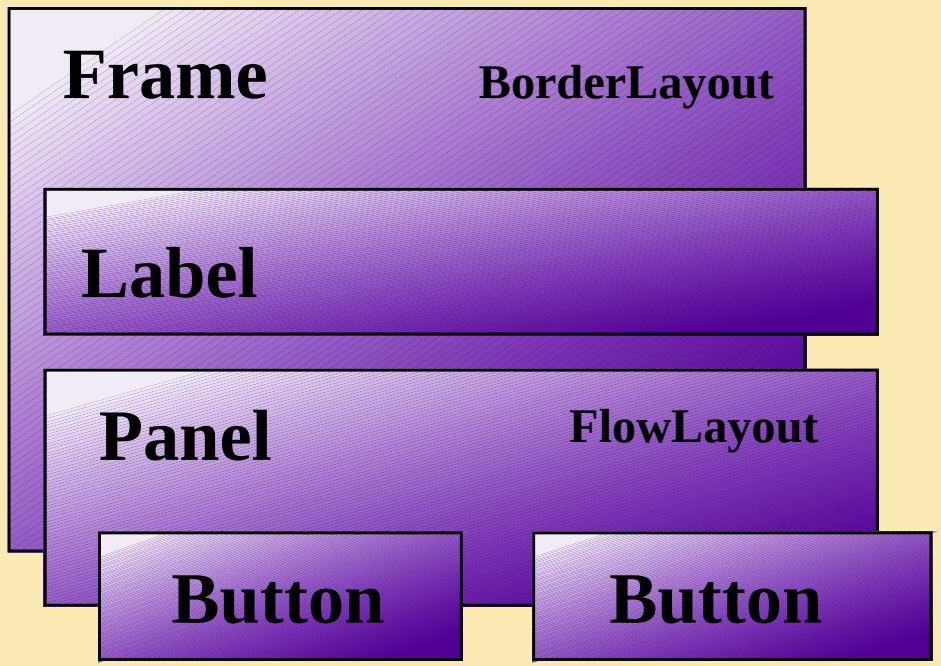
- “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” (p. 163)

Composite Example: java.awt Component and Container

- Abstract superclass: Component
  - Presence on screen, size, location
  - Receive, handle, and deliver events
  - Most AWT GUI elements inherit from Component
- Container is subclass of Component
  - Contains a group of components (“children”)
  - Can create arbitrarily deep containment hierarchy

Component-Container Trees





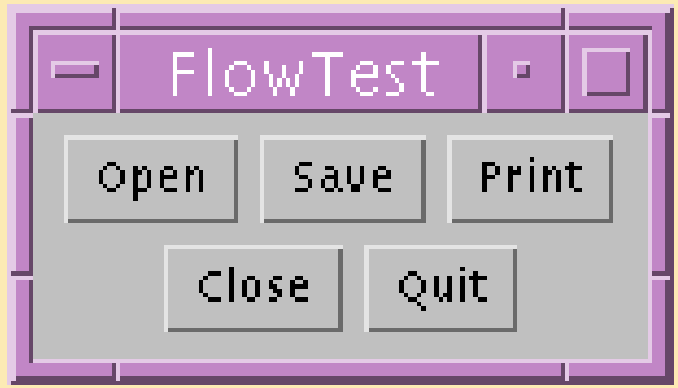
Strategy – The Pattern

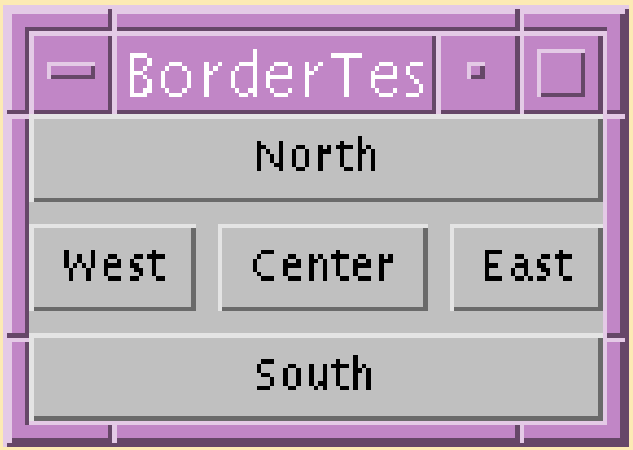
- “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.” (p.315)

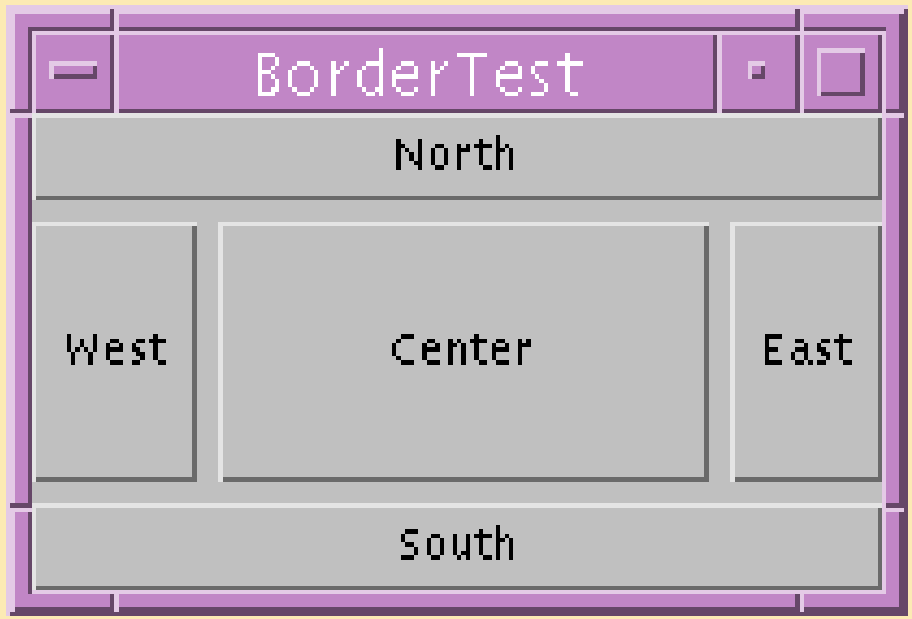
Strategy Example: java.awt.LayoutManager

- How to place components in a container
- Dynamic constraint-based layout
- Interface with 5 methods:
  - Add…, remove…, layout…, minimum…, preferred…
- java.awt package provides:
  - BorderLayout, CardLayout, FlowLayout, GridLayout, GridBagLayout

Flow and Border Layouts



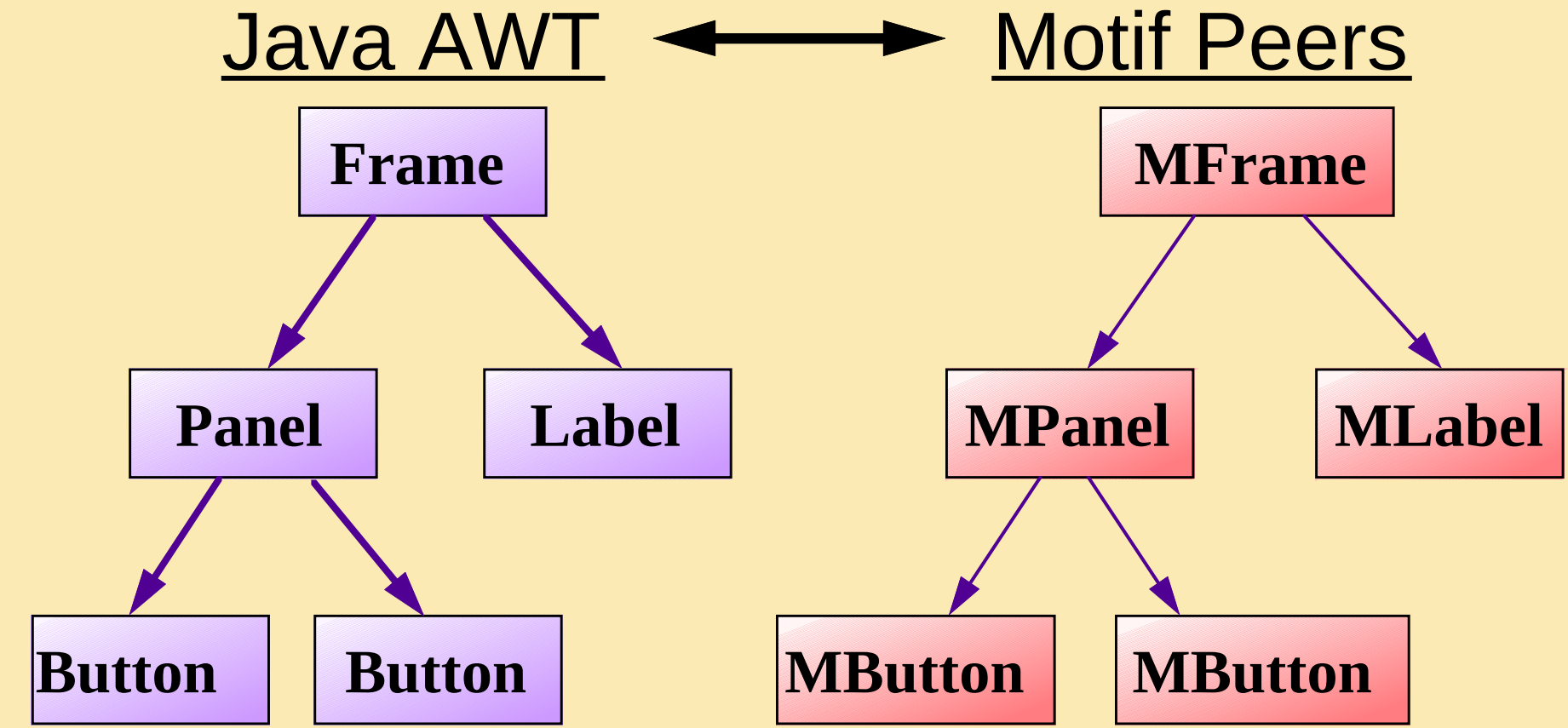




Bridge – The Pattern

- “Decouple an abstraction from its implementation so that the two can vary independently.” (p. 151)

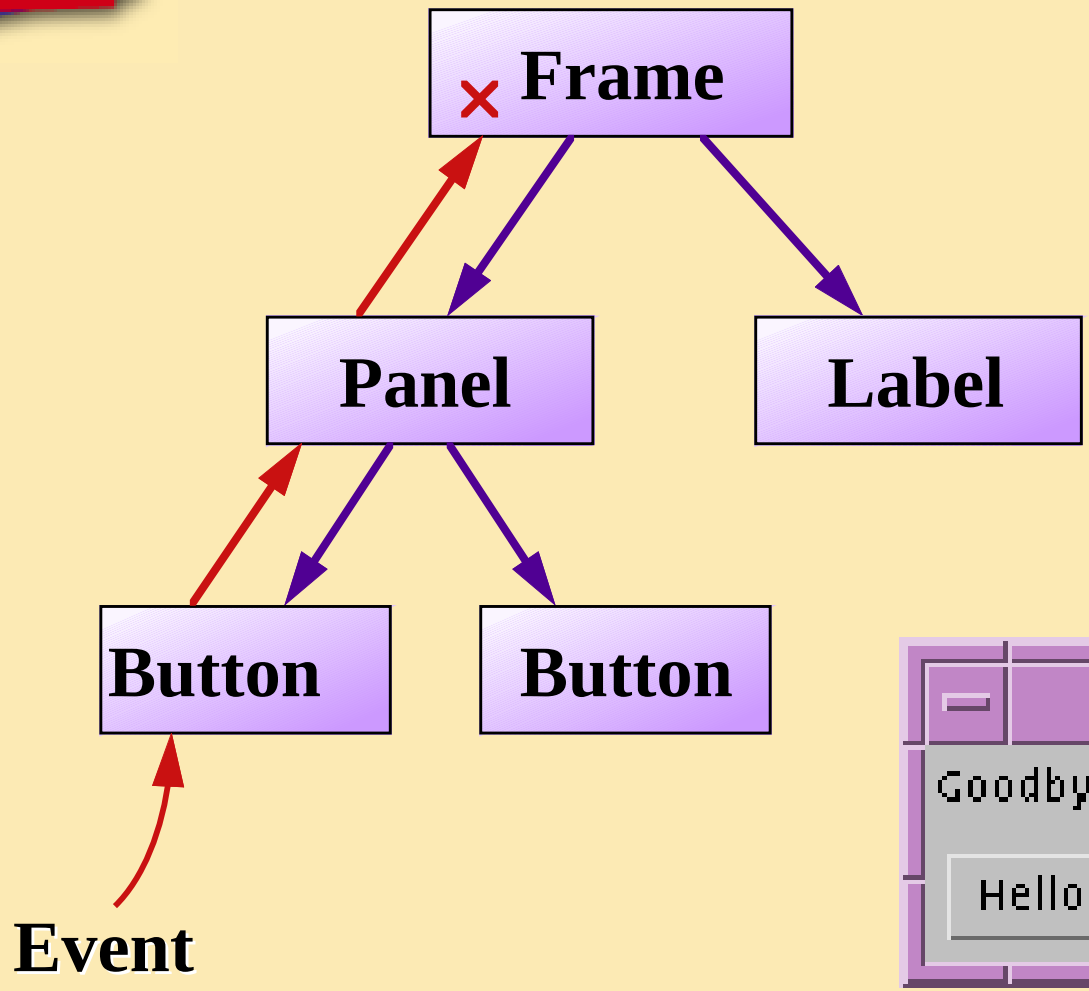
Bridge Example: java.awt Components and Peers



Chain of Responsibility – The Pattern

- “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.” (p. 223)

Chain of Responsibility Example: AWT Events



Extensibility

- Extensibility pervades java.*
  - Subclass a concrete class
  - Subclass an abstract class, prespecified holes
  - Implement an interface

Extend a Concrete Class

- Six favorite concrete classes to extend
  - java.applet.Applet: custom applets
  - java.awt.Canvas: custom GUI components
  - java.awt.Panel: custom GUI containers
  - java.awt.Frame: custom top-level windows
  - java.lang.Thread: custom execution thread
  - ???

Extend an Abstract Class

- extend java.io.InputStream
  - Implement: **read()**
- extend java.lang.ClassLoader
  - Implement: **loadClass(String, boolean)**
- extend java.awt.Graphics
  - 29 abstract methods to implement
  - Example: write your own PSGraphics class

Implement an Interface

- As part of the class's duties
  - java.lang.Runnable in an Applet subclass
- As all of the class's duties
  - java.awt.LayoutManager

Upcoming Extensions

- Security: digital certificates, authentication
- Multi-media: 2-D, 3-D, video, audio
- JDBC: database access and connectivity
- Remote Objects: remote method invocation
- Persistent Objects
- Electronic Commerce

Wrapup – 1.0 java.*

- Simple, ubiquitous, extensible base for your programs:
  - Integrated with language
  - Available on all Java-compatible platforms
- Goals and progress:
  - Learnable, usable, flexible, platform independent, implementable