COMP 303 - Lecture Notes for Week 7 - Java Object Model

- Slides edited from, Object-Oriented Design Patterns, by Cay S. Horstmann
- Original slides available from: http://www.horstmann.com/design_and_patterns.html
- Modifications made by Laurie Hendren, McGill University
- Topics this week:
    - The Java Type System
    - Type Inquiry
    - The `Object` Class
    - Shallow and Deep Copy
    - Serialization
    - Reflection
    - The Java Beans Component Model

Types

- Type: set of values and the operations that can be applied to the values
- Strongly typed language: compiler and run-time system check that no operation can execute that violates type system rules
- Compile-time check

```
Employee e = new Employee();
e.clear(); // ERROR
```

- Run-time check:

```
e = null;
e.setSalary(20000); // ERROR
```

Java Types and Values

Types

- Primitive types:
  `int short long byte`
  `char float double boolean`
- Class types
- Interface types
- Array types
- The `null` type
- Note: `void` is not a type

Java Values

- value of primitive type
- reference to object of class type
- reference to array
- `null`
- Note: Can't have value of interface type

Subtype Relationship

S is a subtype of T if

- S and T are the same type
- S and T are both class types, and T is a direct or indirect superclass of S
- S is a class type, T is an interface type, and S or one of its superclasses implements T
- S and T are both interface types, and T is a direct or indirect superinterface of S
- S and T are both array types, and the component type of S is a subtype of the component type of T
- S is not a primitive type and T is the type `Object`
- S is an array type and T is `Cloneable` or `Serializable`
- S is the `null` type and T is not a primitive type

The ArrayStoreException

- `Rectangle[]` is a subtype of `Shape[]`
- Can assign `Rectangle[]` value to `Shape[]` variable:

  ```
  Rectangle[] r = new Rectangle[10];
  Shape[] s = r;
  ```

- Both `r` and `s` are references to the same array
- That array holds rectangles
- The assignment
  `s[0] = new Polygon();`
  compiles
- Throws an `ArrayStoreException` at runtime
- Each array remembers its component type

Wrapper Classes

- Primitive types aren't classes
- Use wrappers when objects are expected
- Wrapper for each type:

```
Integer Short Long Byte
Character Float Double Boolean
```

- Example: `ArrayList`

```
ArrayList numbers = new ArrayList();
numbers.add(new Integer(13));
int n = ((Integer)numbers.get(0)).intValue();
```

Enumerated Types

- Finite set of values
- Example: `{ SMALL, MEDIUM, LARGE }`
- Java has no syntax for enumerated types
- Can fake them with integer constants

  ```
  public static final int SMALL = 1;
  public static final int MEDIUM = 2;
  public static final int LARGE = 3;
  ```

- Not typesafe

  ```
  int size = LARGE;
  size++;
  ```

Typesafe Enumerations

- Class with fixed number of instances

```
public class Size
{
  private Size(String name)
   { this.name = name;
   }
  private String name;
  public static final Size SMALL =
    new Size("SMALL");
  public static final Size MEDIUM =
    new Size("MEDIUM");
  public static final Size LARGE =
    new Size("LARGE");
}
```

- Private constructor!
- `String` field not necessary (but convenient for `toString`, serialization)
- Typical use:

```
Size imageSize = Size.MEDIUM;
if (imageSize == Size.SMALL) . . .
```

Type Inquiry

- Test whether `e` is a `Shape`:
  `if (e instanceof Shape) . . .`
- Common before casts:
  `Shape s = (Shape) e;`
- Don't know exact type of `e`
- Could be any class implementing `Shape`
- If `e` is `null`, test returns `false` (no exception)

The `Class` Class

- `getClass` method gets class of any object
- Returns object of type `Class`:
- `Class` object describes a **type**

```
Object e = new Rectangle();
Class c = e.getClass();
System.out.println(c.getName()); // prints java.awt.Rectangle
```

- `Class.forName` method yields `Class` object:

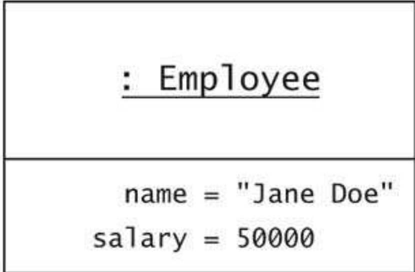
```
Class c = Class.forName(java.awt.Rectangle);
```

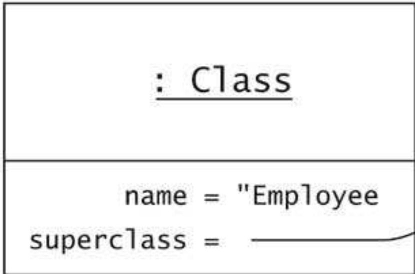
- `.class` suffix yields `Class` object:

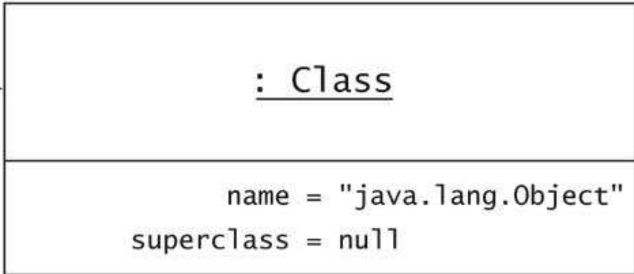
```
Class c = Rectangle.class; // java.awt prefix not needed
```

- `Class` is a misnomer: `int.class`, `void.class`,`Shape.class`

An `Employee` Object vs. the `Employee.class` Object







Type Inquiry

- Test whether `e` is a `Rectangle`:
  `if (e.getClass() == Rectangle.class) . . .`
- Ok to use `==`
- A unique `Class` object for every class
- Test fails for subclasses
- Use `instanceof` to test for subtypes:
  `if (x instanceof Rectangle) . . .`

Array Types

- Can apply `getClass` to an array
- Returned object describes an array type

```
double[] a = new double[10];
Class c = a.getClass();
if (c.isArray())
   System.out.println(c.getComponentType());
      // prints double
```

- `getName` produces strange names for array types

```
[D for double[])
[[java.lang.String; for String[][]
```

Object: The Cosmic Superclass

- All classes extend `Object`
- Most useful methods:
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`
  - `int hashCode()`

The `toString` Method

- Returns a string representation of the object
- Useful for debugging
- Example: `Rectangle.toString` returns something like
  `java.awt.Rectangle[x=5,y=10,width=20,height=30]`
- `toString` used by concatenation operator
- `aString + anObject`
  means
  `aString + anObject`**`.toString()`**
- `Object.toString` prints class name and object address
  `System.out.println(System.out)`
  yields
  `java.io.PrintStream@d2460bf`
- Implementor of `PrintStream` didn't override `toString`:

Overriding the `toString` Method

- Format all fields:

```
public class Employee
{
   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + "]";
   }
   ...
}
```

- Typical string:

```
Employee[name=Harry Hacker,salary=35000]
```

Overriding `toString` in Subclass

- Format superclass first

```
public class Manager extends Employee
{
   public String toString()
   {
      return super.toString()
         + "[department=" + department + "]";
   }
   ...
}
```

- Typical string

```
Manager[name=Dolly Dollar,salary=100000][department=Finance]
```

- Note that superclass reports actual class name

The `equals` Method

- `equals` tests for equal *contents*
- == tests for equal *location*
- Used in many standard library methods
- Example: `ArrayList.indexOf`

```
/**
   Searches for the first occurrence of the given argument,
   testing for equality using the equals method.
   @param elem an object.
   @return the index of the first occurrence
   of the argument in this list; returns -1 if
   the object is not found.
*/
public int indexOf(Object elem)
{
   if (elem == null)
   {
      for (int i = 0; i < size; i++)
         if (elementData[i] == null) return i;
   }
   else
   {
      for (int i = 0; i < size; i++)
         if (elem.equals(elementData[i])) return i;
   }
   return -1;
}
```

Overriding the `equals` Method

- Notion of equality depends on class
- Common definition: compare all fields

```
public class Employee
{
   public boolean equals(Object otherObject)
      // not complete--see below
   {
      Employee other = (Employee)otherObject;
      return name.equals(other.name)
         && salary == other.salary;
   }
   ...
}
```

- Must cast the `Object` parameter to subclass
- Use == for primitive types, `equals` for object fields

Overriding `equals` in Subclass

- Call `equals` on superclass

```
public class Manager
{
   public boolean equals(Object otherObject)
   {
      Manager other = (Manager)otherObject;
      return super.equals(other)
         && department.equals(other.department);
   }
}
```

Not all `equals` Methods are Simple

- Two sets are equal if they have the same elements *in some order*

```
public boolean equals(Object o)
{
   if (o == this) return true;
   if (!(o instanceof Set)) return false;
   Collection c = (Collection) o;
   if (c.size() != size()) return false;
   return containsAll(c);
}
```

The `Object.equals` Method

- `Object.equals` tests for identity:

```
public class Object
{
   public boolean equals(Object obj)
   {
      return this == obj;
   }
   ...
}
```

- Override `equals` if you don't want to inherit that behavior

Requirements for `equals` Method

- *reflexive*: `x.equals(x)`
- *symmetric*: `x.equals(y)` if and only if `y.equals(x)`
- *transitive*: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`
- `x.equals(null)` must return `false`

Fixing `Employee.equals`

- Violates two rules
- Add test for `null`:
  `if (otherObject == null) return false`
- What happens if `otherObject` not an `Employee`
- Should return `false` (because of symmetry)
- Common error: use of `instanceof`
  ```
  if (!(otherObject instanceof Employee)) return
  false;
     // don't do this for non-final classes
  ```
- Violates symmetry: Suppose `e, m` have same name, salary
  `e.equals(m)` is `true` (because `m instanceof Employee`)
  `m.equals(e)` is `false` (because `e` isn't an instance of `Manager`)
- Remedy: Test for class equality
  ```
  if (getClass() != otherObject.getClass())
  return false;
  ```

The Perfect `equals` Method

- Start with these three tests:

```
public boolean equals(Object otherObject)
{
   if (this == otherObject) return true;
   if (otherObject == null) return false;
   if (getClass() != otherObject.getClass()) return false;
   ...
}
```

- First test is an optimization

Hashing

- `hashCode` method used in `HashMap`, `HashSet`
- Computes an `int` from an object
- Example: hash code of `String`

```
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = 31 * h + s.charAt(i);
```

- Hash code of `"eat"` is 100184
- Hash code of `"tea"` is 114704

Hashing

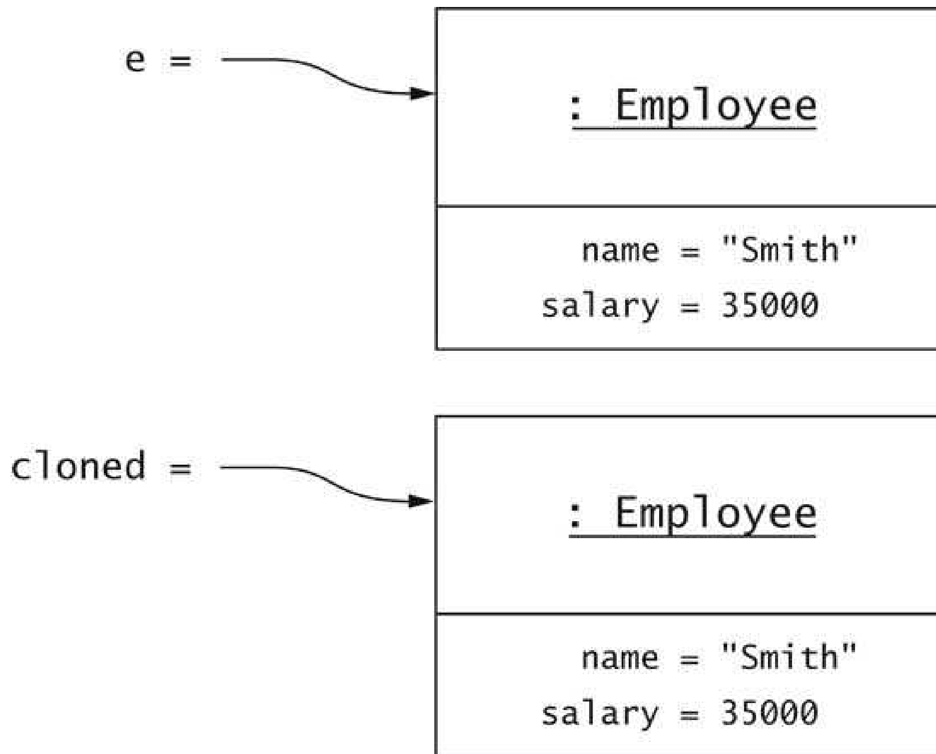
- Must be compatible with `equals`:
  if `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- `Object.hashCode` hashes memory address
- *NOT* compatible with redefined `equals`
- Remedy: Hash all fields and combine codes:

```
public class Employee
{
   public int hashCode()
   {
      return name.hashCode()
         + new Double(salary).hashCode();
   }
   ...
}
```

Shallow and Deep Copy

- Assignment (`copy = e`) makes shallow copy
- Clone to make deep copy
- `Employee cloned = (Employee)e.clone();`

Cloning



Cloning

- `Object.clone` makes new object and copies all fields
- Cloning is subtle
- `Object.clone` is `protected`
- Subclass *must* redefine `clone` to be `public`

```
public class Employee
{
   public Object clone()
   {
      return super.clone(); // not complete
   }
   ...
}
```

The `Cloneable` Interface

- `Object.clone` is nervous about cloning
- Will only clone objects that implement `Cloneable` interface

```
public interface Cloneable
{
}
```

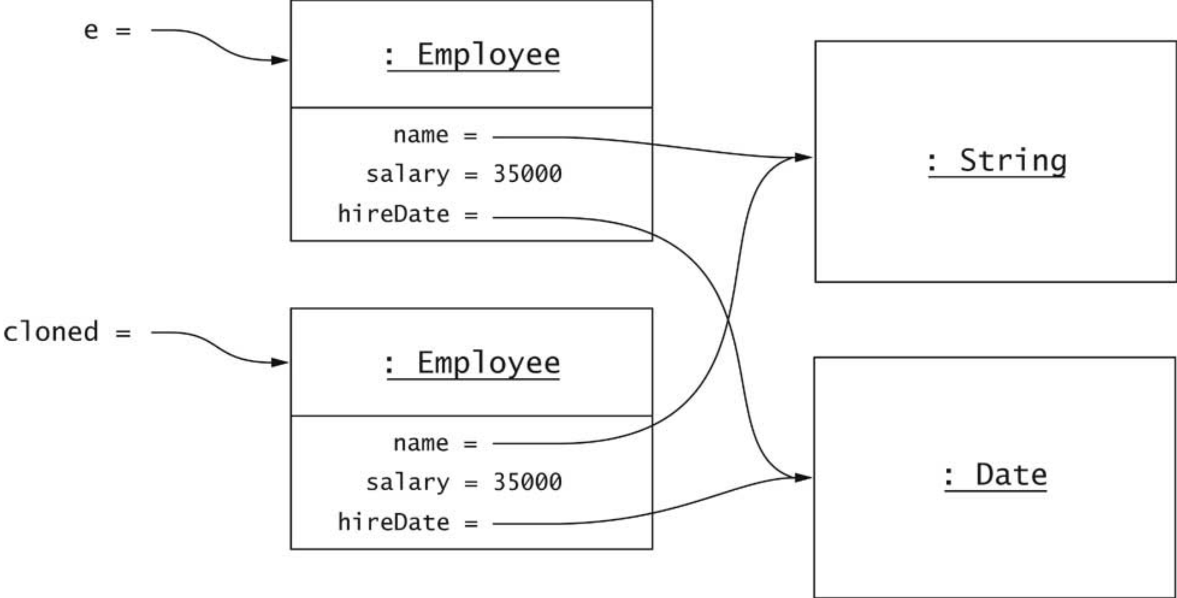
- Interface has no methods!
- Tagging interface--used in test
  `if x implements Cloneable`
- `Object.clone` throws `CloneNotSupportedException`
- A checked exception

The `clone` Method

```
public class Employee
  implements Cloneable
{
   public Object clone()
   {
      try
      {
         return super.clone();
      }
      catch(CloneNotSupportedException e)
      {
         return null; // won't happen
      }
   }
   ...
}
```

Shallow Cloning

- `clone` makes a shallow copy
- Instance fields aren't cloned

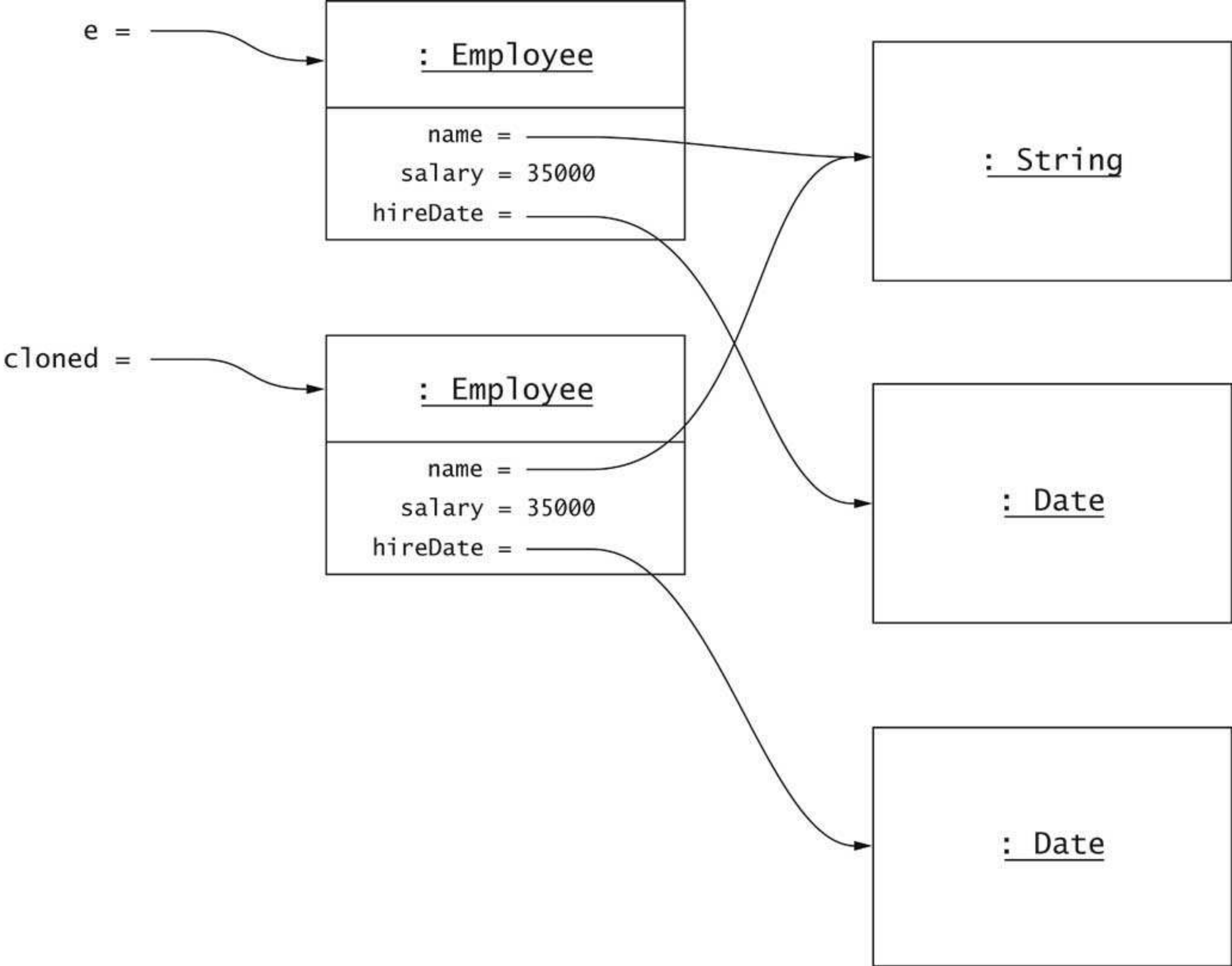


Deep Cloning

- Why doesn't `clone` make a deep copy?
  Wouldn't work for cyclic data structures
- Not a problem for immutable fields
- You must clone mutable fields

```
public class Employee
  implements Cloneable
{
   public Object clone()
   {
      try
      {
         Employee cloned = (Employee)super.clone();
         cloned.hireDate = (Date)hiredate.clone();
         return cloned;
      }
      catch(CloneNotSupportedException e)
      {
         return null; // won't happen
      }
   }
   ...
}
```

Deep Cloning



Cloning and Inheritance

- `Object.clone` is paranoid
  - `clone` is protected
  - clone only clones `Cloneable` objects
  - `clone` throws checked exception
- You don't have that luxury
- `Manager.clone` *must* be defined if `Manager` adds mutable fields
- Rule of thumb: if you extend a class that defines `clone`, redefine `clone`
- Lesson to learn: Tagging interfaces are inherited. Use them only to tag properties that inherit

Serialization

- Save collection of objects to stream

```
Employee[] staff = new Employee[2];
staff.add(new Employee(...));
staff.add(new Employee(...));
```

- Construct `ObjectOutputStream`:

```
ObjectOutputStream out
   = new ObjectOutputStream(
      new FileOutputStream("staff.dat"));
```

- Save the array and close the stream

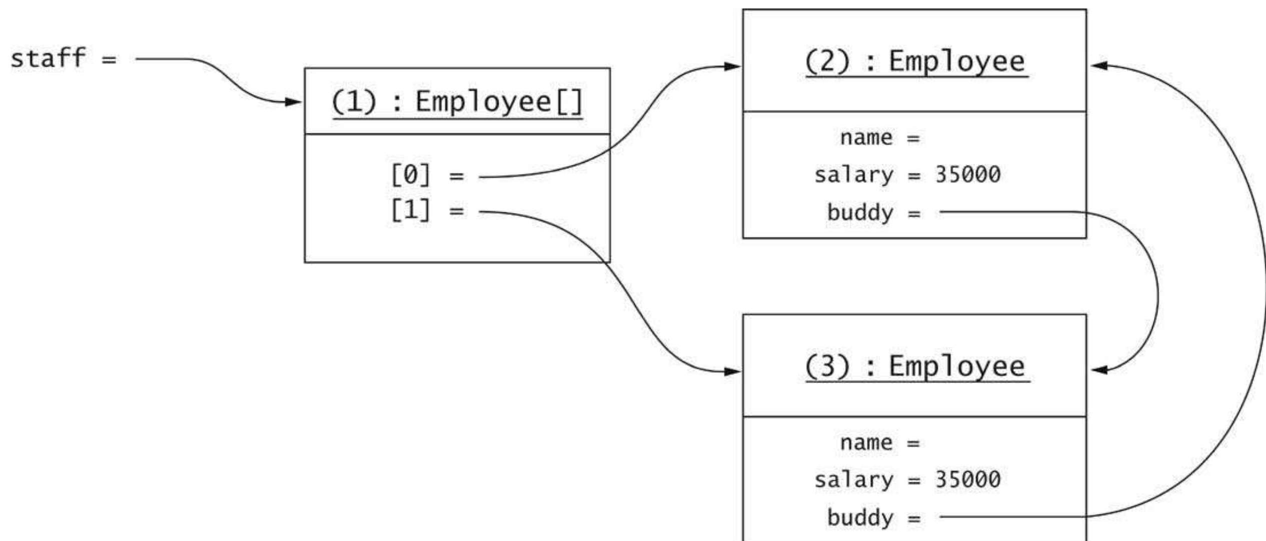
```
out.writeObject(staff);
out.close();
```

Serialization

- The array *and all of its objects and their dependent objects* are saved
- `Employee` doesn't have to define any method
- Needs to implement the `Serializable` interface
- Another tagging interface with no methods

How Serialization Works

- Each newly encountered object is saved
- Each object gets a serial number in the stream
- No object is saved twice
- Reference to already encountered object saved as "reference to #"



Serializing Unserializable Classes

- Some classes are not serializable
- Security? Anonymous classes? Programmer cluelessness?
- Example: `Ellipse2D.Double`
- How can we serialize `Car`?
- Suppress default serialization to avoid exception
- Mark with `transient`:
  `private transient Ellipse2D frontTire;`
- Supply `private` (!) methods
  `private void writeObject(ObjectOutputStream out)`
  `private void readObject(ObjectInputStream in)`
- In these methods
-
  - Call `writeDefaultObject/readDefaultObject`
  - Manually save other data
- Ch7/serial/Car.java

```
001: import java.awt.*;
002: import java.awt.geom.*;
003: import java.io.*;
004:
005: /**
006:    A serializable car shape.
007: */
008: public class Car implements Serializable
009: {
010:    /**
011:       Constructs a car.
012:       @param x the left of the bounding rectangle
013:       @param y the top of the bounding rectangle
014:       @param width the width of the bounding rectangle
015:    */
016:    public Car(int x, int y, int width)
017:    {
018:       body = new Rectangle(x, y + width / 6,
019:          width - 1, width / 6);
020:       roof = new Rectangle(x + width / 3, y,
021:          width / 3, width / 6);
022:       frontTire = new Ellipse2D.Double(x + width / 6, y + width / 3,
023:          width / 6, width / 6);
024:       rearTire = new Ellipse2D.Double(x + width * 2 / 3, y + width / 3,
025:          width / 6, width / 6);
026:    }
027:
028:    private void writeObject(ObjectOutputStream out)
029:       throws IOException
030:    {
031:       out.defaultWriteObject();
032:       writeRectangularShape(out, frontTire);
033:       writeRectangularShape(out, rearTire);
034:    }
035:
036:    /**
037:       A helper method to write a rectangular shape.
038:       @param out the stream onto which to write the shape
039:       @param s the shape to write
040:    */
041:    private static void writeRectangularShape(ObjectOutputStream out,
```

```
042:       RectangularShape s)
043:       throws IOException
044:    {
045:       out.writeDouble(s.getX());
046:       out.writeDouble(s.getY());
047:       out.writeDouble(s.getWidth());
048:       out.writeDouble(s.getHeight());
049:    }
050:
051:    private void readObject(ObjectInputStream in)
052:       throws IOException, ClassNotFoundException
053:    {
054:       in.defaultReadObject();
055:       frontTire = new Ellipse2D.Double();
056:       readRectangularShape(in, frontTire);
057:       rearTire = new Ellipse2D.Double();
058:       readRectangularShape(in, rearTire);
059:    }
060:
061:    /**
062:        A helper method to read a rectangular shape.
063:        @param in the stream from which to read the shape
064:        @param s the shape to read. The method sets the frame
065:        of this rectangular shape.
066:     */
067:    private static void readRectangularShape(ObjectInputStream in,
068:       RectangularShape s)
069:       throws IOException
070:    {
071:       double x = in.readDouble();
072:       double y = in.readDouble();
073:       double width = in.readDouble();
074:       double height = in.readDouble();
075:       s.setFrame(x, y, width, height);
076:    }
077:
078:    /**
079:        Draws the car.
080:        @param g2 the graphics context
081:     */
082:    public void draw(Graphics2D g2)
```

```
083:    {
084:       g2.draw(body);
085:       g2.draw(roof);
086:       g2.draw(frontTire);
087:       g2.draw(rearTire);
088:    }
089:
090:    public String toString()
091:    {
092:       return getClass().getName()
093:          + "[body=" + body
094:          + ",roof=" + roof
095:          + ",frontTire=" + formatRectangularShape(frontTire)
096:          + ",rearTire=" + formatRectangularShape(rearTire)
097:          + "]";
098:    }
099:
100:    /**
101:        A helper method to format a rectangular shape.
102:        @param s the shape to format
103:        @return a formatted representation of the given shape
104:     */
105:    private static String formatRectangularShape(RectangularShape s)
106:    {
107:       return s.getClass().getName()
108:          + "[x=" + s.getX()
109:          + ",y=" + s.getY()
110:          + ",width=" + s.getWidth()
111:          + ",height=" + s.getHeight()
112:          + "]";
113:    }
114:
115:
116:    private Rectangle body;
117:    private Rectangle roof;
118:    private transient Ellipse2D.Double frontTire;
119:    private transient Ellipse2D.Double rearTire;
120: }
121:
122:
```

Reflection

- Ability of running program to find out about its objects and classes
- `Class` object reveals
  - superclass
  - interfaces
  - package
  - names and types of fields
  - names, parameter types, return types of methods
  - parameter types of constructors

Reflection

- `Class getSuperclass()`
- `Class[] getInterfaces()`
- `Package getPackage()`
- `Field[] getDeclaredFields()`
- `Constructor[] getDeclaredConstructors()`
- `Method[] getDeclaredMethods()`

Example: Enumerating static fields `Math` class

```
Field[] fields = Math.class.getDeclaredFields();
for (int i = 0; i < fields.length; i++)
   if (Modifier.isStatic(fields[i].getModifiers()))
      System.out.println(fields[i].getName());
```

Enumerating Constructors

- Print the names and parameter types of all `Rectangle` constructors:

```
for (int i = 0; i < cons.length; i++)
{
   Class[] params = cons[i].getParameterTypes();
   System.out.print("Rectangle(");
   for (int j = 0; j < params.length; j++)
   {
      if (j > 0) System.out.print(", ");
      System.out.print(params[j].getName());
   }
   System.out.println(")");
}
```

- Yields

```
Rectangle()
Rectangle(java.awt.Rectangle)
Rectangle(int, int, int, int)
Rectangle(int, int)
Rectangle(java.awt.Point, java.awt.Dimension)
Rectangle(java.awt.Point)
Rectangle(java.awt.Dimension)
```

Getting A Single Method Descriptor

- Supply method name
- Supply array of parameter types
- Example: Get `Rectangle.contains(int, int)`:

```
Method m = Rectangle.class.getDeclaredMethod(
   "contains",
   new Class[] { int.class, int.class });
```

- Example: Get default `Rectangle` constructor:

```
Constructor c = Rectangle.class.getDeclaredConstructor(
   new Class[] {});
```

Invoking a Method

- Supply implicit parameter (`null` for static methods)
- Supply array of explicit parameter values
- Wrap primitive types
- Unwrap primitive return value
- Example: Call `System.out.println("Hello, World")` the hard way.

```
Method m = PrintStream.class.getDeclaredMethod(
   "println",
   new Class[] { String.class } );
m.invoke(System.out,
   new Object[] { "Hello, World!" });
```

Inspecting Objects

- Can obtain object contents at runtime
- Useful for generic debugging tools
- Need to gain access to private fields

```
Class c = obj.getClass();
Field f = c.getDeclaredField(name);
f.setAccessible(true);
```

- Throws exception if security manager disallows access
- Access field value:

```
Object value = f.get(obj);
f.set(obj, value);
```

- Use wrappers for primitive types

Inspecting Objects

- Example: Peek inside string tokenizer
- Ch7/code/reflect2/FieldTest.java
- Output

```
int currentPosition=0
int newPosition=-1
int maxPosition=13
java.lang.String str=Hello, World!
java.lang.String delimiters=,
boolean retDelims=false
boolean delimsChanged=false
char maxDelimChar=,
---
int currentPosition=5
. . .
```

```
01: import java.lang.reflect.*;
02: import java.util.*;
03:
04: /**
05:    This program shows how to use reflection to print
06:    the names and values of all fields of an object.
07: */
08: public class FieldTest
09: {
10:    public static void main(String[] args)
11:       throws IllegalAccessException
12:    {
13:       String input = "Hello, World!";
14:       StringTokenizer tokenizer = new StringTokenizer(input, ",");
15:       System.out.print(spyFields(tokenizer));
16:       tokenizer.nextToken();
17:       System.out.println("\nAfter calling nextToken:\n");
18:       System.out.print(spyFields(tokenizer));
19:    }
20:
21:    /**
22:       Spies on the field names and values of an object.
23:       @param obj the object whose fields to format
24:       @return a string containing the names and values of
25:       all fields of obj
26:    */
27:    public static String spyFields(Object obj)
28:       throws IllegalAccessException
29:    {
30:       StringBuffer buffer = new StringBuffer();
31:       Field[] fields = obj.getClass().getDeclaredFields();
32:       for (int i = 0; i < fields.length; i++)
33:       {
34:          Field f = fields[i];
35:          f.setAccessible(true);
36:          Object value = f.get(obj);
37:          buffer.append(f.getType().getName());
38:          buffer.append(" ");
39:          buffer.append(f.getName());
40:          buffer.append("=");
41:          buffer.append("" + value);
```

```
42:             buffer.append("\n");
43:         }
44:         return buffer.toString();
45:     }
46: }
```

Inspecting Array Elements

- Use static methods of `Array` class

```
Object value = Array.get(a, i);
Array.set(a, i, value);

int n = Array.getLength(a);
```

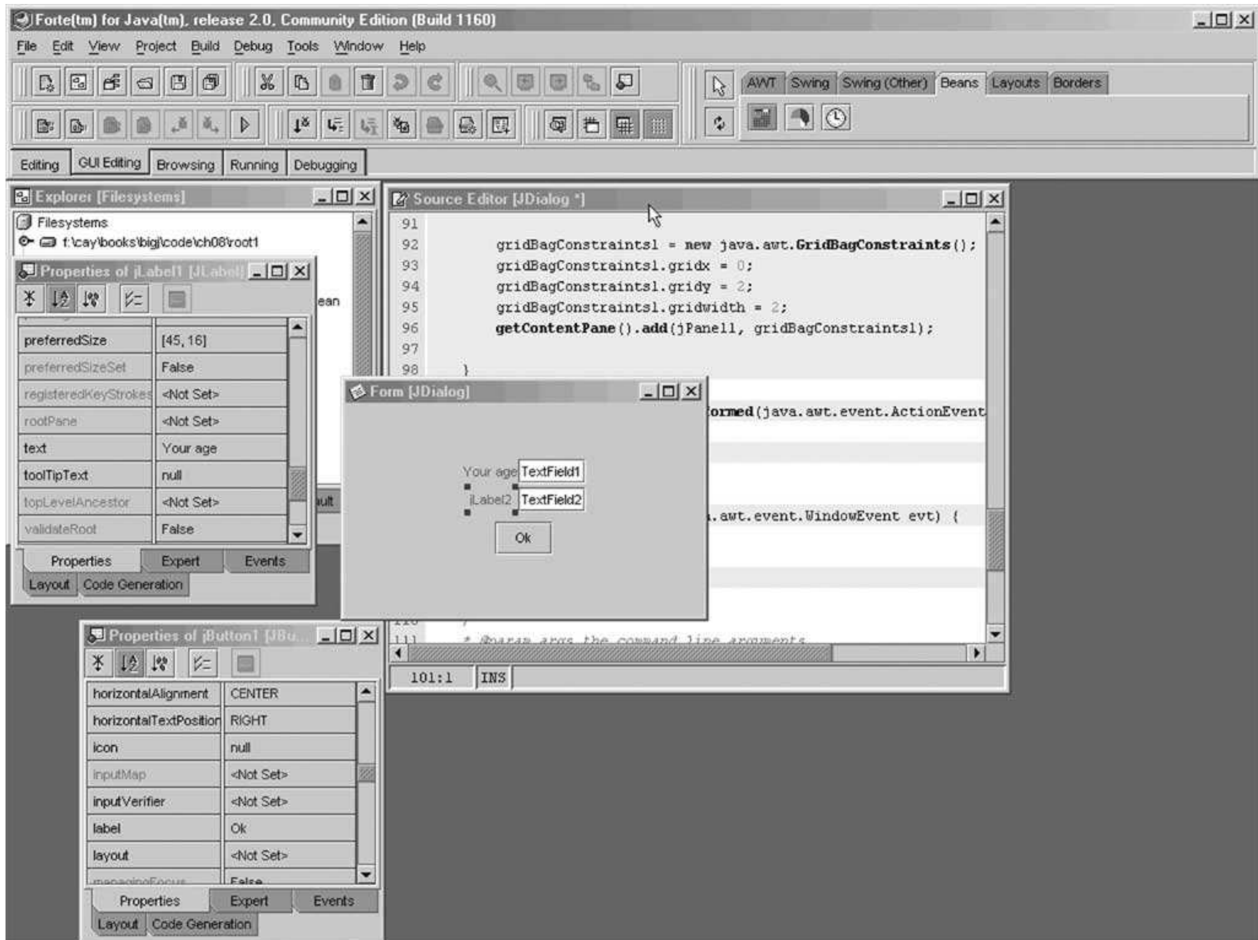
- Construct new array:

```
Object a = Array.newInstance(type, length);
```

Components

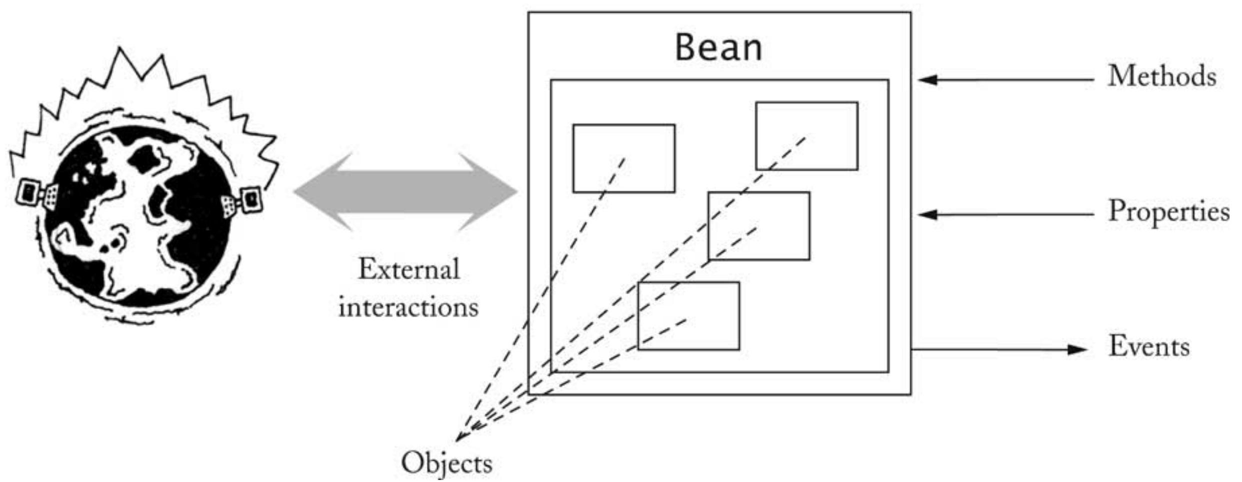
- More functionality than a single class
- Reuse and customize in multiple contexts
- "Plug components together" to form applications
- Successful model: Visual Basic controls
- Examples:
  - calendar
  - graph
  - database
  - link to robot or instrument
- Components composed into program inside builder environment

A Builder Environment



Java Beans

- Java component model
- Bean has:
  - methods (just like classes)
  - properties
  - events



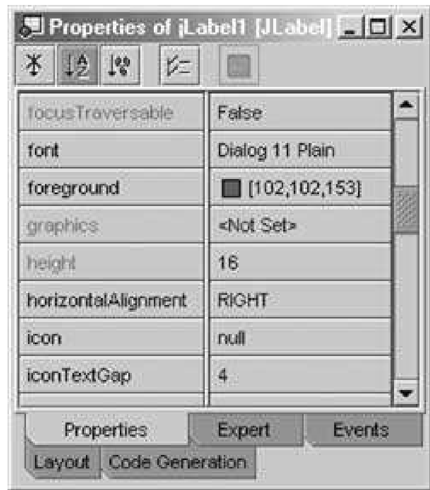
A Calendar Bean

February 2002

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28		

A Property Sheet

- Edit properties with *property sheet*



Facade Class

- Bean usually composed of multiple classes
- One class nominated as *facade class*
- Clients use only facade class methods

Facade Pattern

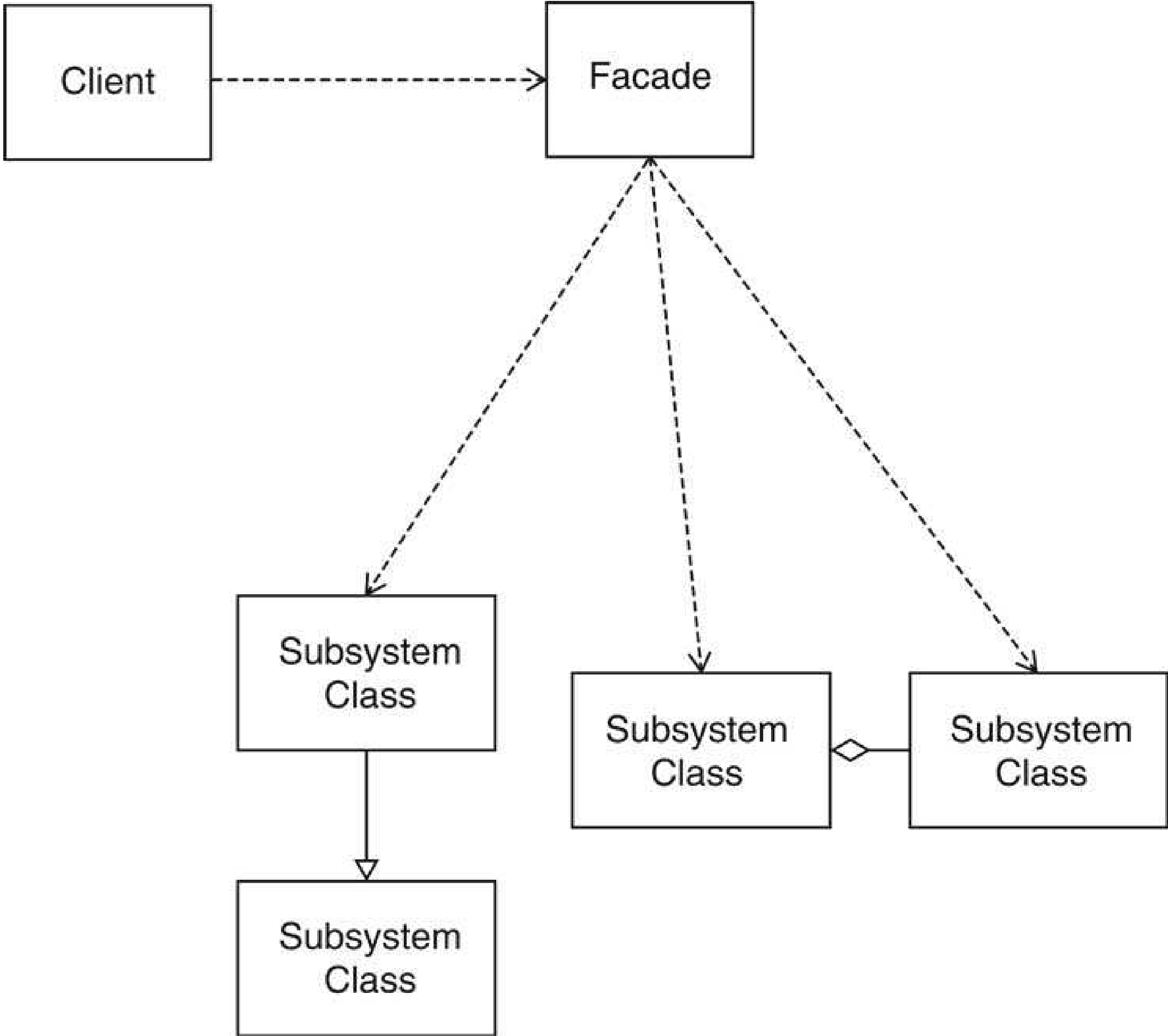
Context

1. A subsystem consists of multiple classes, making it complicated for clients to use
2. Implementor may want to change subsystem classes
3. Want to give a coherent entry point

Solution

1. Define a facade class that exposes all capabilities of the subsystem as methods
2. The facade methods delegate requests to the subsystem classes
3. The subsystem classes do not know about the facade class

Facade Pattern



Facade Pattern

Name in Design Pattern	Actual Name (Beans)
Client	Builder tool
Facade	Main bean class with which the tool interacts
SubsystemClass	Class used to implement bean functionality

Bean Properties

- Property = value that you can get and/or set
- Most properties are get-and-set
- Can also have get-only and set-only
- Property *not the same as* instance field
- Setter can set fields, then call repaint
- Getter can query database

Property Syntax

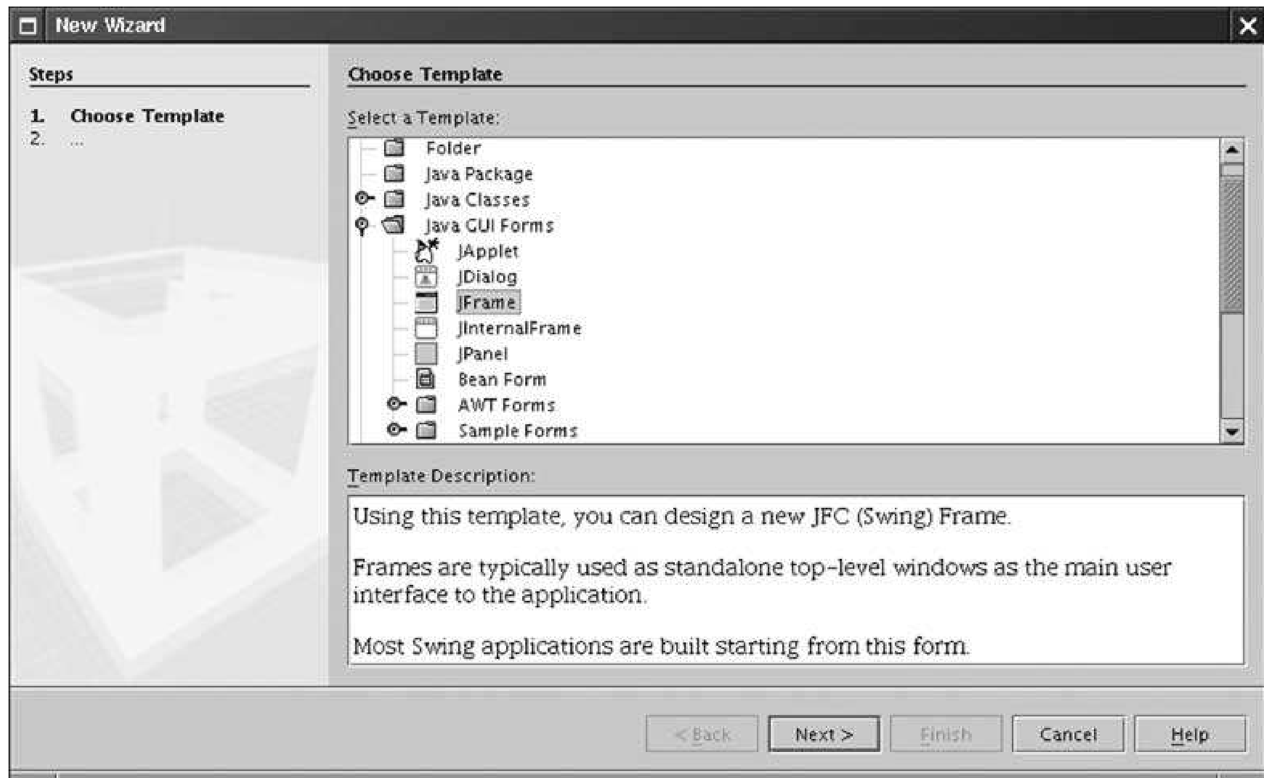
- Not Java :-(
- C#, JavaScript, Visual Basic
- `b.propertyName = value`
  calls setter
- `variable = b.propertyName`
  calls getter

Java Naming Conventions

- property = pair of methods
  `public X get`*`PropertyName`*`()`
  `public void set`*`PropertyName`*`(X newValue)`
- Replace *`propertyName`* with actual name
  (e.g. `getColor`/`setColor`)
- Exception for `boolean` properties:
  `public boolean is`*`PropertyName`*`()`
- Decapitalization hokus-pokus:
  `getColor -> color`
  `getURL -> URL`

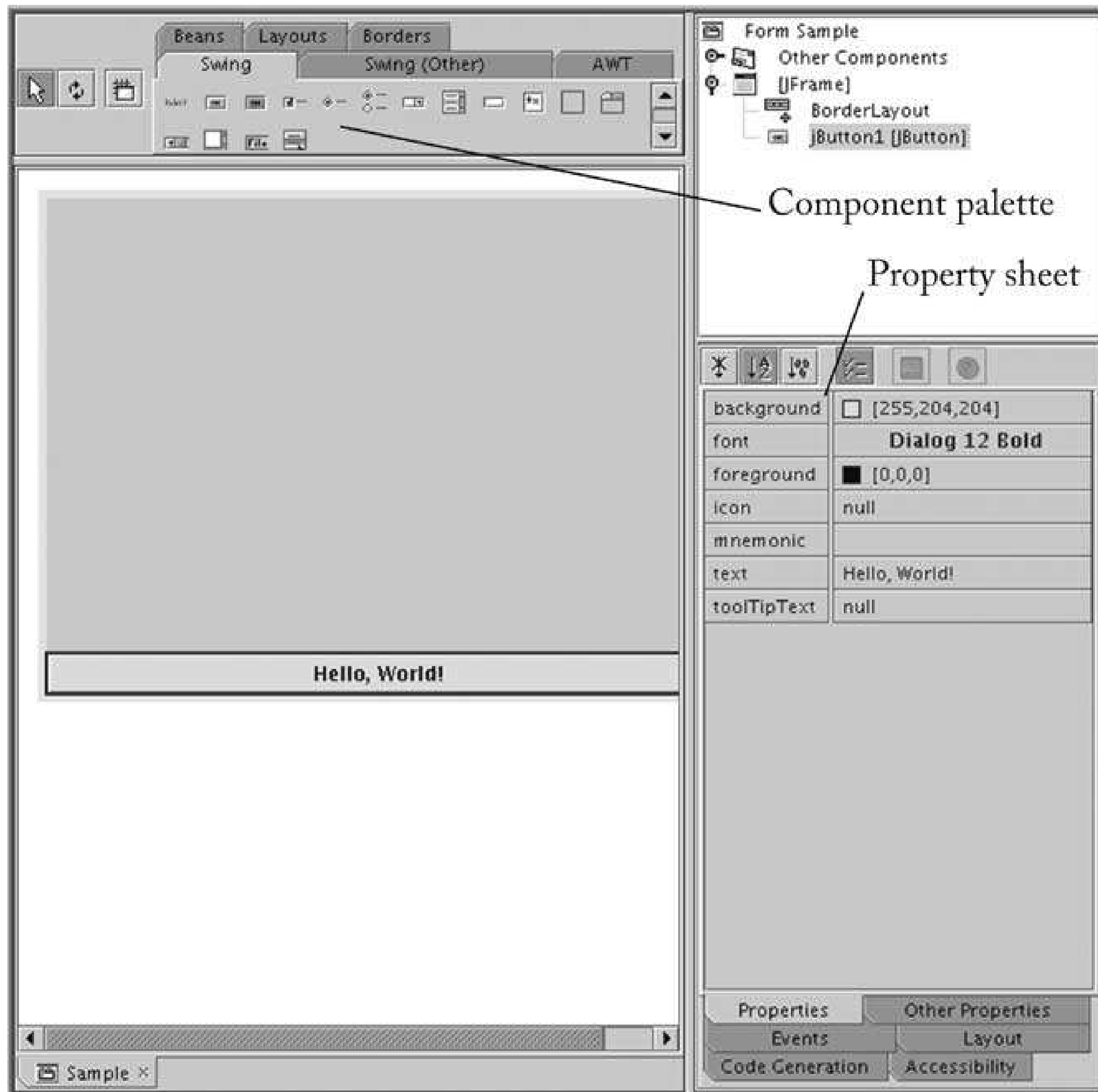
Editing Beans in a Builder Tool

- Use wizard to make empty frame



Editing Beans in a Builder Tool

- Add button to frame, then edit button with property sheet.



Packaging a Bean

- Compile bean classes
  Ch7/carbean1/CarBean.java
- Create manifest file
  Ch7/carbean1/CarBean.mf
- Run JAR tool:
  `jar cvfm CarBean.jar CarBean.mf *.class`
- Import JAR file into builder environment

```
001: import java.awt.*;
002: import java.awt.geom.*;
003: import javax.swing.*;
004:
005: /**
006:    A panel that draws a car shape.
007: */
008: public class CarBean extends JPanel
009: {
010:    /**
011:       Constructs a default car bean.
012:    */
013:    public CarBean()
014:    {
015:       x = 0;
016:       y = 0;
017:       width = DEFAULT_CAR_WIDTH;
018:       height = DEFAULT_CAR_HEIGHT;
019:    }
020:
021:    /**
022:       Sets the x property.
023:       @param newValue the new x position
024:    */
025:    public void setX(int newValue)
026:    {
027:       x = newValue;
028:       repaint();
029:    }
030:
031:    /**
032:       Gets the x property.
033:       @return the x position
034:    */
035:    public int getX()
036:    {
037:       return x;
038:    }
039:
040:    /**
041:       Sets the y property.
```

```
042:        @param newValue the new y position
043:     */
044:    public void setY(int newValue)
045:    {
046:       y = newValue;
047:       repaint();
048:    }
049:
050:    /**
051:        Gets the y property.
052:        @return the y position
053:     */
054:    public int getY()
055:    {
056:       return y;
057:    }
058:
059:    public void paintComponent(Graphics g)
060:    {
061:       super.paintComponent(g);
062:       Graphics2D g2 = (Graphics2D) g;
063:       Rectangle2D.Double body
064:          = new Rectangle2D.Double(x, y + height / 3,
065:             width - 1, height / 3);
066:       Ellipse2D.Double frontTire
067:          = new Ellipse2D.Double(x + width / 6,
068:             y + height * 2 / 3, height / 3, height / 3);
069:       Ellipse2D.Double rearTire
070:          = new Ellipse2D.Double(x + width * 2 / 3,
071:             y + height * 2 / 3, height / 3, height / 3);
072:
073:       // the bottom of the front windshield
074:       Point2D.Double r1
075:          = new Point2D.Double(x + width / 6, y + height / 3);
076:       // the front of the roof
077:       Point2D.Double r2
078:          = new Point2D.Double(x + width / 3, y);
079:       // the rear of the roof
080:       Point2D.Double r3
081:          = new Point2D.Double(x + width * 2 / 3, y);
082:       // the bottom of the rear windshield
```

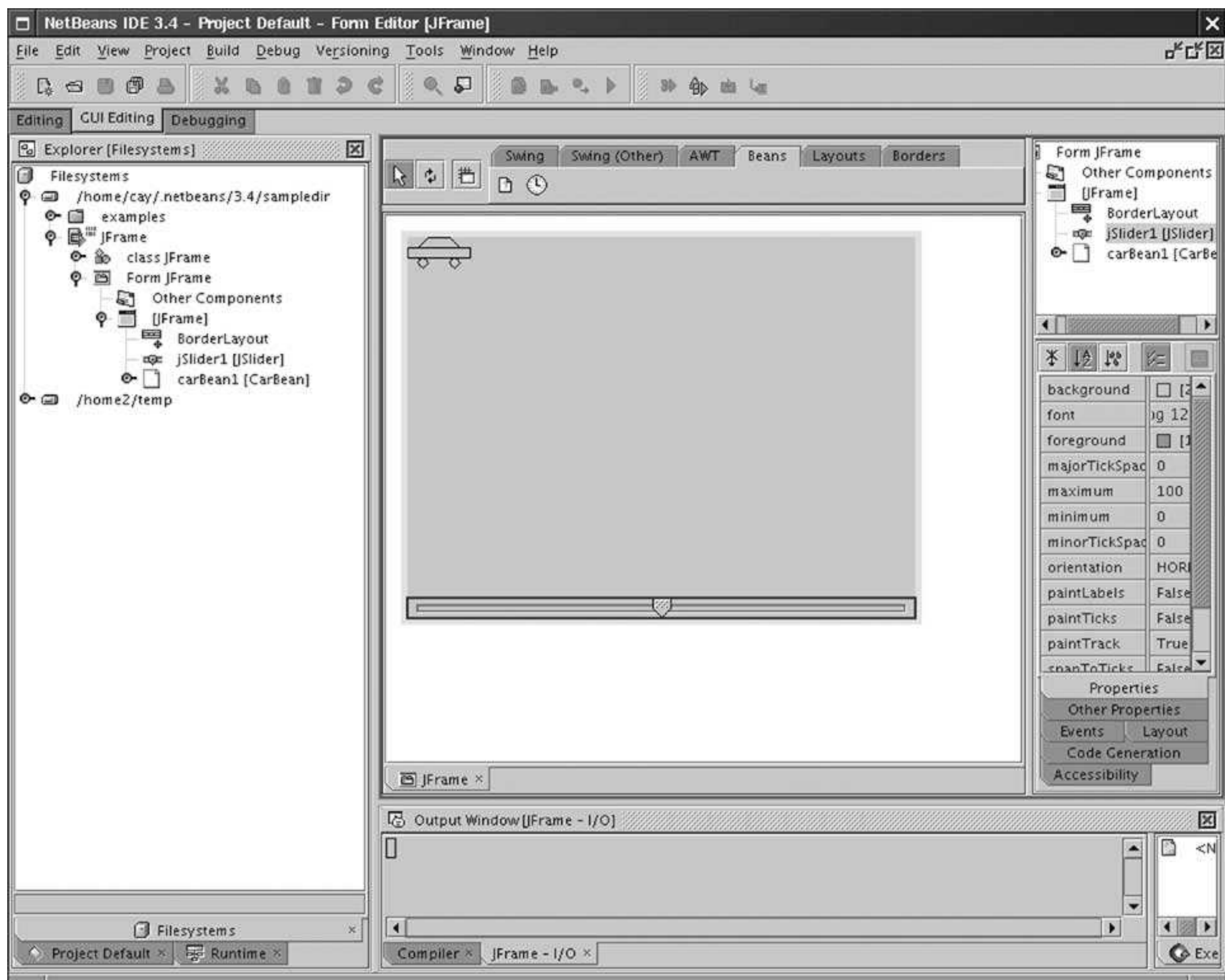
```
083:       Point2D.Double r4
084:          = new Point2D.Double(x + width * 5 / 6, y + height / 3);
085:
086:       Line2D.Double frontWindshield
087:          = new Line2D.Double(r1, r2);
088:       Line2D.Double roofTop
089:          = new Line2D.Double(r2, r3);
090:       Line2D.Double rearWindshield
091:          = new Line2D.Double(r3, r4);
092:
093:       g2.draw(body);
094:       g2.draw(frontTire);
095:       g2.draw(rearTire);
096:       g2.draw(frontWindshield);
097:       g2.draw(roofTop);
098:       g2.draw(rearWindshield);
099:    }
100:
101:    public Dimension getPreferredSize()
102:    {
103:       return new Dimension(DEFAULT_PANEL_WIDTH,
104:          DEFAULT_PANEL_HEIGHT);
105:    }
106:
107:    private int x;
108:    private int y;
109:    private int width;
110:    private int height;
111:
112:    private static final int DEFAULT_CAR_WIDTH = 60;
113:    private static final int DEFAULT_CAR_HEIGHT = 30;
114:    private static final int DEFAULT_PANEL_WIDTH = 160;
115:    private static final int DEFAULT_PANEL_HEIGHT = 130;
116: }
```

Composing Beans

- Make new frame
- Add car bean, slider to frame
- Edit `stateChanged` event of slider
- Add handler code

```
carBean1.setX(jSlider1.getValue());
```

- Compile and run
- Move slider: the car moves


NetBeans IDE 3.4 - Project Default - Form Editor [JFrame]
File Edit View Project Build Debug Versioning Tools Window Help
Editing GUI Editing Debugging
Explorer [Filesystems]
Filesystems
/home/cay/.netbeans/3.4/sampledir
examples
JFrame
class JFrame
Form JFrame
Other Components
[JFrame]
BorderLayout
jSlider1 [JSlider]
carBean1 [CarBean]
/home2/temp
Filesystems
Project Default
Runtime
Swing Swing (Other) AWT Beans Layouts Borders
JFrame
background
font
foreground
majorTickSpac 0
maximum 100
minimum 0
minorTickSpac 0
orientation
paintLabels False
paintTicks False
paintTrack True
Properties
Other Properties
Events Layout
Code Generation
Accessibility
Output Window [JFrame - I/O]
Compiler
JFrame - I/O

Bean Information

- Builder environment loads beans
- Looks for `get/set` methods in facade class
- Can discover spurious properties
  `JButton: Object getTreeLock()`
- Alternate mechanism: BeanInfo class
- Must have name *`FacadeClassName`*`BeanInfo`
- E.g. `HouseBeanBeanInfo`

The BeanInfo Interface

```
Image getIcon(int iconKind)
BeanDescriptor getBeanDescriptor()
MethodDescriptor[] getMethodDescriptors()
PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
int getDefaultEventIndex()
int getDefaultPropertyIndex()
BeanInfo[] getAdditionalBeanInfo()
```

Removing Spurious Properties

```
class MyBeanBeanInfo extends SimpleBeanInfo
{
   public PropertyDescriptor[] getPropertyDescriptors()
   {
      try
      {
         return new PropertyDescriptor[]
         {
            new PropertyDescriptor("x", CarBean.class);
            new PropertyDescriptor("y", CarBean.class);
         };
      }
      catch (IntrospectionException exception)
      { return null; }
   }
}
```

Property Editors

- Property sheet enumerates properties
- Allows user to edit property values
- How can one edit values of arbitrary types?
- Built-in editors for `String`, `Color`, etc
- Supply custom editor for your own types

Custom Property Editors

- Three kinds of editors
  - Text
  - Finite set of choices
  - Arbitrary painting and editing
- Implement `PropertyEditor` interface
- Or extend `PropertyEditorSupport` class

Editing Text Properties

- Convert between your type and `String`
- Define two methods

```
public String getAsText()
public void setAsText(String s)
```

- Property sheet uses text field

Editing Choice Properties

- Your type has finite set of string choices
- E.g. `DrawMode.DRAW`, `DrawMode.FILL`
- `String[] getTags()` returns array of choices
- Also need to define `getAsText/setAsText`
- Property sheet uses combo box

Editing Arbitrary Properties

- Your type isn't easily editable as string
- E.g. `Color`
- Property editor pops up your edit dialog
  - `boolean supportsCustomEditor()` must return `true`
  - `Component getCustomEditor()` returns dialog
- Property editor can paint current value of your type
  - `boolean isPaintable()` must return `true`
  - `void paintValue(Graphics g, Rectangle bounds)` paints

Registering Property Editors

- Global setting

```
PropertyEditorManager.registerEditor(valueClass, editorClass)
```

- Per-bean setting
  In bean info class:

```
PropertyDescriptor dimensionProperty = new PropertyDescriptor(
   "dimension", CarBean.class);
dimensionProperty.setPropertyEditorClass(DimensionEditor.class);
```

Example: CarBean

- Ch7/carbean2/CarBean.java
- Ch7/carbean2/CarBeanBeanInfo.java
- Ch7/carbean2/DimensionEditor.java
- Ch7/carbean2/DrawMode.java
- Ch7/carbean2/DrawModeEditor.java
- Ch7/carbean2/CustomColorEditor.java

```
001: import java.awt.*;
002: import java.awt.geom.*;
003: import javax.swing.*;
004:
005: /**
006:    A panel that draws a car shape.
007: */
008: public class CarBean extends JPanel
009: {
010:    /**
011:       Constructs a default car bean.
012:    */
013:    public CarBean()
014:    {
015:       x = 0;
016:       y = 0;
017:       width = DEFAULT_WIDTH;
018:       height = DEFAULT_HEIGHT;
019:       color = DEFAULT_COLOR;
020:       fill = false;
021:    }
022:
023:    /**
024:       Sets the color property.
025:       @param color the new color
026:    */
027:    public void setColor(Color color)
028:    {
029:       this.color = color;
030:       repaint();
031:    }
032:
033:    /**
034:       Gets the color property.
035:       @return the current color
036:    */
037:    public Color getColor() { return color; }
038:
039:    /**
040:       Sets the dimension property.
041:       @param dimension the new dimension of the house
```

```
042:     */
043:    public void setDimension(Dimension dimension)
044:    {
045:       width = (int) dimension.getWidth();
046:       height = (int) dimension.getHeight();
047:       repaint();
048:    }
049:
050:    /**
051:       Gets the dimension property.
052:       @return the current dimension of the house
053:     */
054:    public Dimension getDimension()
055:    {
056:       return new Dimension(width, height);
057:    }
058:
059:    /**
060:       Sets the drawMode property.
061:       @param drawMode the new drawMode (DRAW or FILL)
062:     */
063:    public void setDrawMode(DrawMode drawMode)
064:    {
065:       if (drawMode == DrawMode.DRAW)
066:       {
067:          fill = false;
068:          repaint();
069:       }
070:       else if (drawMode == DrawMode.FILL)
071:       {
072:          fill = true;
073:          repaint();
074:       }
075:    }
076:
077:    /**
078:       Gets the drawMode property.
079:       @return the current drawMode (DRAW or FILL)
080:     */
081:    public DrawMode getDrawMode()
082:    {
```

```
083:        if (fill) return DrawMode.FILL;
084:        else return DrawMode.DRAW;
085:     }
086:
087:     public void paintComponent(Graphics g)
088:     {
089:        super.paintComponent(g);
090:        Graphics2D g2 = (Graphics2D) g;
091:        Rectangle2D.Double body
092:           = new Rectangle2D.Double(x, y + height / 3,
093:              width - 1, height / 3);
094:        Ellipse2D.Double frontTire
095:           = new Ellipse2D.Double(x + width / 6,
096:              y + height * 2 / 3, height / 3, height / 3);
097:        Ellipse2D.Double rearTire
098:           = new Ellipse2D.Double(x + width * 2 / 3,
099:              y + height * 2 / 3, height / 3, height / 3);
100:
101:        // the bottom of the front windshield
102:        Point2D.Double r1
103:           = new Point2D.Double(x + width / 6, y + height / 3);
104:        // the front of the roof
105:        Point2D.Double r2
106:           = new Point2D.Double(x + width / 3, y);
107:        // the rear of the roof
108:        Point2D.Double r3
109:           = new Point2D.Double(x + width * 2 / 3, y);
110:        // the bottom of the rear windshield
111:        Point2D.Double r4
112:           = new Point2D.Double(x + width * 5 / 6, y + height / 3);
113:        Line2D.Double frontWindshield
114:           = new Line2D.Double(r1, r2);
115:        Line2D.Double roofTop
116:           = new Line2D.Double(r2, r3);
117:        Line2D.Double rearWindshield
118:           = new Line2D.Double(r3, r4);
119:
120:        g2.setColor(color);
121:        if (fill)
122:        {
123:           g2.fill(body);
```

```
124:         g2.fill(frontTire);
125:         g2.fill(rearTire);
126:      }
127:      else
128:      {
129:         g2.draw(body);
130:         g2.draw(frontTire);
131:         g2.draw(rearTire);
132:      }
133:      g2.draw(frontWindshield);
134:      g2.draw(roofTop);
135:      g2.draw(rearWindshield);
136:   }
137:
138:   public static final int DRAW = 0;
139:   public static final int FILL = 1;
140:
141:   private int x;
142:   private int y;
143:   private Color color;
144:   private int width;
145:   private int height;
146:   private boolean fill;
147:
148:   private static final int DEFAULT_WIDTH = 50;
149:   private static final int DEFAULT_HEIGHT = 80;
150:   private static final Color DEFAULT_COLOR = Color.BLACK;
151: }
```

```
01: import java.beans.*;
02:
03: /**
04:    The bean info for the car bean, specifying the properties
05:    and their editors.
06: */
07: public class CarBeanBeanInfo extends SimpleBeanInfo
08: {
09:    public PropertyDescriptor[] getPropertyDescriptors()
10:    {
11:       try
12:       {
13:          PropertyDescriptor colorProperty
14:             = new PropertyDescriptor("color",
15:                CarBean.class);
16:          colorProperty.setPropertyEditorClass(
17:             CustomColorEditor.class);
18:
19:          PropertyDescriptor dimensionProperty
20:             = new PropertyDescriptor("dimension",
21:                CarBean.class);
22:          dimensionProperty.setPropertyEditorClass(
23:             DimensionEditor.class);
24:
25:          PropertyDescriptor drawModeProperty
26:             = new PropertyDescriptor("drawMode",
27:                CarBean.class);
28:          drawModeProperty.setPropertyEditorClass(
29:             DrawModeEditor.class);
30:
31:          return new PropertyDescriptor[]
32:             {
33:                colorProperty,
34:                dimensionProperty,
35:                drawModeProperty
36:             };
37:       }
38:       catch (IntrospectionException exception)
39:       {
```

```
40:             return null;
41:         }
42:     }
43: }
```

```
01: import java.awt.*;
02: import java.beans.*;
03: import java.util.*;
04:
05: /**
06:    A property editor for the Dimension type that presents
07:    a dimension as a text "width x height".
08: */
09: public class DimensionEditor extends PropertyEditorSupport
10: {
11:    public String getAsText()
12:    {
13:       Dimension value = (Dimension) getValue();
14:       return (int) value.getWidth()
15:          + "x" + (int) value.getHeight();
16:    }
17:
18:    public void setAsText(String s)
19:    {
20:       try
21:       {
22:          StringTokenizer tokenizer
23:             = new StringTokenizer(s, "x");
24:          if (!tokenizer.hasMoreTokens())
25:             throw new IllegalArgumentException();
26:          int width
27:             = Integer.parseInt(tokenizer.nextToken().trim());
28:          if (!tokenizer.hasMoreTokens())
29:             throw new IllegalArgumentException();
30:          int height
31:             = Integer.parseInt(tokenizer.nextToken().trim());
32:          setValue(new Dimension(width, height));
33:       }
34:       catch (NumberFormatException exception)
35:       {
36:          throw new IllegalArgumentException();
37:       }
38:    }
39: }
```

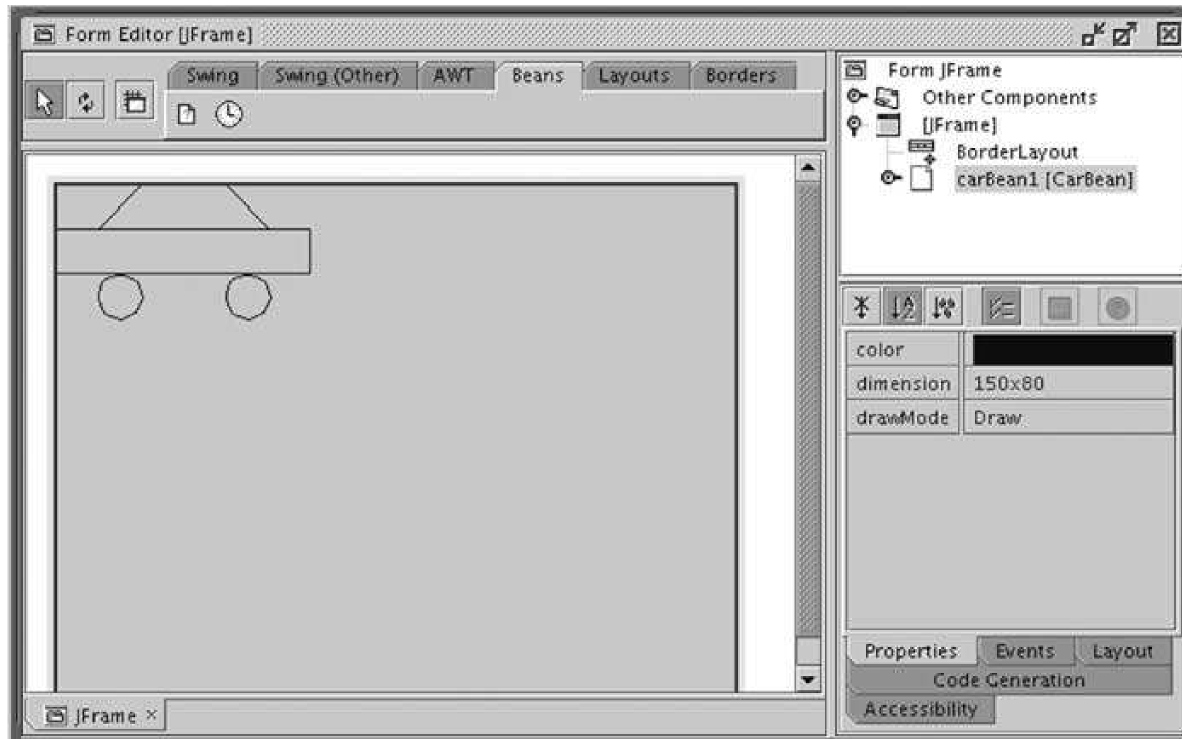
```
01: import java.io.*;
02:
03: public class DrawMode implements Serializable
04: {
05:    private DrawMode(String name) { this.name = name; }
06:    private String name;
07:
08:    protected Object readResolve()
09:    {
10:       if (name.equals("DRAW")) return DrawMode.DRAW;
11:       else if (name.equals("FILL")) return DrawMode.FILL;
12:       else return null;
13:    }
14:
15:
16:    public static final DrawMode DRAW = new DrawMode("DRAW");
17:    public static final DrawMode FILL = new DrawMode("FILL");
18: }
```

```
01: import java.beans.*;
02:
03: /**
04:    A property editor for the draw mode of the CarBean.
05: */
06: public class DrawModeEditor extends PropertyEditorSupport
07: {
08:    public String[] getTags()
09:    {
10:       return new String[] { "Draw", "Fill" };
11:    }
12:
13:    public String getAsText()
14:    {
15:       DrawMode value = (DrawMode) getValue();
16:       if (value == DrawMode.DRAW) return "Draw";
17:       if (value == DrawMode.FILL) return "Fill";
18:       return null;
19:    }
20:
21:    public void setAsText(String s)
22:    {
23:       if (s.equals("Draw")) setValue(DrawMode.DRAW);
24:       else if (s.equals("Fill")) setValue(DrawMode.FILL);
25:       else throw new IllegalArgumentException();
26:    }
27: }
```

```
01: import java.awt.*;
02: import java.beans.*;
03: import javax.swing.*;
04: import javax.swing.event.*;
05:
06: /**
07:    A property editor for the Color type that uses a JColorChooser.
08: */
09: public class CustomColorEditor extends PropertyEditorSupport
10: {
11:    public String getAsText()
12:    {
13:       return null;
14:    }
15:
16:    public boolean supportsCustomEditor()
17:    {
18:       return true;
19:    }
20:
21:    public Component getCustomEditor()
22:    {
23:       final JColorChooser chooser = new JColorChooser();
24:       chooser.getSelectionModel().addChangeListener(
25:          new ChangeListener()
26:          {
27:             public void stateChanged(ChangeEvent event)
28:             {
29:                setValue(chooser.getColor());
30:             }
31:          });
32:       return chooser;
33:    }
34:
35:    public boolean isPaintable()
36:    {
37:       return true;
38:    }
39:
40:    public void paintValue(Graphics g, Rectangle boundingBox)
41:    {
```

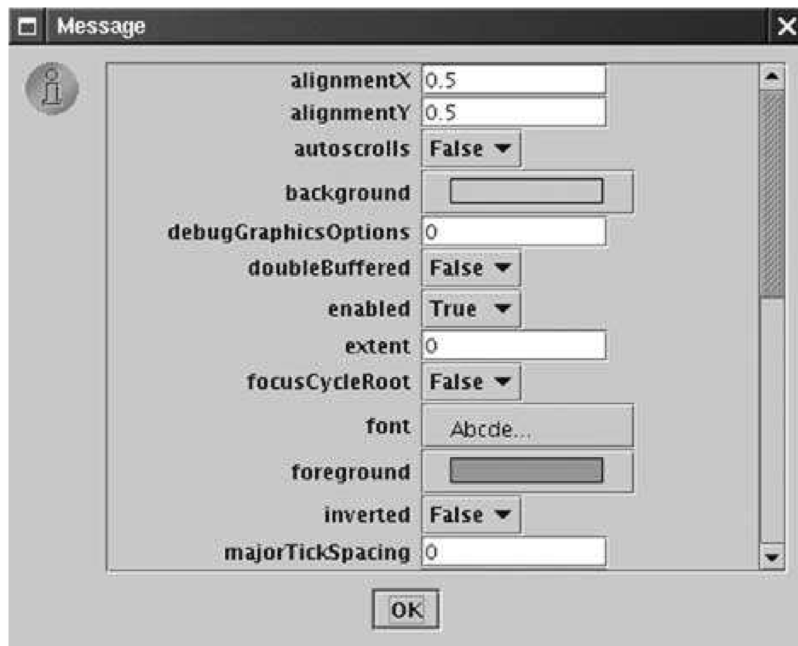
```
42:         Graphics2D g2 = (Graphics2D) g;
43:         Color color = (Color) getValue();
44:         g2.setColor(color);
45:         g2.fill(boundingBox);
46:         g2.setColor(Color.BLACK);
47:         g2.draw(boundingBox);
48:     }
49: }
```

Example: CarBean



Implementing a Property Sheet

- Used for graph framework in chapter 8
- Form shows property names on left, editors on right



Implementing a Property Sheet

- Get bean info for class
- Get properties from bean info
- Obtain property getter/setter methods from property descriptor
- Use these methods to read and write property values
- Each editor is text field, combo box, or button with painted icon
- Clicking on button brings up dialog
- Ch7/propedit/PropertySheet.java
- Ch7/propedit/PropertySheetTest.java

```
001: import java.awt.*;
002: import java.awt.event.*;
003: import java.beans.*;
004: import java.lang.reflect.*;
005: import java.util.*;
006: import javax.swing.*;
007: import javax.swing.event.*;
008:
009: /**
010:    A component filled with editors for all editable properties
011:    of an object.
012: */
013: public class PropertySheet extends JPanel
014: {
015:    /**
016:       Constructs a property sheet that shows the editable
017:       properties of a given object.
018:       @param object the object whose properties are being edited
019:    */
020:    public PropertySheet(Object bean)
021:    {
022:       try
023:       {
024:          BeanInfo info
025:             = Introspector.getBeanInfo(bean.getClass());
026:          PropertyDescriptor[] descriptors
027:             = info.getPropertyDescriptors();
028:          setLayout(new FormLayout());
029:          for (int i = 0; i < descriptors.length; i++)
030:          {
031:             PropertyEditor editor
032:                = getEditor(bean, descriptors[i]);
033:             if (editor != null)
034:             {
035:                add(new JLabel(descriptors[i].getName()));
036:                add(getEditorComponent(editor));
037:             }
038:          }
039:       }
040:       catch (IntrospectionException exception)
041:       {
```

```
042:          exception.printStackTrace();
043:       }
044:    }
045:
046:    /**
047:       Gets the property editor for a given property,
048:       and wires it so that it updates the given object.
049:       @param bean the object whose properties are being edited
050:       @param descriptor the descriptor of the property to
051:       be edited
052:       @return a property editor that edits the property
053:       with the given descriptor and updates the given object
054:    */
055:    public PropertyEditor getEditor(final Object bean,
056:       PropertyDescriptor descriptor)
057:    {
058:       try
059:       {
060:          Method getter = descriptor.getReadMethod();
061:          if (getter == null) return null;
062:          final Method setter = descriptor.getWriteMethod();
063:          if (setter == null) return null;
064:          final PropertyEditor editor;
065:          Class editorClass = descriptor.getPropertyEditorClass();
066:          if (editorClass != null)
067:             editor = (PropertyEditor) editorClass.newInstance();
068:          else
069:             editor = PropertyEditorManager.findEditor(
070:                descriptor.getPropertyType());
071:          if (editor == null) return null;
072:
073:          Object value = getter.invoke(bean, new Object[] {});
074:          editor.setValue(value);
075:          editor.addPropertyChangeListener(new
076:             PropertyChangeListener()
077:             {
078:                public void propertyChange(PropertyChangeEvent event)
079:                {
080:                   try
081:                   {
082:                      setter.invoke(bean,
```

```
083:                         new Object[] { editor.getValue() });
084:                   }
085:                   catch (IllegalAccessException exception)
086:                   {
087:                   }
088:                   catch (InvocationTargetException exception)
089:                   {
090:                   }
091:                }
092:             });
093:          return editor;
094:       }
095:       catch (InstantiationException exception)
096:       {
097:          return null;
098:       }
099:       catch (IllegalAccessException exception)
100:       {
101:          return null;
102:       }
103:       catch (InvocationTargetException exception)
104:       {
105:          return null;
106:       }
107:    }
108:
109:    /**
110:       Wraps a property editor into a component.
111:       @param editor the editor to wrap
112:       @return a button (if there is a custom editor),
113:       combo box (if the editor has tags), or text field (otherwise)
114:    */
115:    public Component getEditorComponent(final PropertyEditor editor)
116:    {
117:       String[] tags = editor.getTags();
118:       String text = editor.getAsText();
119:       if (editor.supportsCustomEditor())
120:       {
121:          // Make a button that pops up the custom editor
122:          final JButton button = new JButton();
123:          // if the editor is paintable, have it paint an icon
```

```
124:          if (editor.isPaintable())
125:          {
126:             button.setIcon(new
127:                Icon()
128:                {
129:                   public int getIconWidth() { return WIDTH - 8; }
130:                   public int getIconHeight() { return HEIGHT - 8; }
131:
132:                   public void paintIcon(Component c, Graphics g,
133:                      int x, int y)
134:                   {
135:                      g.translate(x, y);
136:                      Rectangle r = new Rectangle(0, 0,
137:                         getIconWidth(), getIconHeight());
138:                      Color oldColor = g.getColor();
139:                      g.setColor(Color.BLACK);
140:                      editor.paintValue(g, r);
141:                      g.setColor(oldColor);
142:                      g.translate(-x, -y);
143:                   }
144:                });
145:          }
146:          else
147:             button.setText(buttonText(text));
148:          // pop up custom editor when button is clicked
149:          button.addActionListener(new
150:             ActionListener()
151:             {
152:                public void actionPerformed(ActionEvent event)
153:                {
154:                   JOptionPane.showMessageDialog(null,
155:                      editor.getCustomEditor());
156:                   if (editor.isPaintable())
157:                      button.repaint();
158:                   else
159:                      button.setText(buttonText(editor.getAsText()));
160:                }
161:             });
162:          return button;
163:       }
164:       else if (tags != null)
```

```
165:       {
166:          // make a combo box that shows all tags
167:          final JComboBox comboBox = new JComboBox(tags);
168:          comboBox.setSelectedItem(text);
169:          comboBox.addItemListener(new
170:             ItemListener()
171:             {
172:                public void itemStateChanged(ItemEvent event)
173:                {
174:                   if (event.getStateChange() == ItemEvent.SELECTED)
175:                      editor.setAsText(
176:                         (String) comboBox.getSelectedItem());
177:                }
178:             });
179:          return comboBox;
180:       }
181:       else
182:       {
183:          final JTextField textField = new JTextField(text, 10);
184:          textField.getDocument().addDocumentListener(new
185:             DocumentListener()
186:             {
187:                public void insertUpdate(DocumentEvent e)
188:                {
189:                   try
190:                   {
191:                      editor.setAsText(textField.getText());
192:                   }
193:                   catch (IllegalArgumentException exception)
194:                   {
195:                   }
196:                }
197:                public void removeUpdate(DocumentEvent e)
198:                {
199:                   try
200:                   {
201:                      editor.setAsText(textField.getText());
202:                   }
203:                   catch (IllegalArgumentException exception)
204:                   {
205:                   }
```

```
206:               }
207:               public void changedUpdate(DocumentEvent e)
208:               {
209:               }
210:            });
211:         return textField;
212:      }
213:   }
214:
215:   /**
216:       Formats text for the button that pops up a
217:       custom editor.
218:       @param text the property value as text
219:       @return the text to put on the button
220:   */
221:   private static String buttonText(String text)
222:   {
223:      if (text == null || text.equals(""))
224:         return " ";
225:      if (text.length() > MAX_TEXT_LENGTH)
226:         return text.substring(0, MAX_TEXT_LENGTH) + "...";
227:      return text;
228:   }
229:
230:   private ArrayList changeListeners = new ArrayList();
231:   private static final int WIDTH = 100;
232:   private static final int HEIGHT = 25;
233:   private static final int MAX_TEXT_LENGTH = 15;
234: }
235:
```

```
01: import java.awt.*;
02: import javax.swing.*;
03:
04: /**
05:    This program tests the property sheet by displaying a
06:    slider and a property sheet to edit the slider’s properties.
07: */
08: public class PropertySheetTest
09: {
10:    public static void main(String[] args)
11:    {
12:       JFrame frame = new JFrame();
13:       Component comp = new JSlider();
14:       frame.getContentPane().add(comp);
15:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:       frame.pack();
17:       frame.show();
18:
19:       JPanel editor = new PropertySheet(comp);
20:       JComponent pane = new JScrollPane(editor);
21:       pane.setPreferredSize(new Dimension(400, 300));
22:       JOptionPane.showMessageDialog(frame, pane);
23:    }
24: }
```