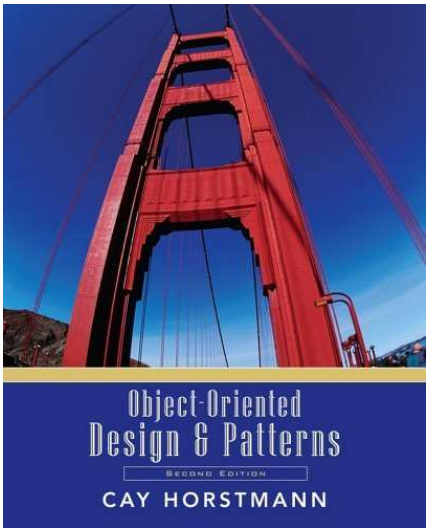
Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 7

The Java Object Model



Chapter Topics

- 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

- 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

Exercise: What kind of type?

- `int`
- `Rectangle`
- `Shape`
- `String[]`
- `double[][]`

Java Values

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

Exercise: What kind of value?

- 13
- `new Rectangle(5, 10, 20, 30);`
- `"Hello"`
- `new int[] { 2, 3, 5, 7, 11, 13 }`
- `null`

Subtype Relationship

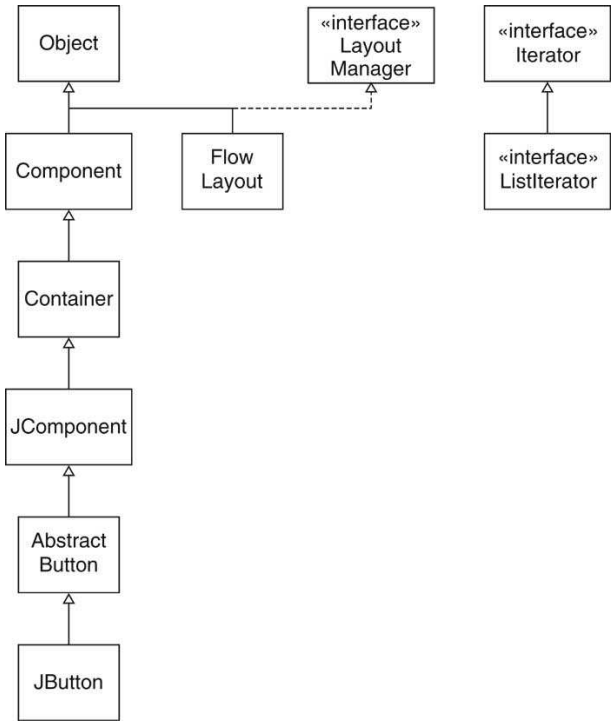
S is a subtype of T if

- 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

Subtype Examples

- `Container` is a subtype of `Component`
- `JButton` is a subtype of `Component`
- `FlowLayout` is a subtype of `LayoutManager`
- `ListIterator` is a subtype of `Iterator`
- `Rectangle[]` is a subtype of `Shape[]`
- `int[]` is a subtype of `Object`
- `int` is not a subtype of `long`
- `long` is not a subtype of `int`
- `int[]` is not a subtype of `Object[]`

Subtype Examples



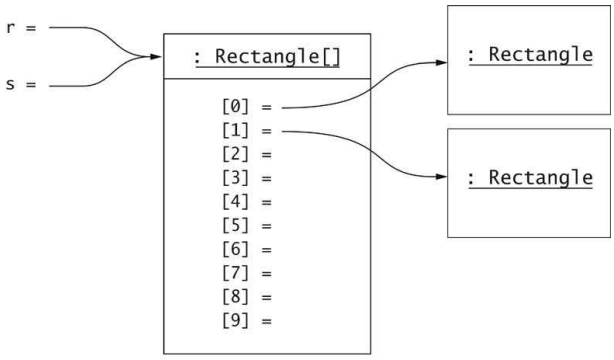
The ArrayStoreException

- `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

Array References



Wrapper Classes

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

```
Integer Short Long ByteCharacter Float Double Boolean
```

- Auto-boxing and auto-unboxing

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


Enumerated Types

- Finite set of values
- Example: `enum Size { SMALL, MEDIUM, LARGE }`
- Typical use:
  ```
  Size imageSize = Size.MEDIUM;
  if (imageSize == Size.SMALL) . . .
  ```
- Safer than integer constants
  ```
  public static final int SMALL = 1;
  public static final int MEDIUM = 2;
  public static final int LARGE = 3;
  ```

Typesafe Enumerations

- `enum` equivalent to class with fixed number of instances

```
public class Size
{
   private /* ! */ Size() {  }
   public static final Size SMALL = new
Size();
   public static final Size MEDIUM = new
Size();
   public static final Size LARGE = new
Size();
}
```

- `enum` types are classes; can add methods, fields, constructors

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)

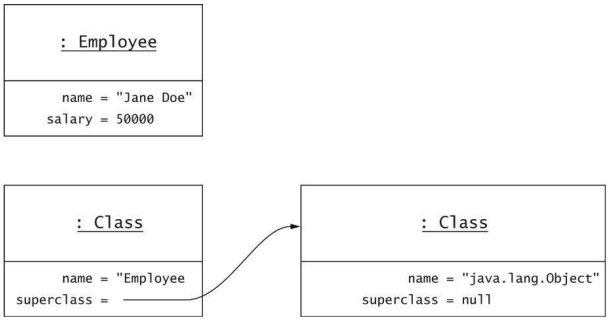
The `Class` Class

- `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 (e 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`

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

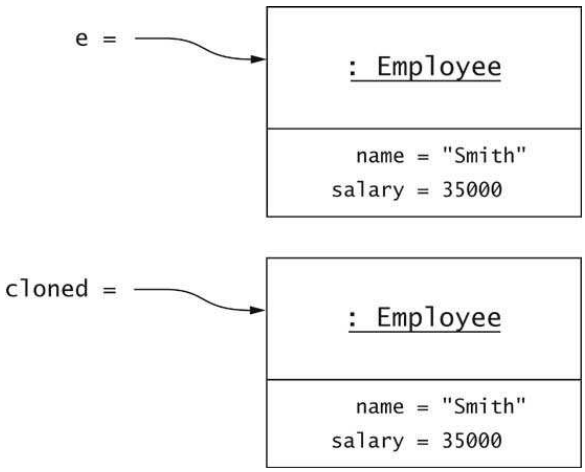
- Must be compatible with `equals`:
  if `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- `Object.hashCode` hashes memory address
- *NOT* compatible with redefined `equals`
- Remedy: Hash all fields and combine codes:

```
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 OObject 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{}
```

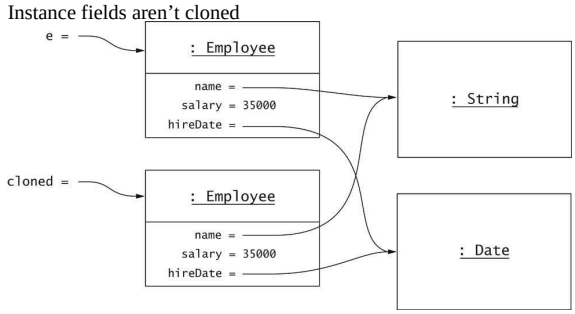
- Interface has no methods!
- Tagging interface--used in test
  `if x implements Cloneable`
- `Object.clone` throws `CloneNotSupportedException`
- A checked exception

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

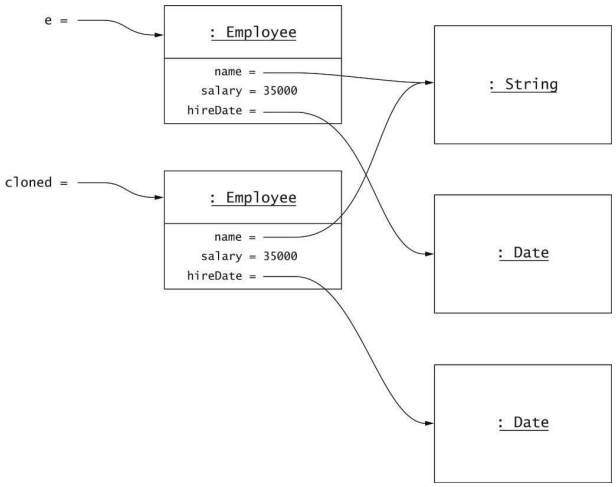


Deep Cloning

- 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



previous | start | next

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

previous | start | next

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();
```

previous | start | next

Serialization

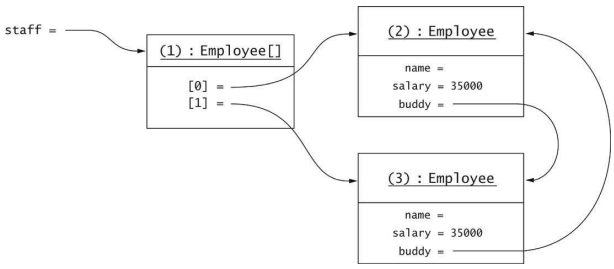
- The array *and all of its objects and their dependent objects* are saved
- `Employee` doesn't have to define any method
- Needs to implement the `Serializable` interface
- Another tagging interface with no methods

previous | start | next

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 #"

How Serialization Works



Serialing 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 RectangularShape.class.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

previous | start | next

Reflection

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

Enumerating Fields

- Print the names of all static fields of the `Math` class:

```
Field[] fields = Math.class.getDeclaredFields(); for (Field f : fields)    if (Modifier.isStatic(f.getModifiers()))       System.out.println(f.getName());
```


Enumerating Constructors

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

```
for (Constructor c : cons) {   Class[] params = c.getParameterTypes();   System.out.print("Rectangle(");   boolean first = true;   for (Class p : params)   {      if (first) first = false; else System.out.print(", ");      System.out.print(p.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", int.class, int.class);
```

- Example: Get default `Rectangle` constructor:

```
Constructor c = Rectangle.class.getDeclaredConstructor();
```

- `getDeclaredMethod`, `getDeclaredConstructor` are varargs methods

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", String.class);m.invoke(System.out, "Hello, World!");
```

- `invoke` is a varargs method

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/FieldTester.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 nonstatic fields of an object.
07: */
08: public class FieldTester
09: {
10:    public static void main(String[] args)
11:          throws IllegalAccessException
12:    {
13:       Random r = new Random();
14:       System.out.print(spyFields(r));
15:       r.nextInt();
16:       System.out.println("\nAfter calling nextInt:\n");
17:       System.out.print(spyFields(r));
18:    }
19:
20:    /**
21:       Spies on the field names and values of an object.
22:       @param obj the object whose fields to format
23:       @return a string containing the names and values of
24:       all fields of obj
25:    */
26:    public static String spyFields(Object obj)
27:          throws IllegalAccessException
28:    {
29:       StringBuffer buffer = new StringBuffer();
30:       Field[] fields = obj.getClass().getDeclaredFields();
31:       for (Field f : fields)
32:       {
33:          if (!Modifier.isStatic(f.getModifiers()))
34:          {
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:       }
45:       return buffer.toString();
46:    }
47: }
```

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);
```


Generic Types

- A generic type has one or more type variables
- Type variables are instantiated with class or interface types
- Cannot use primitive types, e.g. no `ArrayList<int>`
- When defining generic classes, use type variables in definition:

```
public class ArrayList<E>{   public E get(int i) { . . . }   public E set(int i, E newValue) { . . . }   . . .   private E[] elementData;}
```

- NOTE: If `S` a subtyoe of `T`, `ArrayList<S>` is *not* a subtype of `ArrayList<T>`.

Generic Methods

- Generic method = method with type parameter(s)
-
```
public class Utils{   public static <E> void fill(ArrayList<E> a, E value, int count)   {      for (int i = 0; i < count; i++)         a.add(value);   } }
```
- A generic method in an ordinary (non-generic) class
- Type parameters are inferred in call

```
ArrayList<String> ids = new ArrayList<String>();Utils.fill(ids, "default", 10); // calls Utils.<String>fill
```


Type Bounds

- Type variables can be constrained with type bounds
- Constraints can make a method more useful
- The following method is limited:

```
public static <E> void append(ArrayList<E> a, ArrayList<E> b, int count){   for (int i = 0; i < count && i < b.size(); i++)      a.add(b.get(i));}
```

  Cannot append an `ArrayList<Rectangle>` to an `ArrayList<Shape>`

Type Bounds

- Overcome limitation with type bound:

```
public static <E, F extends E> void append(   ArrayList<E> a, ArrayList<F> b, int count){   for (int i = 0; i < count && i < b.size(); i++)      a.add(b.get(i));}
```

- `extends` means "subtype", i.e. extends or implements
- Can specify multiple bounds:
  `E extends Cloneable & Serializable`

Wildcards

- Definition of `append` never uses type `F`. Can simplify with *wildcard*:

```
public static <E> void append(   ArrayList<E> a, ArrayList<? extends E> b, int count){   for (int i = 0; i < count && i < b.size(); i++)      a.add(b.get(i));}
```


Wildcards

- Wildcards restrict methods that can be called:
  `ArrayList<? extendsE>.set` method has the form
  `? extends E add(? extends E newElement)`
- You cannot call this method!
- No value matches `? extends E` because ? *is unknown*
- Ok to call `get`:
  `? extends E get(int i)`
- Can assign return value to an element of type `E`

Wildcards

- Wildcards can be bounded in opposite direction
- `? super F` matches any supertype of `F`
- ```
  public static <F> void append(
     ArrayList<? super F> a, ArrayList<F> b,
  int count)
  {
     for (int i = 0; i < count && i < b.size();
  i++)
        a.add(b.get(i));
  }
  ```
- Safe to call `ArrayList<? super F>.add`:
  `boolean add(? super F newElement)`
- Can pass any element of type `F` (but not a supertype!)
  ```

Wildcards

- Typical example--start with

```
public static <E extends Comparable<E>> E getMax(ArrayList<E> a){   E max = a.get(0);   for (int i = 1; i < a.size(); i++)      if (a.get(i).compareTo(max) > 0) max = a.get(i);   return max;}
```

- `E` extends `Comparable<E>` so that we can call `compareTo`
- Too restrictive--can't call with `ArrayList<GregorianCalendar>`
- `GregorianCalendar` *does not* implement `Comparable<GregorianCalendar>`, only `Comparable<Calendar>`
- Wildcards to the rescue:

```
public static <E extends Comparable<? super E>> E getMax(ArrayList<E> a)
```


Type Erasure

- Virtual machine does not know about generic types
- Type variables are *erased*--replaced by type bound or `Object` if unbounded
- Ex. `ArrayList<E>` becomes

```
public class ArrayList{   public Object get(int i) { . . . }   public Object set(int i, Object newValue) { . . . }   . . .   private Object[] elementData;}
```

- Ex. `getmax` becomes

```
public static Comparable getMax(ArrayList a)   // E extends Comparable<? super E> erased to Comparable
```

- Erasure necessary to interoperate with legacy (pre-JDK 5.0) code

Limitations of Generics

- Cannot replace type variables with primitive types
- Cannot construct new objects of generic type

```
a.add(new E()); // Error--would erase to new Object()
```

- Workaround: Use class literals

```
public static <E> void fillWithDefaults(ArrayList<E>,   Class<? extends E> cl, int count)   throws InstantiationException, IllegalAccessException{   for (int i = 0; i < count; i++)      a.add(cl.newInstance());}
```

- Call as `fillWithDefaults(a, Rectangle.class, count)`

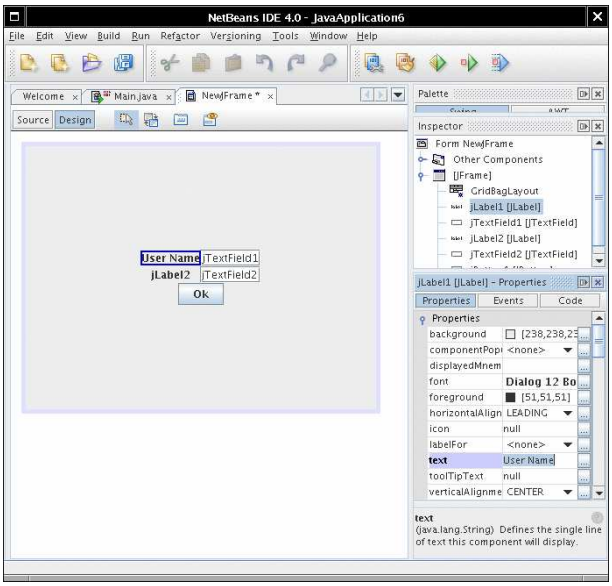
Limitations of Generics

- Cannot form arrays of parameterized types `Comparable<E>[]` is illegal. Remedy: `ArrayList<Comparable<E>>`
- Cannot reference type parameters in a static context (static fields, methods, inner classes)
- Cannot throw or catch generic types
- Cannot have type clashes after erasure. Ex. `GregorianCalendar` cannot implement `Comparable<GregorianCalendar>` since it already implements `Comparable<Calendar>`, and both erase to `Comparable`

Components

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

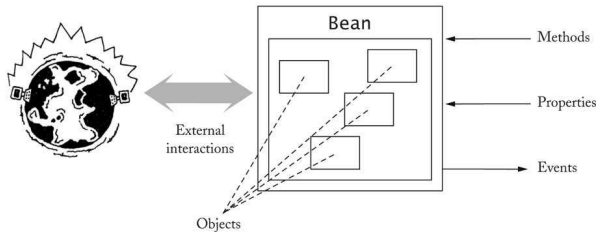
A Builder Environment



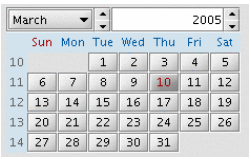
Java Beans

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

Java Beans

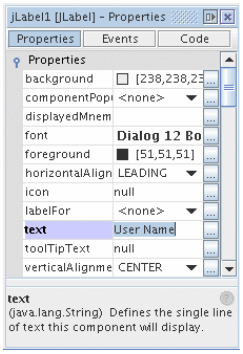


A Calendar Bean



A Property Sheet

- Edit properties with *property sheet*



Façade Class

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

Façade Pattern

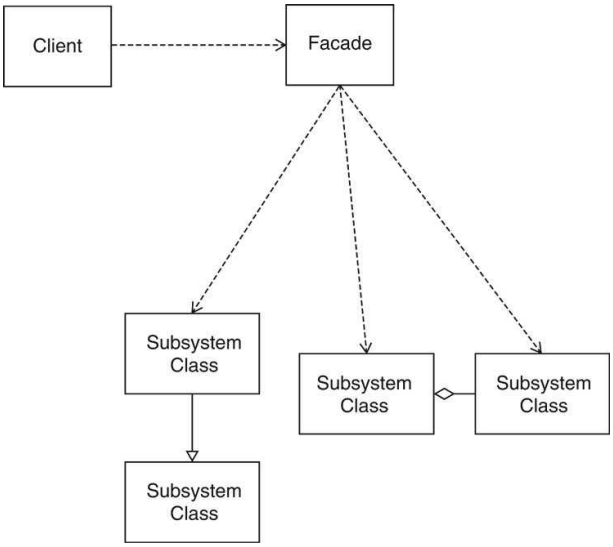
Context

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

Façade Pattern



Façade 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

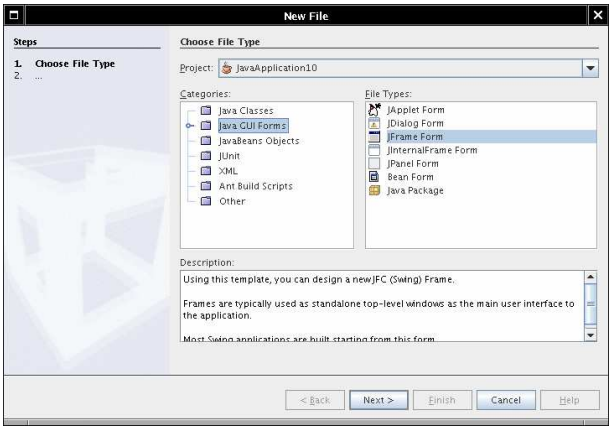
- Not Java :-(
- C#, JavaScript, Visual Basic
- `b.`*`propertyName`*` = value`
  calls setter
- `variable = b.`*`propertyName`*
  calls getter

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`

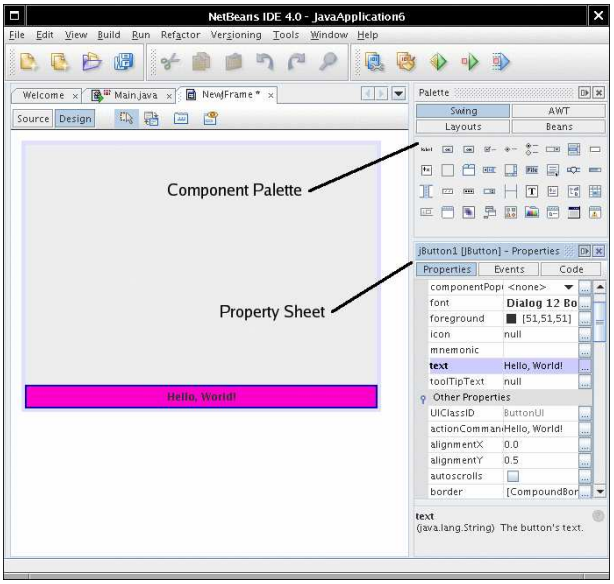
Editing Beans in a Builder Tool

- Use wizard to make empty frame



Editing Beans in a Builder Tool

- Add button to frame
- Edit button with property sheet



Packaging a Bean

- Compile bean classes
  Ch7/carbean/CarBean.java
- Create manifest file
  Ch7/carbean/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 component that draws a car shape.
007: */
008: public class CarBean extends JComponent
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:       Graphics2D g2 = (Graphics2D) g;
062:       Rectangle2D.Double body
063:          = new Rectangle2D.Double(x, y + height / 3,
064:             width - 1, height / 3);
065:       Ellipse2D.Double frontTire
066:          = new Ellipse2D.Double(x + width / 6,
067:             y + height * 2 / 3, height / 3, height / 3);
068:       Ellipse2D.Double rearTire
069:          = new Ellipse2D.Double(x + width * 2 / 3,
070:             y + height * 2 / 3, height / 3, height / 3);
071:
072:       // The bottom of the front windshield
073:       Point2D.Double r1
074:          = new Point2D.Double(x + width / 6, y + height / 3);
075:       // The front of the roof
076:       Point2D.Double r2
077:          = new Point2D.Double(x + width / 3, y);
078:       // The rear of the roof
079:       Point2D.Double r3
080:          = new Point2D.Double(x + width * 2 / 3, y);
081:       // The bottom of the rear windshield
082:       Point2D.Double r4
083:          = new Point2D.Double(x + width * 5 / 6, y + height / 3);
084:
085:       Line2D.Double frontWindshield
086:          = new Line2D.Double(r1, r2);
087:       Line2D.Double roofTop
088:          = new Line2D.Double(r2, r3);
```

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

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

Composing Beans