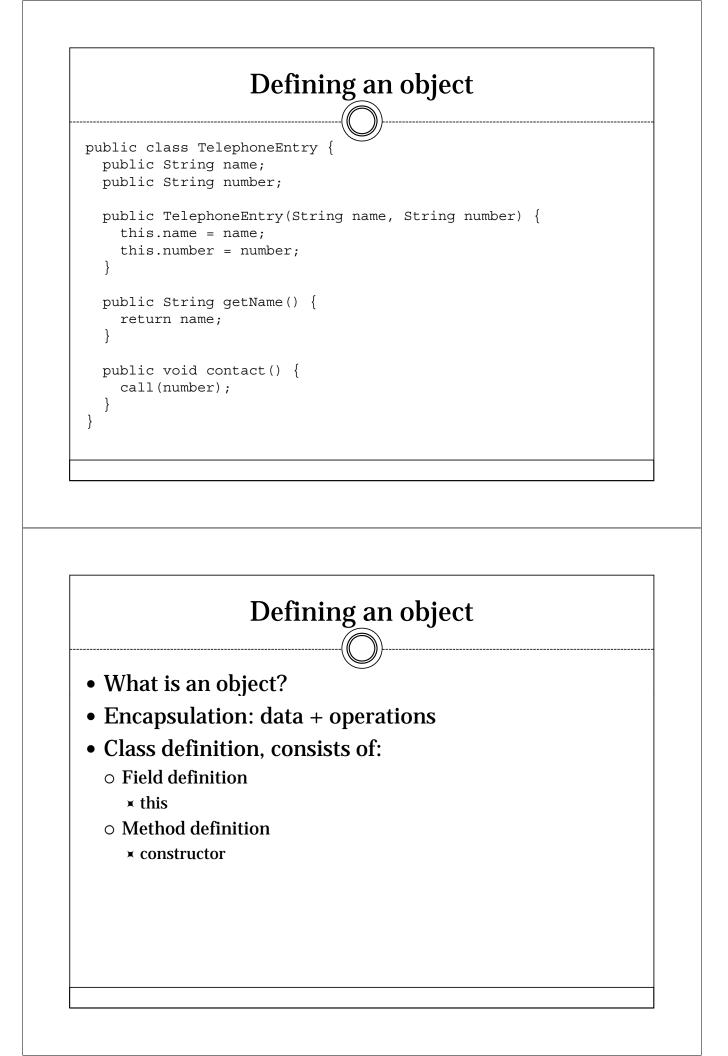


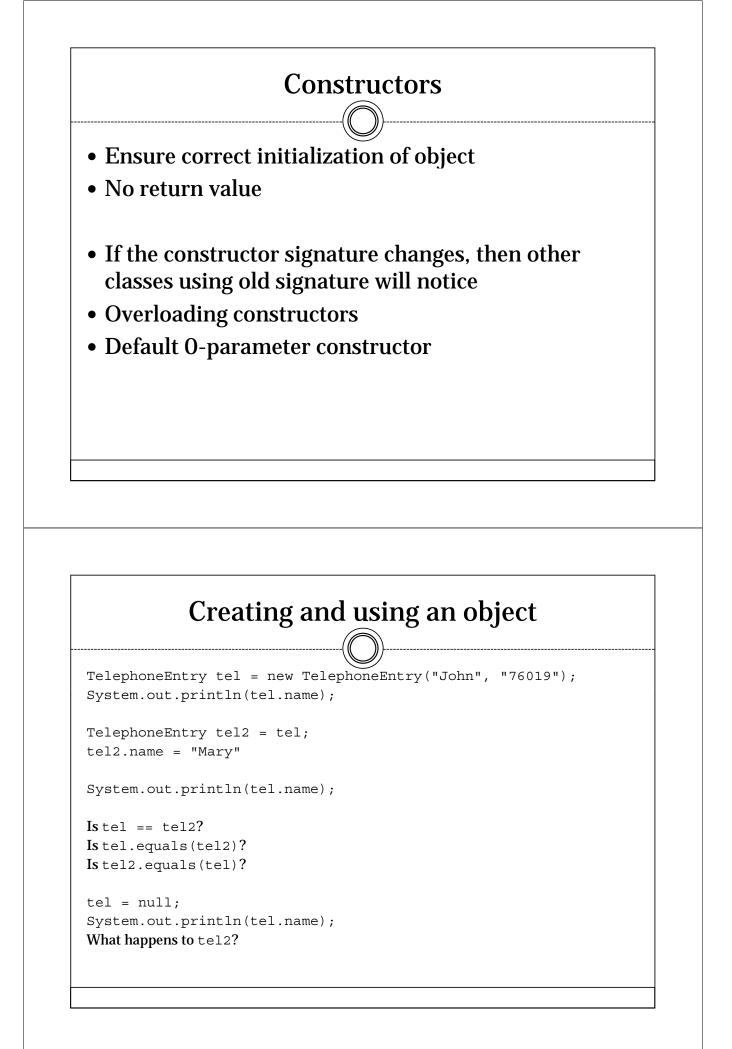
Reading list

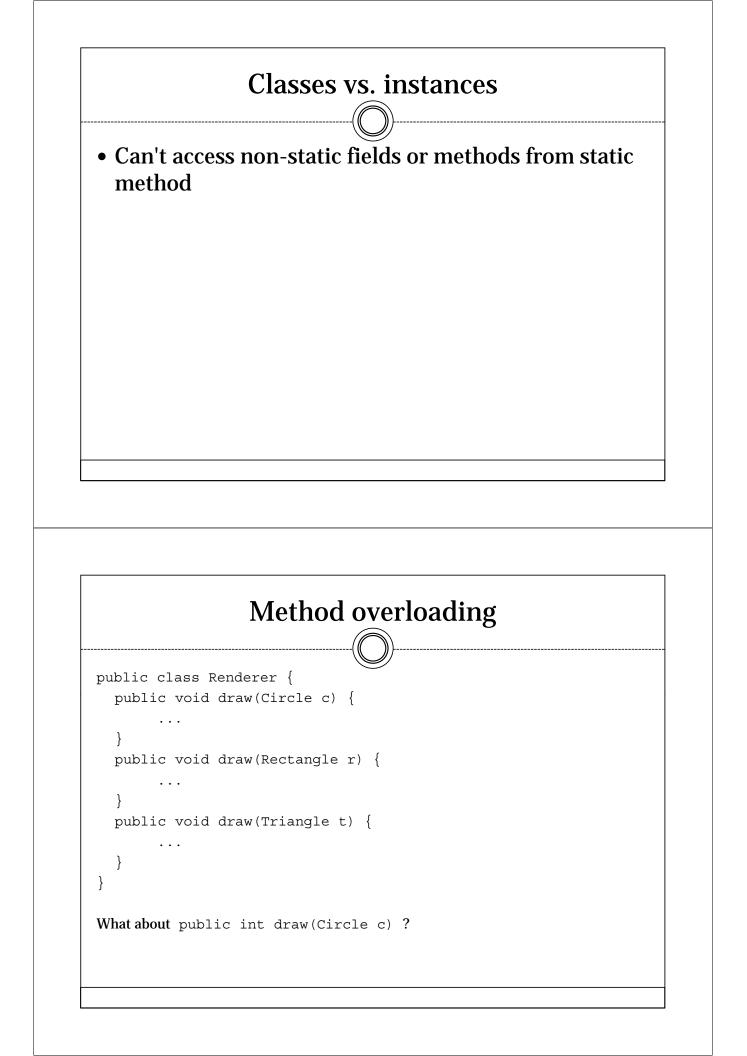
Bruce Eckel, *Thinking in Java* (2006) http://www.mindview.net/Books/TIJ4/ Erich Gamma et al., *Design Patterns* (1994) Doug Lea, *Concurrent Programming in Java* (1999) Jean Bacon, *Concurrent Systems* (2002) Jean Bacon and Tim Harris, *Operating Systems: Concurrent and Distributed Software Design* (2003) Glenford Myers et al., *The Art of Software Testing* (2004) James Gosling et al., *The Java Language Specification* (2005) http://java.sun.com/docs/books/jls/

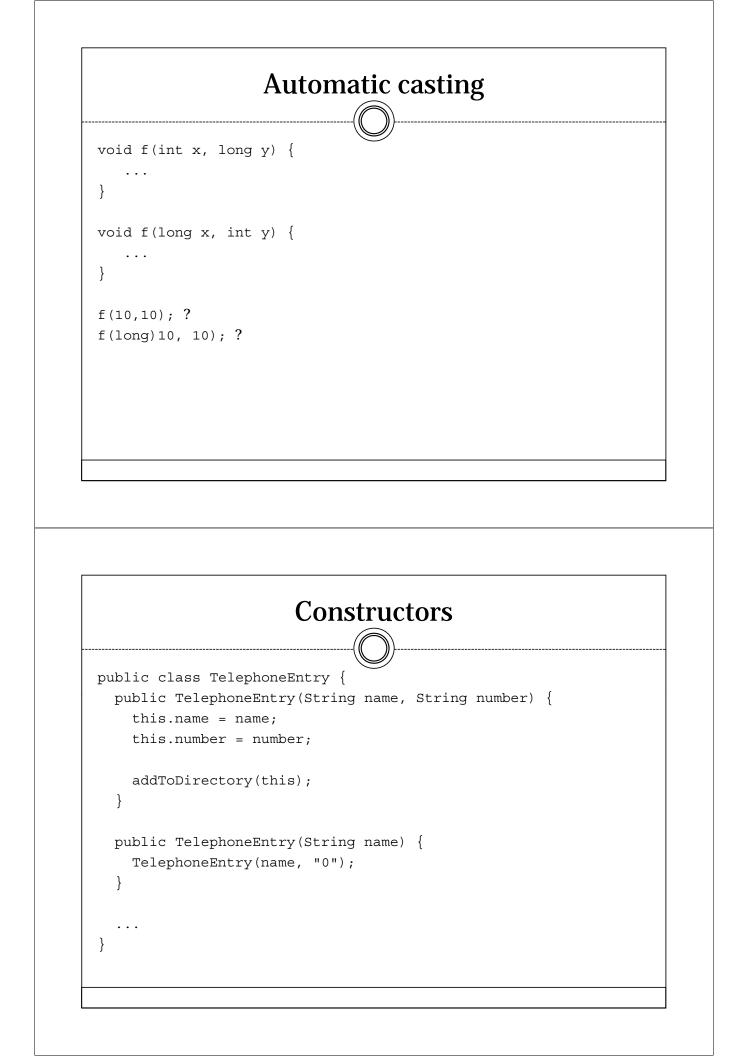
Review: Programming with Objects

- Introduction
- Objects and classes
- Packages
- Interfaces
- Nested classes
- Design patterns

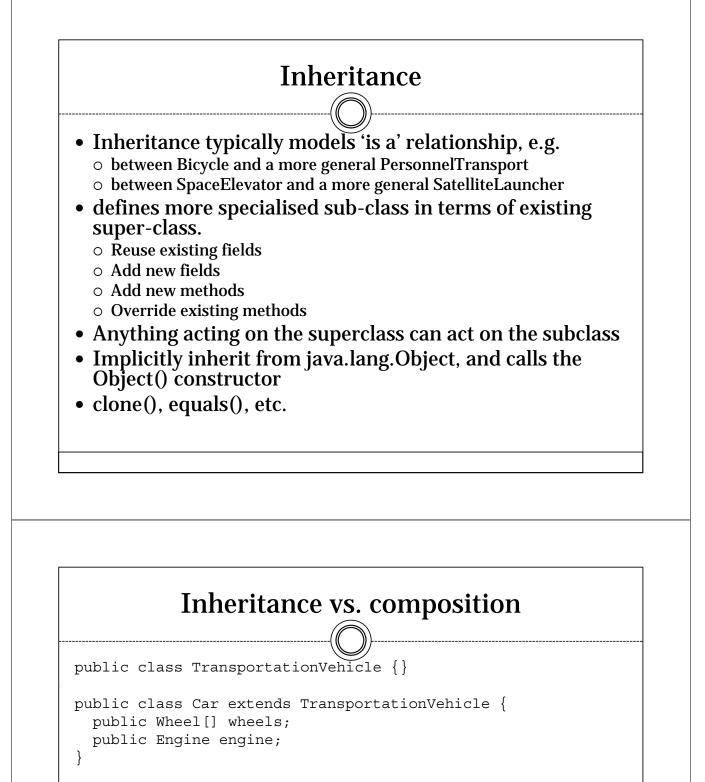












public class Plane extends TransportationVehicle {
 public Wing[] wings;
 public Tail tail;

Car **is a** TransportationVehicle Car **has an** Engine

}

Inheritance and references

```
class A { ... }
class B extends A { ... }
A objectA = new A();
B objectB = new B();
A refToA;
B refToB;
refToA = objectA;
refToA = objectB;
Do these assignments work?
refToB = refToA;
refToB = (B) refToA;
```

refToB = (B) objectA;

 Object references

 • Objects instantiated by new

 • (constructor implicitly called)

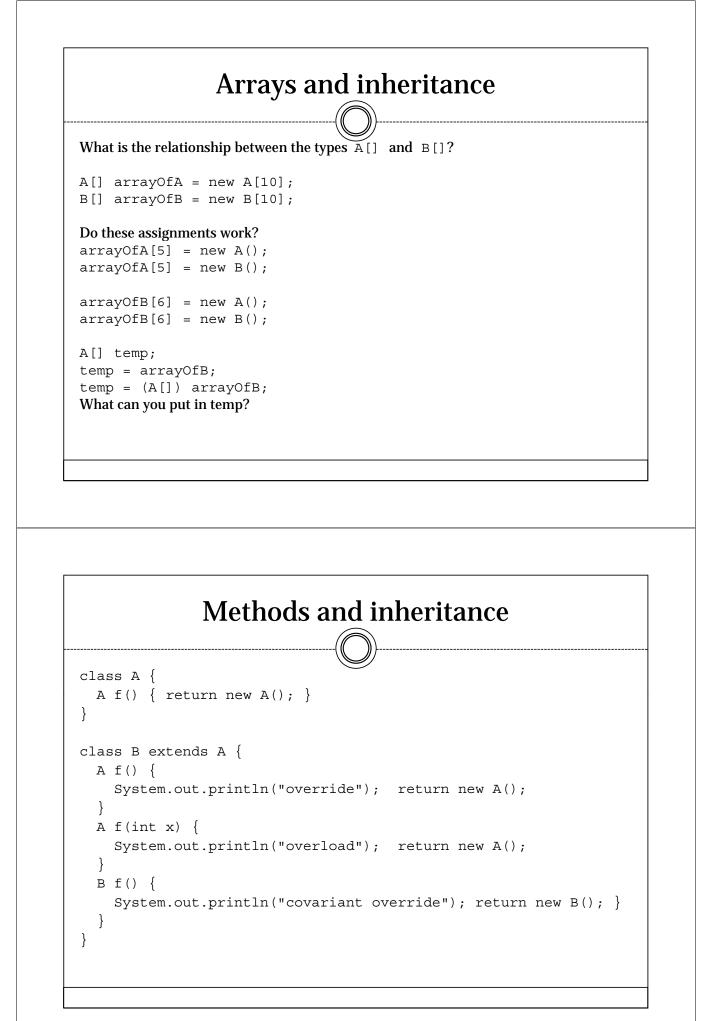
 • Objects manipulated through references

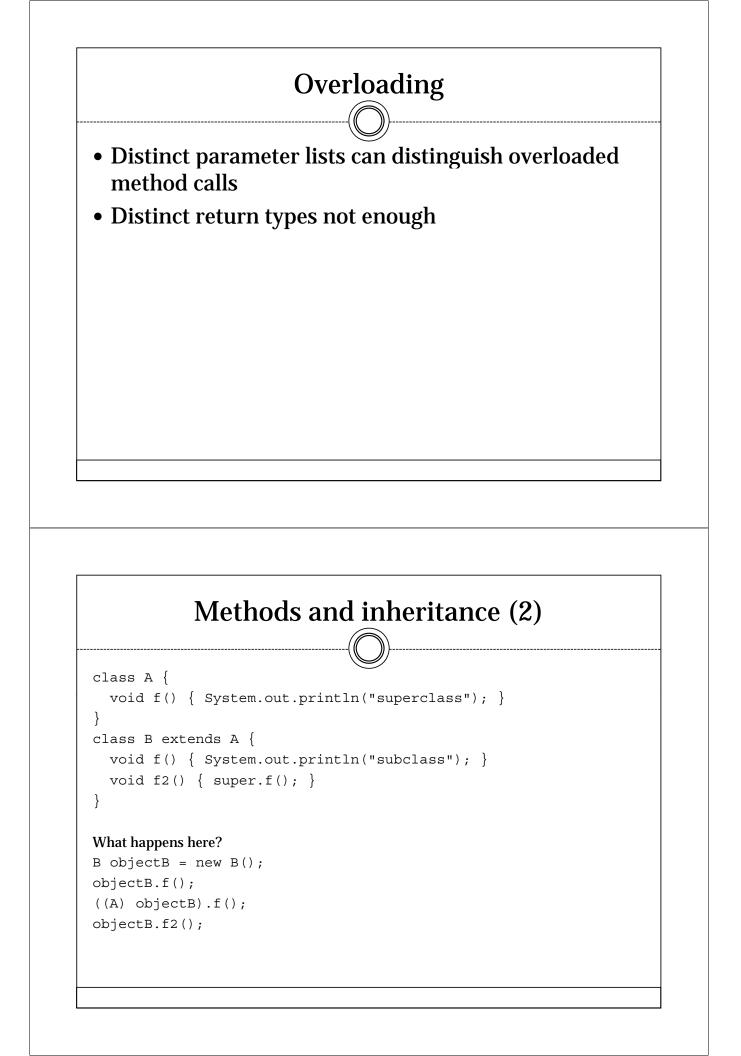
 • either a particular instance, or null

 • Two references can refer to the same object

 • Different types of object equality

 • Garbage collection

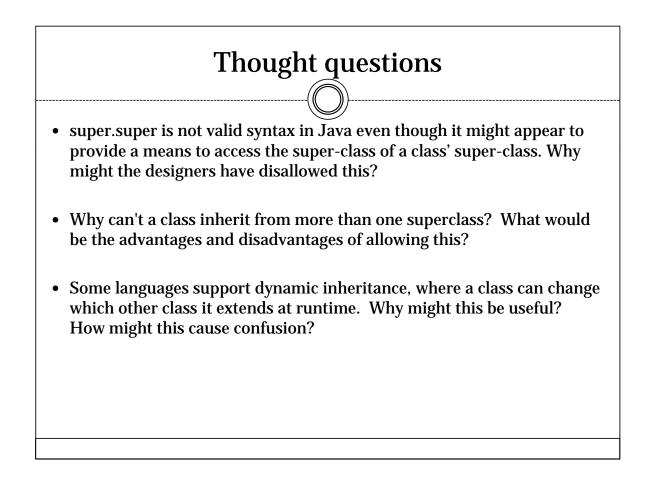


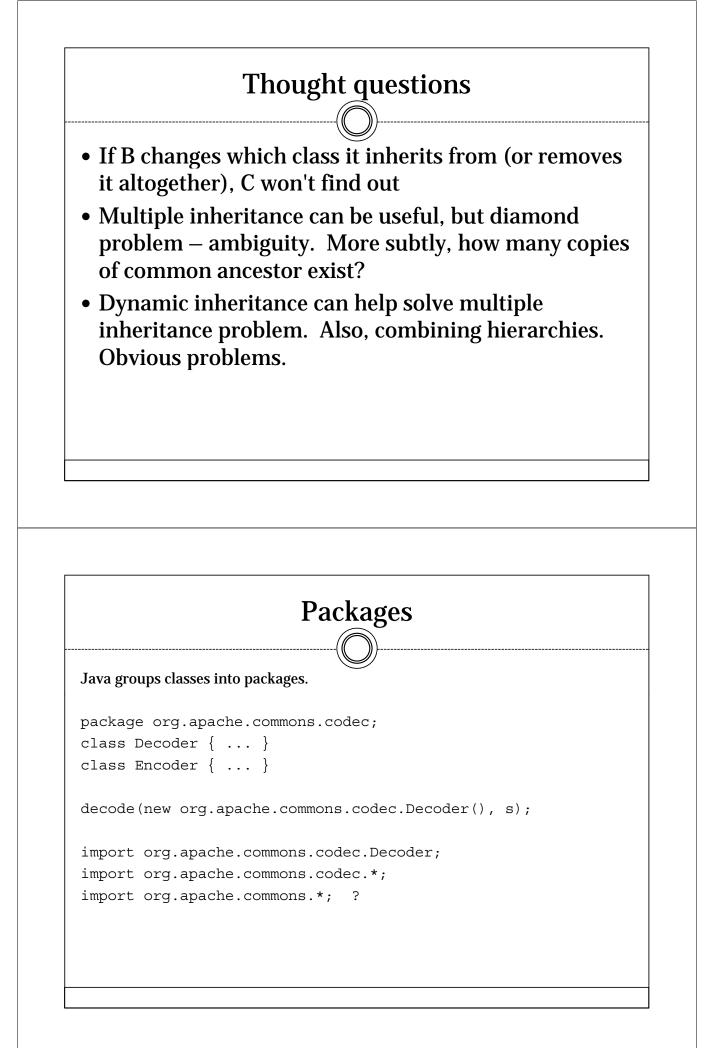


Methods and inheritance (3)

```
public class Renderer {
   public void draw(Shape s) {
      s.draw();
   }
}
public class Circle extends Shape {
   public void draw() { ... }
}
public class Rectangle extends Shape {
   public void draw() { ... }
}
public class Triangle extends Shape {
   public void draw() { ... }
}
```

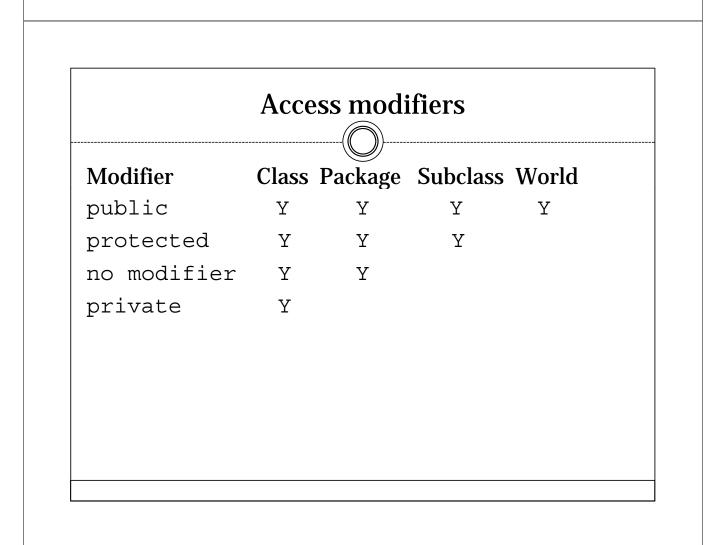
The dual of polymorphism.

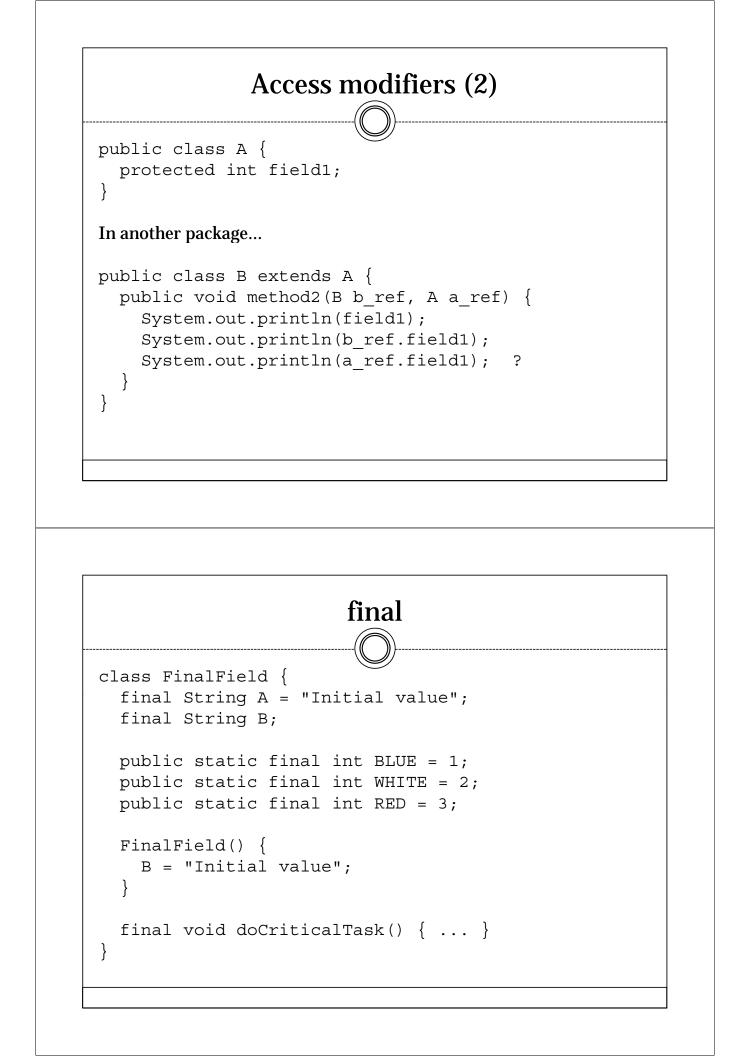


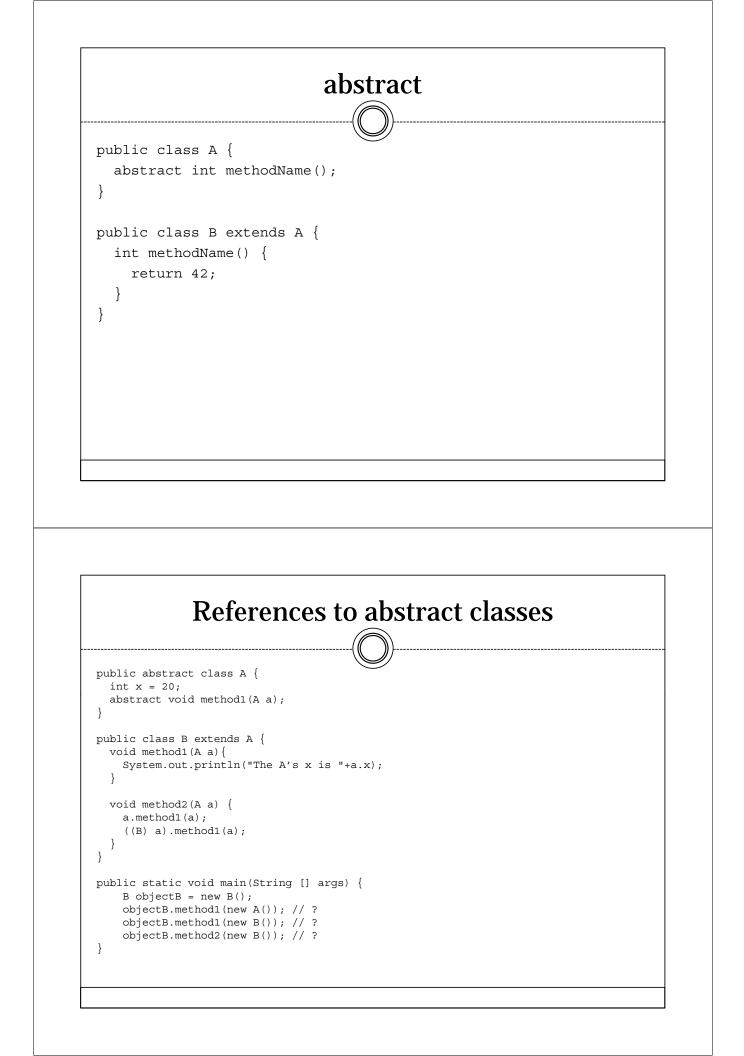


Packages

- Classes within a package are typically written by cooperating programmers and are expected to be used together.
- fully qualified name = package + class name
- Don't have to create the package in any way; just quote the name in any package statement.
- Some compilers create subdirectories in the file system, nesting one directory level for each full stop in the package's fully-qualified name.
- Packages don't inherit





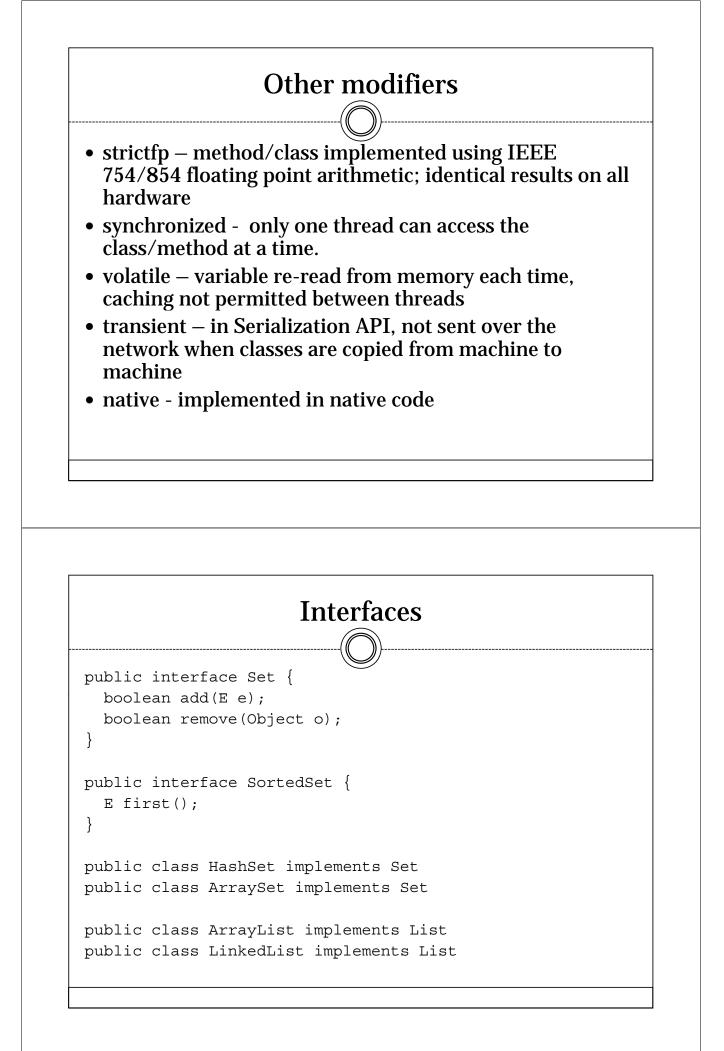


Combining common functionality

```
public abstract AbstractList {
 public abstract Object get(int index);
 public List subList(int fromIndex, int toIndex) {
  }
}
class java.util.AbstractMap
class java.util.AbstractList
                           static
• Can be applied to any method or field definition (also
  nested classes)
• The field/method is associated with the class as a whole
  rather than with any particular object.
• There is only one value for the whole class, rather than a
  separate value for each object.
• Similarly, static methods are not associated with a
  current object—unqualified instance field names and the
  this keyword cannot be used.

    Static methods called by explicitly naming the class

  within which the method is defined. The named class is
  searched, then its super-class, etc. Otherwise the search
  begins from the class in which the method call is made.
```

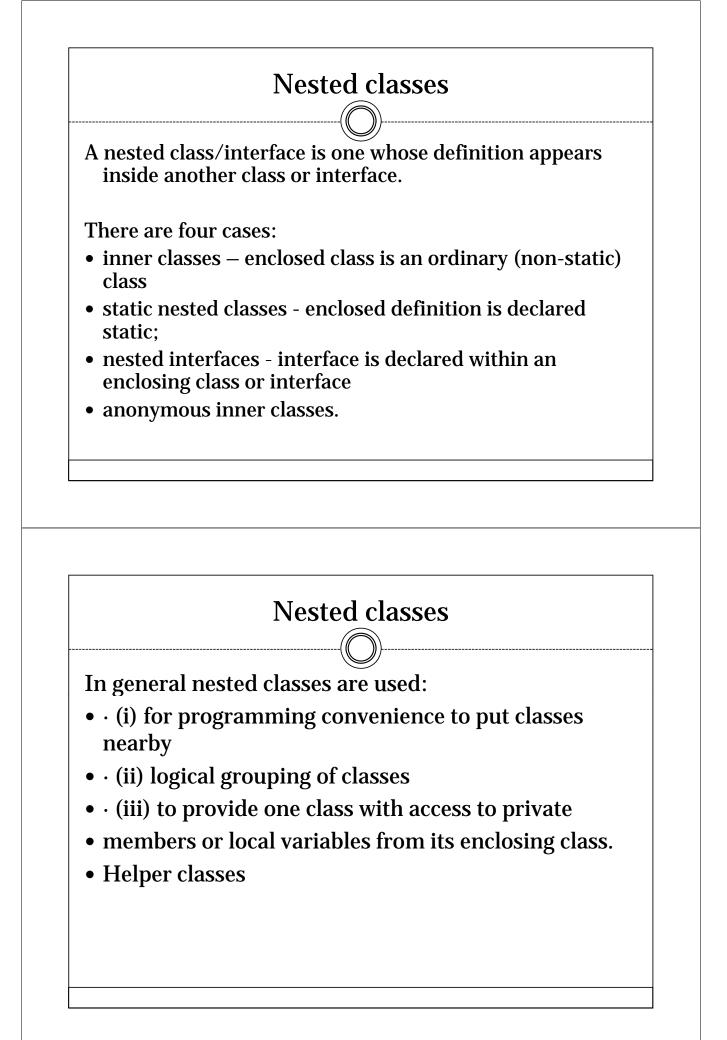


Interfaces

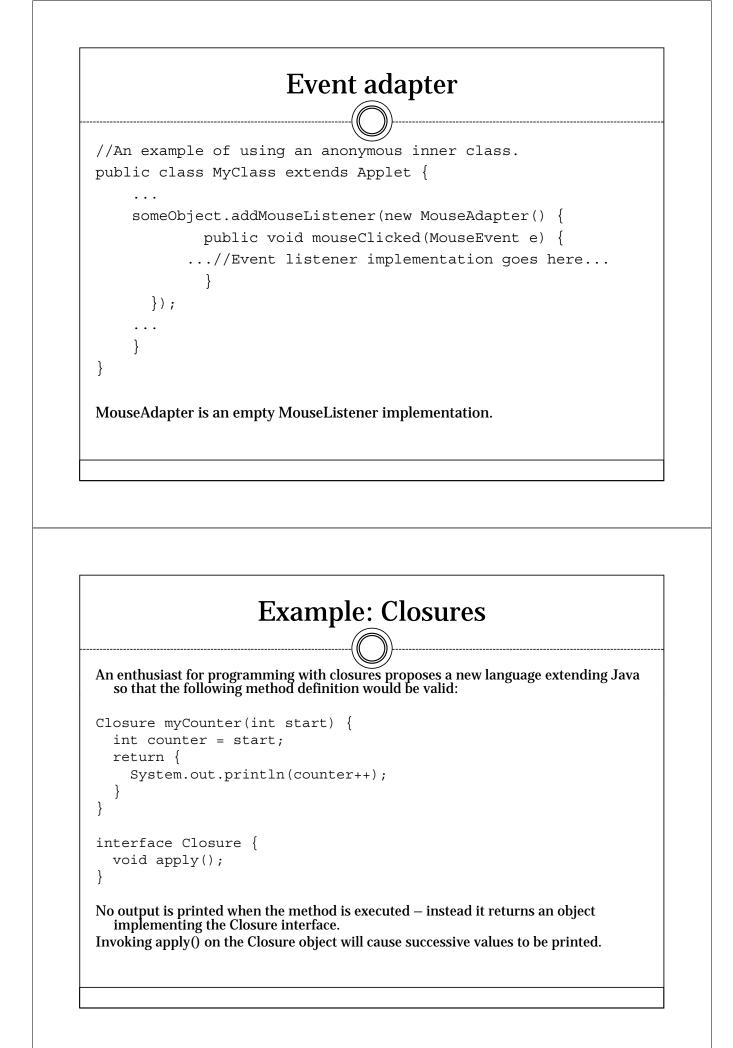
- · Groups of classes that provide different
- implementations of the same kind of functionality.
- • e.g. the collection classes in java.util–HashSet and
- ArraySet provide set operations; ArrayList and
- LinkedList provide list-based operations.
- In that example there are some operations available on all
- collections, further operations on all sets, and a third set of
- operations on the HashSet class itself.
- Inheritance and abstract classes can be used to move
- common functionality into super-classes such as
- Collection and Set.
- • Each class can only has a single super-class (in Java),
- so should HashSet extend a class representing the
- hashtable aspects of its behaviour (capacity, load factor), or a class
- representing the set-like operations available on it?
- More generally, it is often desireable to separate the
- definition of a standard programming interface (e.g.
- set-like operations) from their implementation using an
- actual data structure (e.g. a hash table).

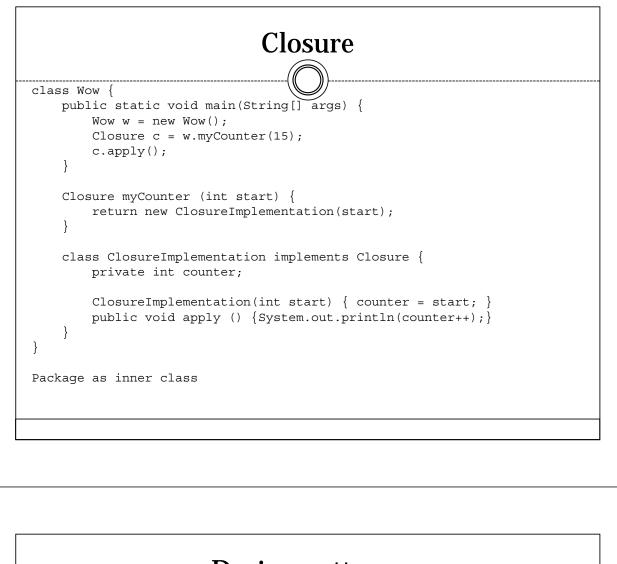
Interfaces

- Each Java class may extend only a single super-class, but it can implement a number of interfaces.
- An interface definition just declares method signatures and static final fields (constants).
- An ordinary interface may have public or default access.
- javaAll methods and fields are implicitly public.
- An interface may extend one or more super-interfaces.
- A class that implements an interface must either: supply definitions for each of the declared methods; or be declared an abstract class.

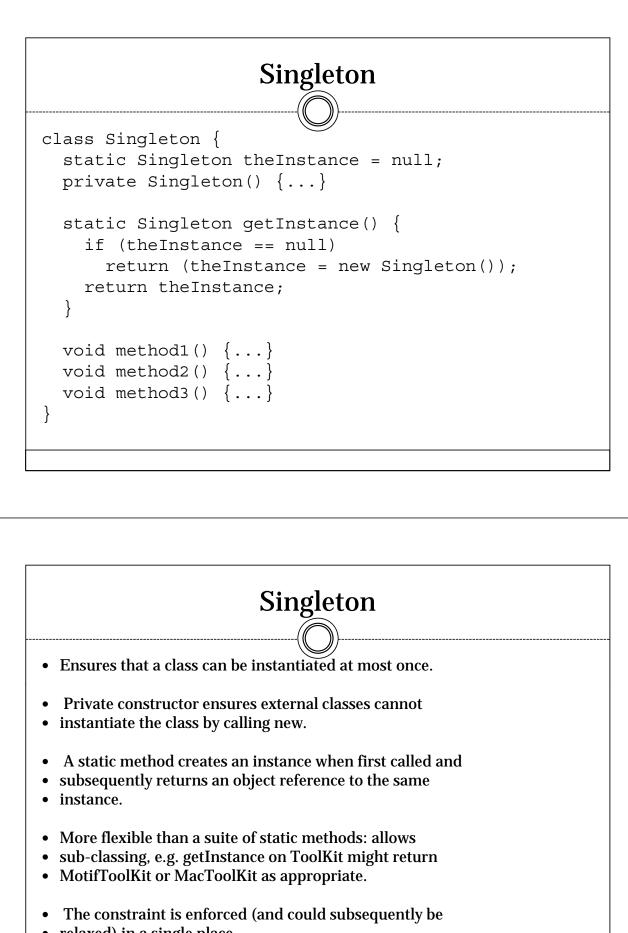


• An inner cla enclosing cl	ass definition associates each instance of the enclosed class with an instance of the ass, e.g.
class Bus { Engine e;	
class Whe	el {
···· } }	
Each instance access the f	of Wheel is associated with an enclosing instance of Bus. e.g. Wheel methods can ield without qualification or access the enclosing Bus as Bus.this.
Can Bus find o	ut what Wheels are associated with it?
static nestednested inter	d class - not associated with any instance of an enclosing class. faces – implicitly static
	Anonymous inner class
	Anonymous inner class
class A {	y of defining inner classes.
class A { void method	y of defining inner classes.
class A { void method Object ref void me	y of defining inner classes.
class A { void method Object ref	y of defining inner classes. al () { = new Object() {
<pre>class A { void method Object ref void me }; } } An anonymous ir</pre>	y of defining inner classes. al () { = new Object () { ethod2 () {}; mer class may be defined using an interface name rather than a class name—providing inline
<pre>class A { void method Object ref void me }; } An anonymous ir implementations</pre>	y of defining inner classes. Al () { = new Object() { ethod2() {};
<pre>class A { void method Object ref void me }; } } An anonymous ir implementations class A { void method Ifc i = r</pre>	<pre>y of defining inner classes. iii () { = new Object() { ethod2() {}; mer class may be defined using an interface name rather than a class name—providing inline of all the methods. iii () { hew Ifc() { </pre>
<pre>class A { void method Object ref void me }; } } An anonymous ir implementations class A { void method Ifc i = r publi };</pre>	<pre>y of defining inner classes. al () {</pre>
<pre>class A { void method Object ref void me }; } } An anonymous ir implementations class A { void method Ifc i = r public </pre>	<pre>y of defining inner classes. iii () { = new Object() { ethod2() {}; mer class may be defined using an interface name rather than a class name—providing inline of all the methods. iii () { hew Ifc() { </pre>

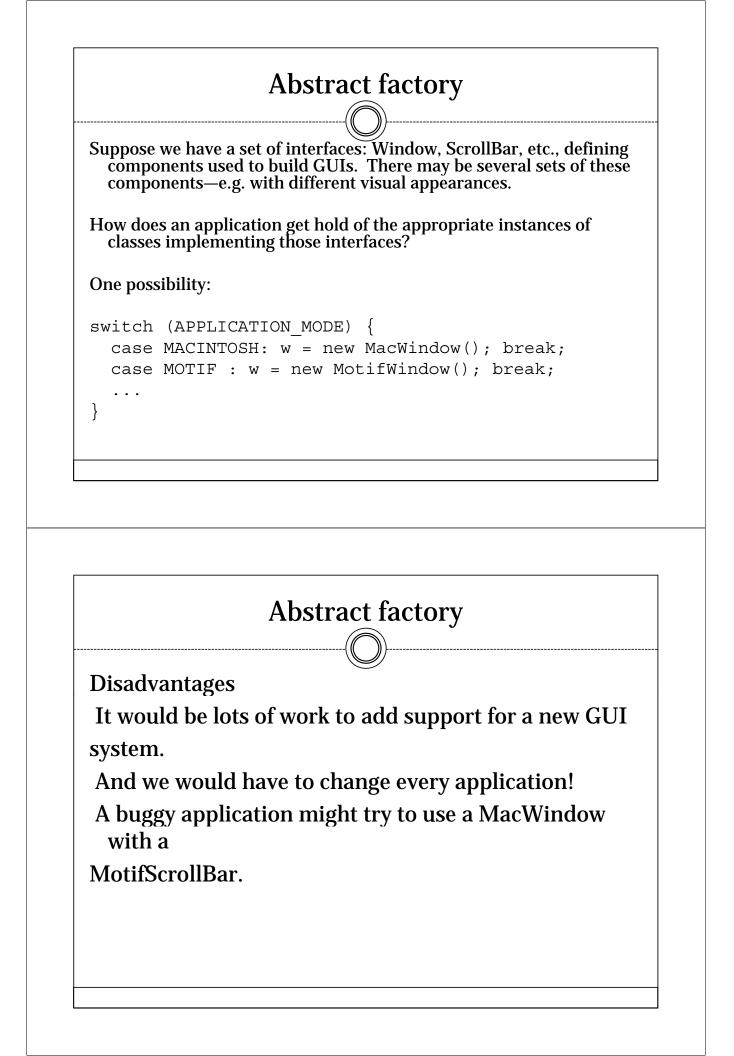


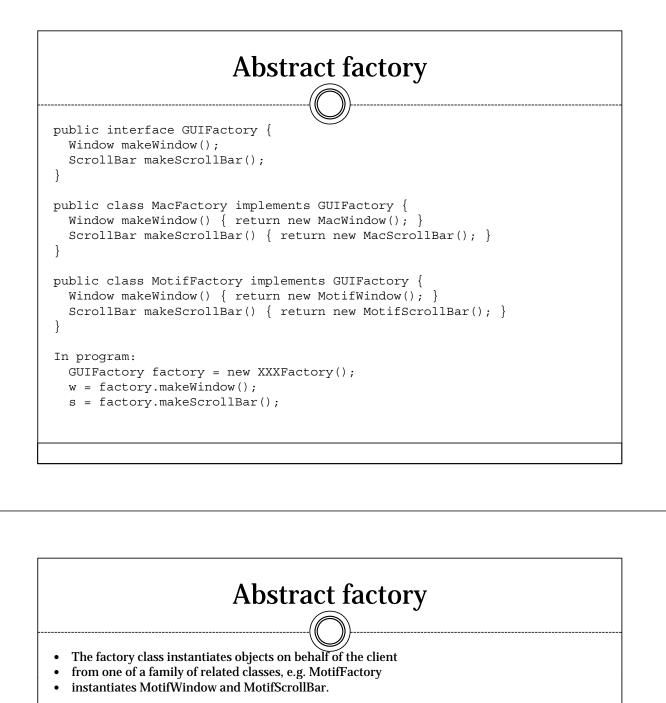


Design patterns Common idioms frequently emerge in object-oriented programming. Studying these design patterns provides: • common terminology for describing program organisation and conveying the purposes of interrelated classes; and • examples of how to structure programs for flexibility and re-use.

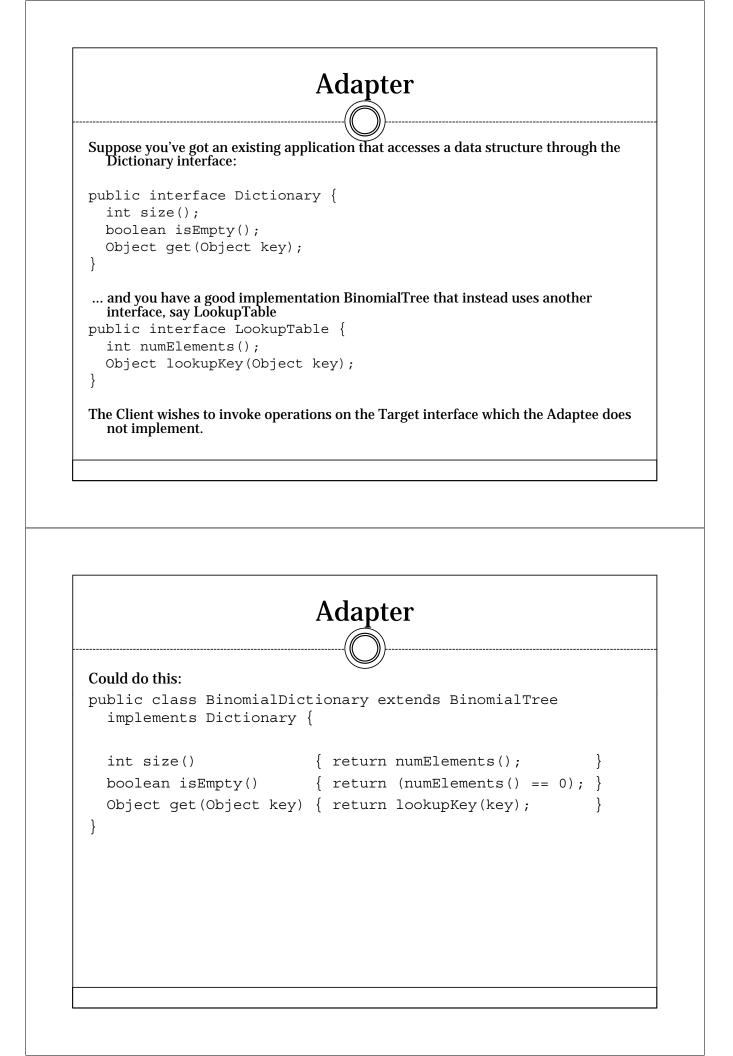


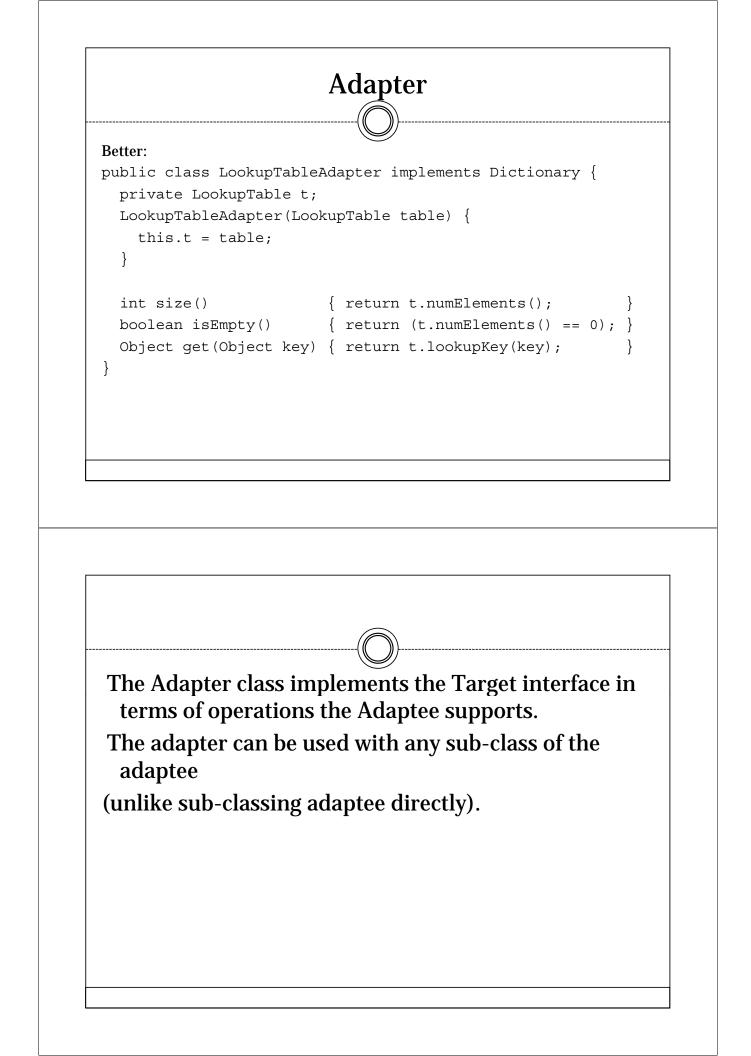
• relaxed) in a single place.



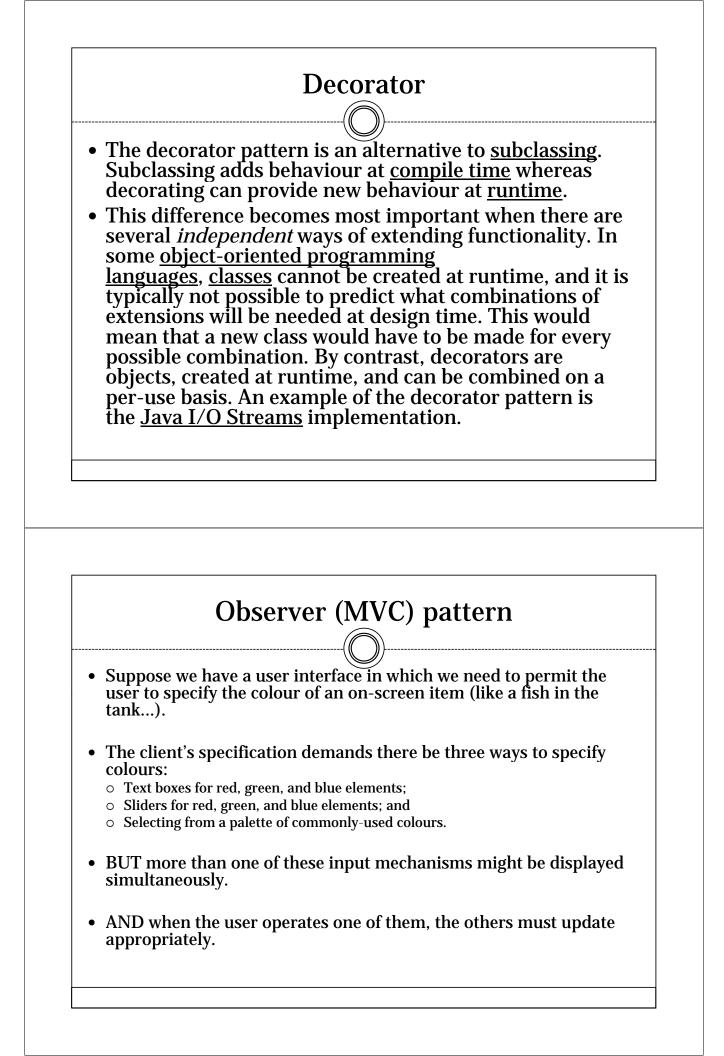


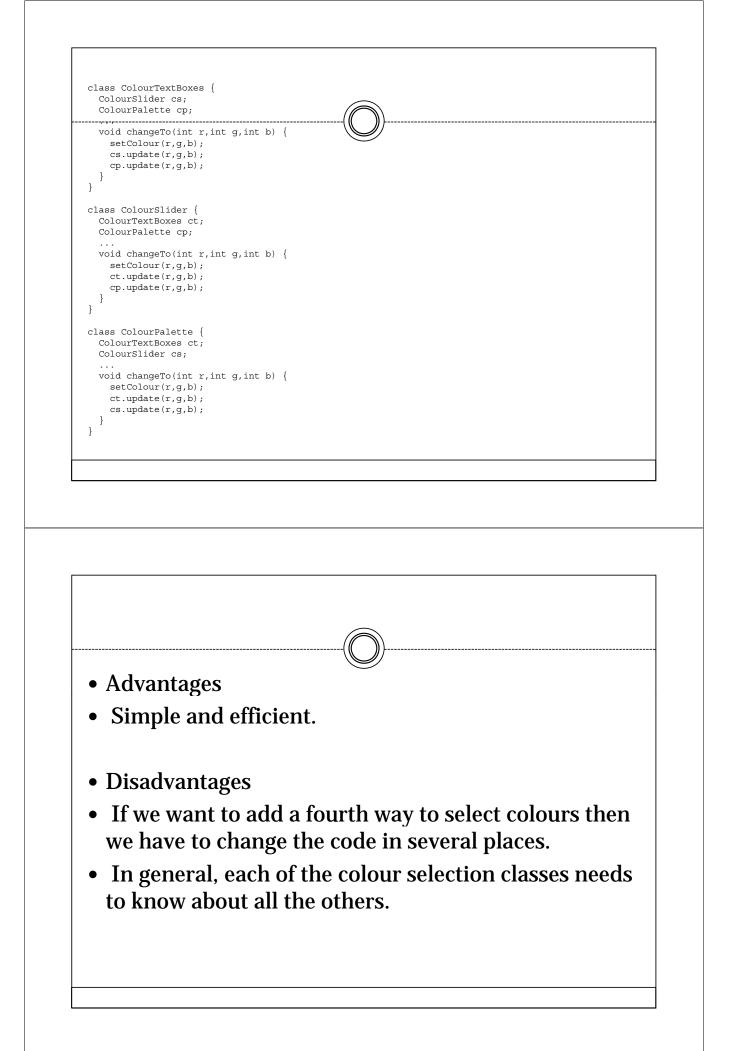
- New families can be introduced by providing the client with
- an instance of a new sub-class of Factory.
- The factory can ensure classes are instantiated
- consistently—e.g. MotifWindow always with
- MotifScrollBar.
- Adding a new operation involves co-ordinated change to
- the Factory class and all its sub-classes.
- ... but the problem hasn't entirely gone away: how does
- the application know which Factory to use?
- An instance of more general Strategy pattern





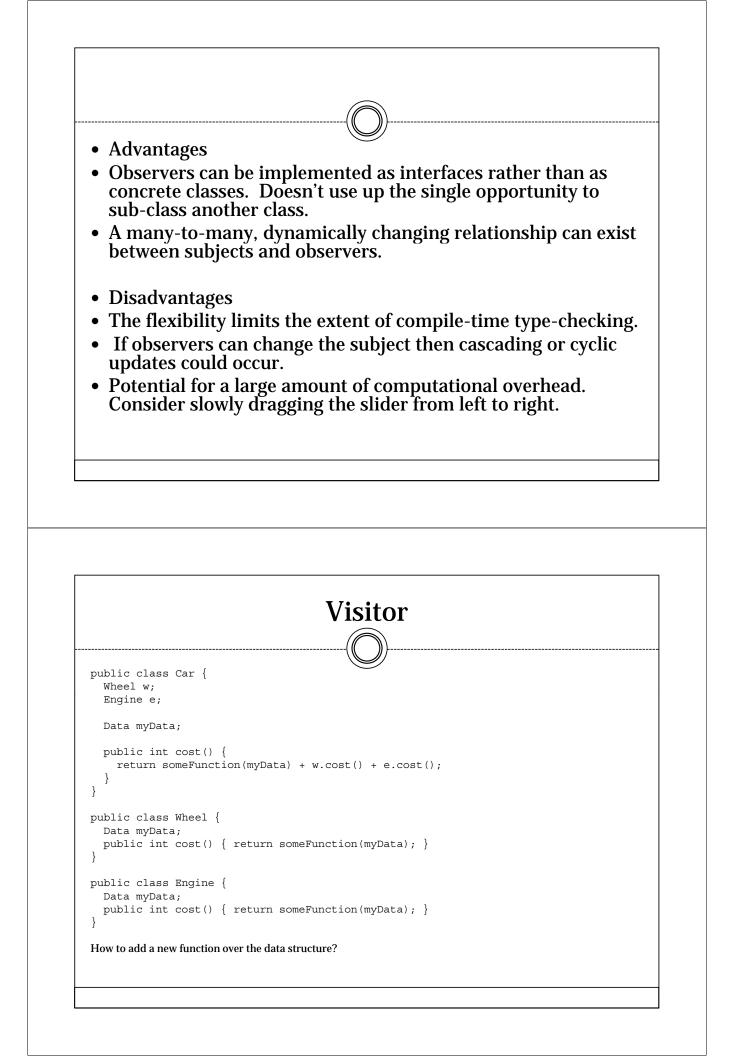
Decorator			
<pre>public class (void write() }</pre>			
public class H	BufferedOutputStream extends OutputStream		
Suppose we want t	to add a cipher capability to all streams, buffered or not:		
public class (BufferedOut	CipheredBufferedOutputStream extends putStream		
public class (OutputStream	CipheredUnbufferedOutputStream extends m		
	worse with more functionalities:		
_	edBufferedOutputStream eredBufferedOutputStream etc		
	Decorator		
	Decorator		
	Decorator		
interface OutputSt			
interface OutputSt	rream		
class SimpleOutput	tream Stream implements OutputStream		
class SimpleOutput abstract class Out	tream tream tream implements OutputStream trputStreamDecorator implements OutputStream {		
class SimpleOutput abstract class Out protected Output public OutputStr	tream tream tream implements OutputStream trputStreamDecorator implements OutputStream {		
class SimpleOutput abstract class Out protected Output	tream Stream implements OutputStream cputStreamDecorator implements OutputStream { tStream os;		
class SimpleOutput abstract class Out protected Output public OutputStr	tream Stream implements OutputStream cputStreamDecorator implements OutputStream { tStream os;		
<pre>class SimpleOutput abstract class Out protected Output public OutputStr this.os = os; } } class BufferedOutp</pre>	<pre>cream cream cstream implements OutputStream cputStreamDecorator implements OutputStream { cstream os; reamDecorator(OutputStream os) { putStream extends OutputStreamDecorator { }</pre>		
<pre>class SimpleOutput abstract class Out protected Output public OutputStr this.os = os; } } class BufferedOutp write(byte[] b) // do bufferin</pre>	<pre>tream tream tream tream treamDecorator implements OutputStream { treamOcorator(OutputStream os) { treamDecorator(OutputStream os) { treamDecorator { { } } }</pre>		
<pre>class SimpleOutput abstract class Out protected Output public OutputStr this.os = os; } } class BufferedOutp write(byte[] b)</pre>	<pre>tream tream tream tream treamDecorator implements OutputStream { treamOcorator(OutputStream os) { treamDecorator(OutputStream os) { treamDecorator { { } } }</pre>		
<pre>class SimpleOutput abstract class Out protected Output public OutputStr this.os = os; } } class BufferedOutp write(byte[] b) // do bufferin</pre>	<pre>tream tream tream tream treamDecorator implements OutputStream { treamOcorator(OutputStream os) { treamDecorator(OutputStream os) { treamDecorator { { } } }</pre>		
<pre>class SimpleOutput abstract class Out protected Output public OutputStr this.os = os; } } class BufferedOutp write(byte[] b) // do bufferin</pre>	<pre>tream tream tream tream treamDecorator implements OutputStream { treamOcorator(OutputStream os) { treamDecorator(OutputStream os) { treamDecorator { { } } }</pre>		
<pre>class SimpleOutput abstract class Out protected Output public OutputStr this.os = os; } } class BufferedOutp write(byte[] b)</pre>	<pre>cream cream cream implements OutputStream cputStreamDecorator implements OutputStream { cstream os; ceamDecorator(OutputStream os) { putStream extends OutputStreamDecorator { { { g; } ceam = new CipheredOutputStream(new BufferedOutputStream(new } </pre>		

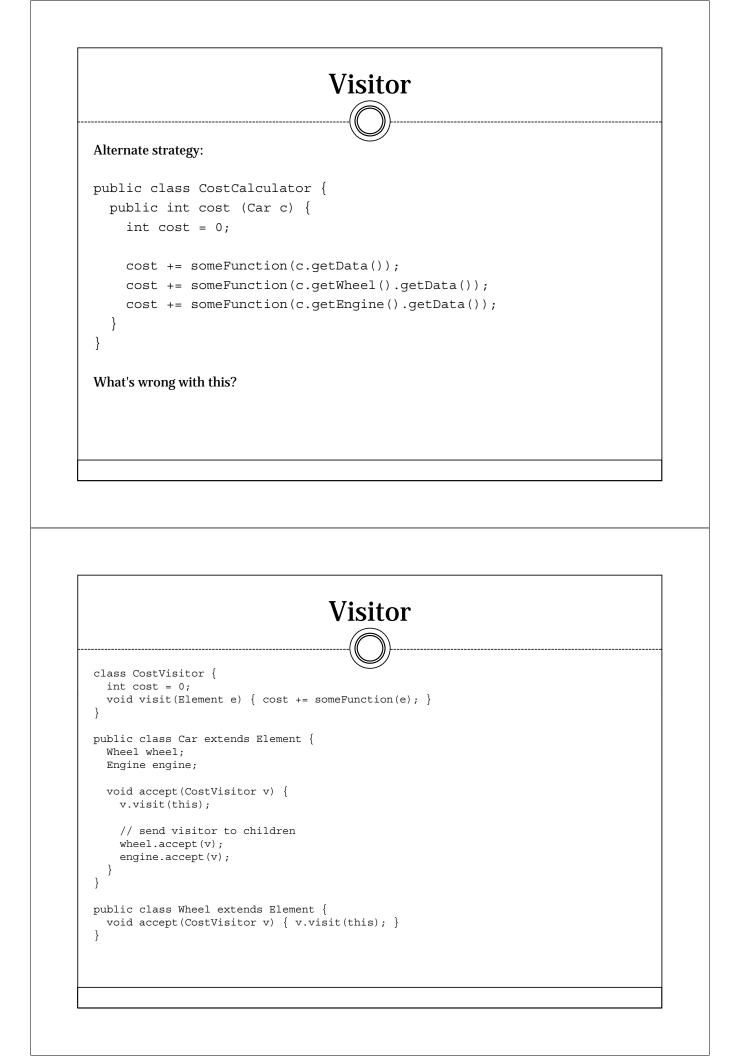


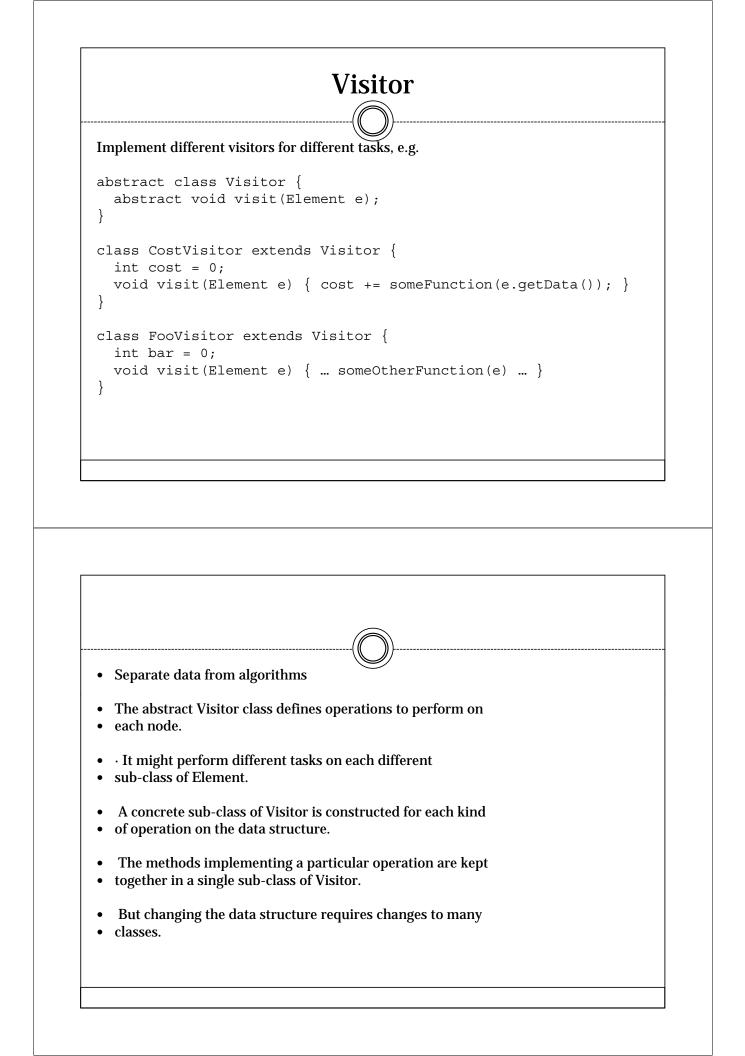


```
class Colour {
  int r,g,b;
  ColourTextBoxes ct;
  ColourSlider cs;
  ColourPalette cp;
-----void-setColour(int-r, int-g, int-b)--{---
   this.r = r; this.g = g; this.b = b;
   ct.update();
   cs.update();
   cp.update();
 }
}
class ColourTextBoxes extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}
class ColourSlider extends Colour {
  void changeTo(int r, int g, int b) {
   setColour(r,g,b);
  }
}
class ColourPalette extends Colour {
 void changeTo(int r,int g,int b) {
   setColour(r,g,b);
  }
}
• Advantages
• Remains simple.
• Efficient.
• Easier to maintain.
• Disadvantages
• Messy-why should Colour have to know about all
  the sub-classes?
• Adds clutter to Colour.
```

class Colour { int r,g,b; void setColour(int r,int g,int b) { this.r = r;this.g = g; -----this.b-=-b;-----_____ } } abstract class ColourObserver { ColourSubject colsubj; abstract void update(); } class ColourSubject { Colour c; ColourObserver observers []; void setColour(int r, int g, int b) { c.setColour(r,g,b); for (int x=0;x<observers.length;++x)</pre> observers[x].update(); } void addObserver(ColourObserver co) { /* insert into observers[] */ . . . } } class ColourSlider extends ColourObserver { void update() { /* read colour, redraw our GUI */ . . . } void changeTo(int r, int g, int b) { colsubj.setColour(r,g,b); } } class ColourTextBoxes extends ColourObserver class ColourPalette extends ColourObserver







Common themes Explicitly creating objects by specifying their class commits to a particular implementation. It is often better to separate code responsible for instantiating objects—Abstract Factory and Singleton patterns. Extending functionality by subclassing commits at compile time to a particular organisation of extensions. Composition and delegation may be preferable – Adapter and Decorator patterns Tight coupling between classes makes independent reuse difficult. Separate data storage from different ways of operating on the data – Observer and Visitor patterns