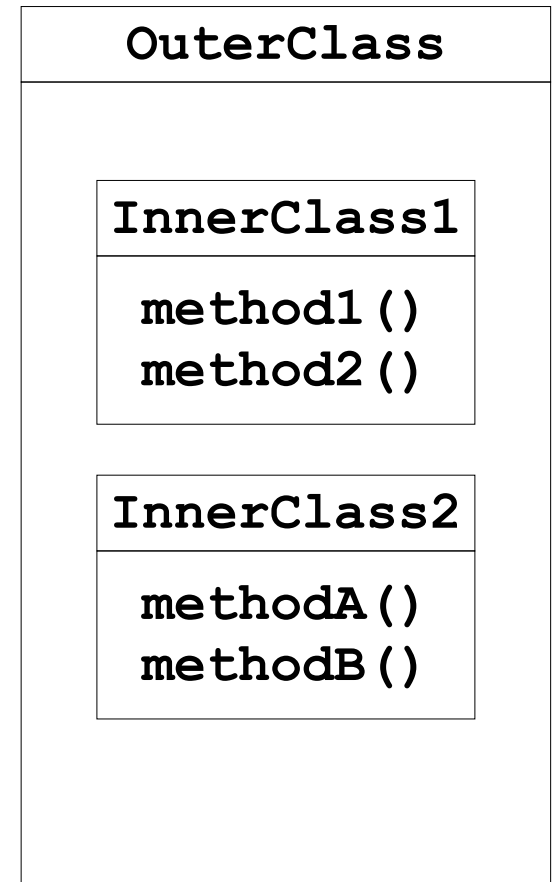


Inner Classes

- Introduction
 - The basics of inner classes
 - The *state design pattern* revisited.
 - Summary
- Moving on with Inner Classes (NOT TODAY)
 - Accessing data in the outer classes
 - A callback mechanism implemented with interfaces and inner classes
 - Nesting interfaces
 - Creating instances of inner classes
 - Anonymous inner classes
 - The template design pattern
 - Summary

What is an Inner Class?

- Main idea: Nest a class within a class
- Class names can be hidden
- More than hiding and organization
 - Call-back mechanism.
 - Can access members of enclosing object.
- Fundamental new language feature
 - added in Java 1.1
- Used a lot in JFC/Swing (GUI programming)



Inner Classes, Example

```
public class Parcel1 { // [Source: bruceeckel.com]
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
}
```

Interfaces and Inner Classes

- An outer class will often have a method that returns a reference to an inner class.

```
// [Source: bruceeckel.com]
public interface Contents {
    int value();
}

public interface Destination {
    String readLabel();
}
```

Interfaces and Inner Classes, cont

```
public class Parcel3 { // [Source: bruceeckel.com]
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }

    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }

    public Destination dest(String s) {
        return new PDestination(s);
    }
    public Contents cont() {
        return new PContents();
    }
}
```

Interfaces and Inner Classes, cont

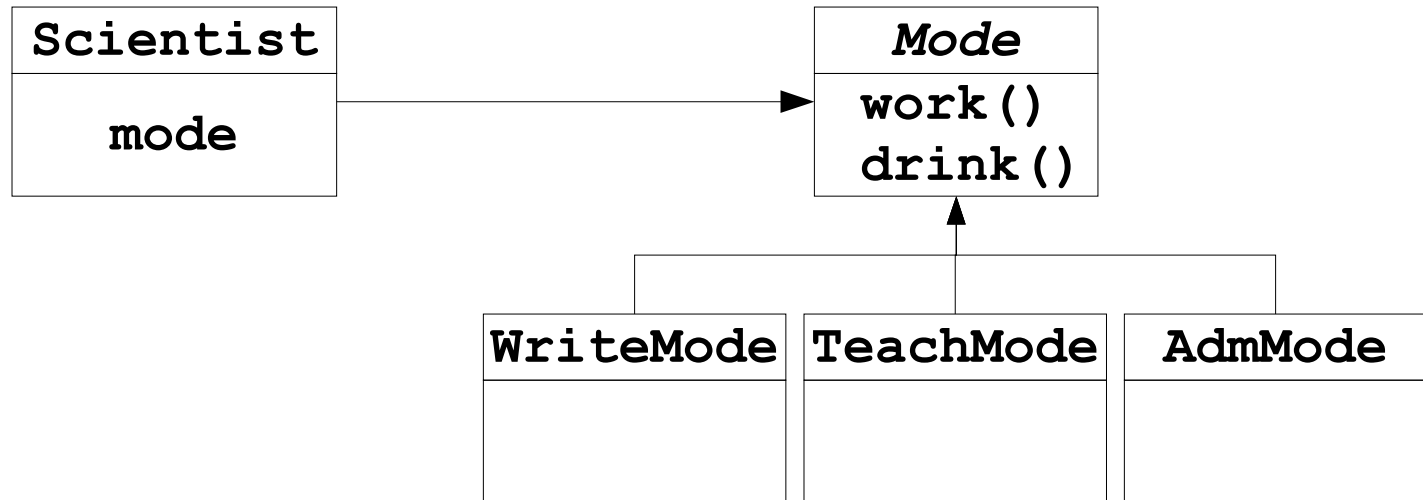
```
// [Source: bruceeckel.com]
class Test {
    public static void main(String[] args) {

        Parcel3 p = new Parcel3();
        Contents c = p.cont();
        Destination d = p.dest("Tanzania");

        // Illegal -- can't access private class:
        //! Parcel3.PContents pc = p.new PContents();
    }
}
```

The State Design Pattern Revisited

- A scientist does three very different things (modes)
 - Writes paper (and drinking coffee)
 - Teaches classes (and drinking water)
 - Administration (and drinking tea)
- The implementation of each is assumed to be very complex
- Must be able to change dynamically between these modes



Implementing of State Design Pattern

```
public class Scientist{

    private interface Mode{
        void work();
        void drink();
    }

    private class WriteMode implements Mode{
        public void work() { System.out.println("write"); }
        public void drink() { System.out.println("coffee"); }
    }

    protected class TeachMode implements Mode{
        public void work() { System.out.println("teach"); }
        public void drink() { System.out.println("water"); }
    }

    //snip
}
```


Implementing of State Design Pattern, cont.

```
public class Scientist{ // nothing has changed here
    //snip
    private Mode mode;
    public Scientist(){
        mode = new WriteMode(); /* default mode */
    }

    // what scientist does
    public void doing() { mode.work(); }
    public void drink() { mode.drink(); }

    // change mode methods
    public void setWrite()          { mode = new WriteMode(); }
    public void setTeach()          { mode = new TeachMode(); }
    public void setAdministrate() { mode = new AdmMode(); }

    public static void main(String[] args){
        Scientist einstein = new Scientist();
        einstein.doing(); einstein.setTeach();
        einstein.doing();
    }
}
```

Evaluation of State Design Pattern

- Can change modes dynamically
 - Main purpose!
- Different modes are isolated in separate classes
 - Complexity is reduced (nice side-effect)
- Client of the **Scientist** class can see the **Mode** class (and its subclasses).
 - This may unnecessarily confuse these clients. (FIXED)
- **Scientist** class *cannot* change mode added after it has been compiled, e.g., **SleepMode**. (CANNOT BE FIXED)
- Can make instances of **Mode** class. This should be prevented. (FIXED)
- The *state design pattern*
 - Nice design!

Why Inner Classes?

- Each inner class can independently inherit from other classes
 - outer class inherits from one class
 - inner class inherits from another class
- The only way to get “multiple implementation inheritance”.
 - Reuse code from more than one superclass
- Makes design more modular
- Reduces complexity
 - By encapsulation/information hiding

Summary for Introduction

- There can be more than one Java class in a file
 - 0..1 public class
 - 0..n inner classes
- Classes can be nested in Java
 - Named inner classes
- Can be used to make the internals of classes more readable

Moving-on with Inner Classes

- Accessing data in the outer classes
- A callback mechanism
 - implemented with interfaces and inner classes
- Creating instances of inner classes
- Anonymous inner classes
- The *template design pattern*

Access to Outer Data

```
public interface Diplomat{
    /** Attend a meeting */
    void attendMeeting();
    /** Talk nicely */
    void talkNice();
}

public class Embassy{
    private String[] secrets = new String[100];
    private int counter = 0;

    private class Agent implements Diplomat {
        public void attendMeeting() {
            System.out.println("Be nice");
        }
        public void talkNice() {
            System.out.println("Talk nice");
        }
        protected void tellSecrets(){
            System.out.println("Tell a secret");
            secrets[counter++] = new String("A new secret");
        }
    }
}
```

Callback Mechanism

```
public class Embassy{
    // snip

    public class Secretary implements Diplomat {
        public void attendMeeting() {
            System.out.println("Secretary.attendMeeting()");
        }
        // snip
    }

    boolean sendAgent = true;
    // send out diplomats some are secret agents (call-back)
    public Diplomat sendDiplomat(){
        if (sendAgent){
            sendAgent = false; return new Agent();
        }
        else {
            sendAgent = true; return new Secretary();
        }
    }
}
```

Callback Mechanism, cont

```
public class Embassy{
    // snip
    // get reports back from diplomats
    public void getReport(Diplomat dip) {
        if (dip instanceof Agent) {
            ((Agent)dip).tellSecrets();
        }
        else {dip.talkNice();}
    }

    public static void main() {
        Diplomat[] dips = new Diplomat[10];
        Embassy cccp = new Embassy();
        for(int i = 0; i < 10; i++){
            dips[i] = cccp.sendDiplomat();
        }
        for(int j = 0; j < 10; j++){
            cccp.getReport(dips[j]);
        }
    }
}
```


Nesting Interfaces

- Interfaces can be nested however public rule still apply!

```
public interface NestedInterface{
    private interface A{    /* not allowed compile error */
        int aData = 1;
        int a();
    }
    protected interface B{ /* not allowed compile error */
        int bData = 2;
        int b();
    }
    interface C{            /* default public */
        int cData = 3;
        int c();
    }
    public interface D{      /* explicit public */
        int dData = 4;
        int d();
    }
    int outer();            /* method on outer interface */
}
```

Nesting Interfaces, cont

- Private and protected interfaces possible in classes

```
public class NestedInterfacesInClass{
    private interface A{
        int aData = 1;
        int a();
    }
    protected interface B{
        int bData = 2;
        int b();
    }
    interface C{                /* friendly */
        int cData = 3;
        int c();
    }
    public interface D{
        int dData = 4;
        int d();
    }
}
```

Creating Instances of Inner Classes

- Cannot create inner classes without an instance of outer class is available (special cases see *nested classes* pp. 344 in the book)
- The syntax for creating instances of inner classes is awkward

```
public class A{
    public A(){ System.out.println("A"); }
    public class BB {
        public BB(){ System.out.println("BB"); }
        public class CCC{
            public CCC(){ System.out.println("CCC"); }
        }
    }

    public static void main(String[] args){
        A a          = new A();
        A.BB bb       = a.new BB();
        A.BB.CCC ccc  = bb.new CCC();
    }
}
```

Anonymous Inner Classes, Example

- When a class is only needed in one place.
- Convenient shorthand.
- Works for both interfaces and classes.

// [Source: bruceeckel.com]

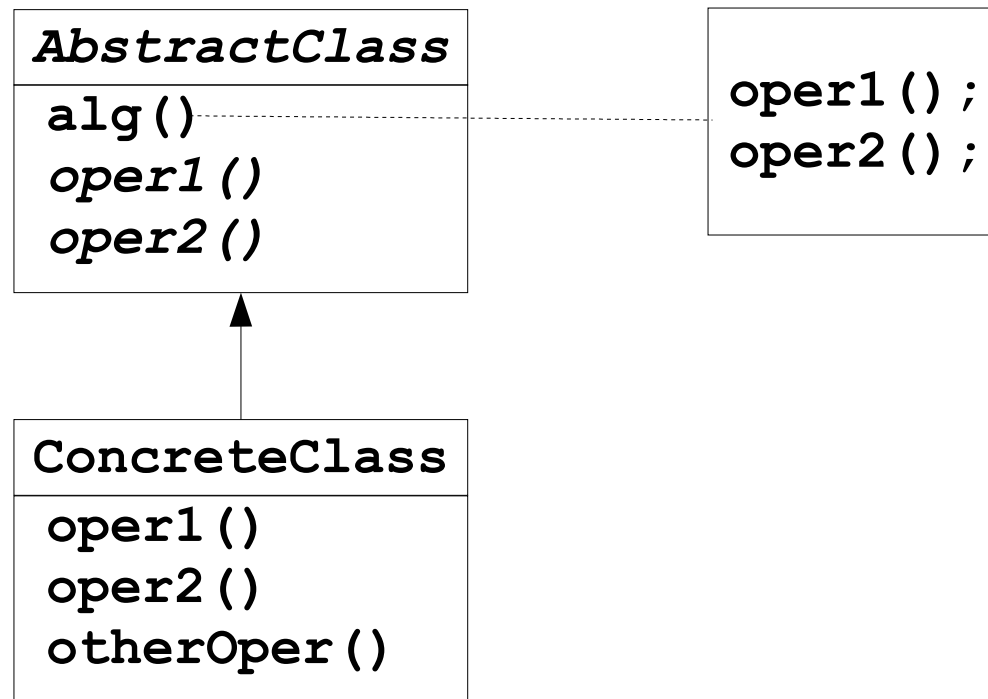
```
public class Parcel6 {  
    public Contents cont() {  
        return new Contents() {  
            private int i = 11;  
            public int value() { return i; }  
        };  
    }  
    public static void main(String[] args) {  
        Parcel6 p = new Parcel6();  
        Contents c = p.cont();  
    }  
}
```

note the
semicolon



The Template Design Pattern

- Intent: Define the skeleton for an algorithm in an operation, deferring some steps to subclasses [GOF pp 325].
- Well-suited to be implemented using inner classes (+ interfaces)
- Used extensively in many frameworks, e.g., Swing
 - “Factoring out common code”



The Template Design Pattern, cont

- A student participates in many events during a day, as examples
 - wakes up
 - eats breakfast
 - does math homework
 - goes to bed
- Each event always consists of three ordered steps
 - Prepare
 - Do the actual event
 - Clean up

```
// interface for modeling an event
public interface Event{
    public void open();
    public void doStuff();
    public void close();
}
```

The Template Design Pattern, cont

- Build the abstract class and make use of the **Event** interface

```
public abstract class Controller {
    private Event[] events;
    private int counter;
    public Controller() { // constructor on abstract class
        events = new Event[100];
        counter = 0;
    }
    /** Adds an event */
    public final void add(Event e) {events[counter++] = e;}

    /** final to make sure not overwritten in subclass */
    public final void run() {
        for(int i = 0; i <= counter - 1; i++) {
            Event e = events[i];
            e.open();
            e.doStuff();
            e.close();
        }
    }
}
```

The Template Design Pattern, cont

```
public class StudentController extends Controller{

    public StudentController(){
        super(); // calls constructor on abstract class!
    }

    public class DoMath implements Event{
        public void open() { System.out.println("open math book"); }
        public void doStuff() { System.out.println("read"); }
        public void close() { System.out.println("close math book"); }
    }

    public class WakeUp implements Event{
        public void open() { System.out.println("open eyes"); }
        public void doStuff() { System.out.println("leave bed"); }
        public void close() { System.out.println("turn-on light"); }
    }

    // similar implementations for Sleep and Eat events
}
```

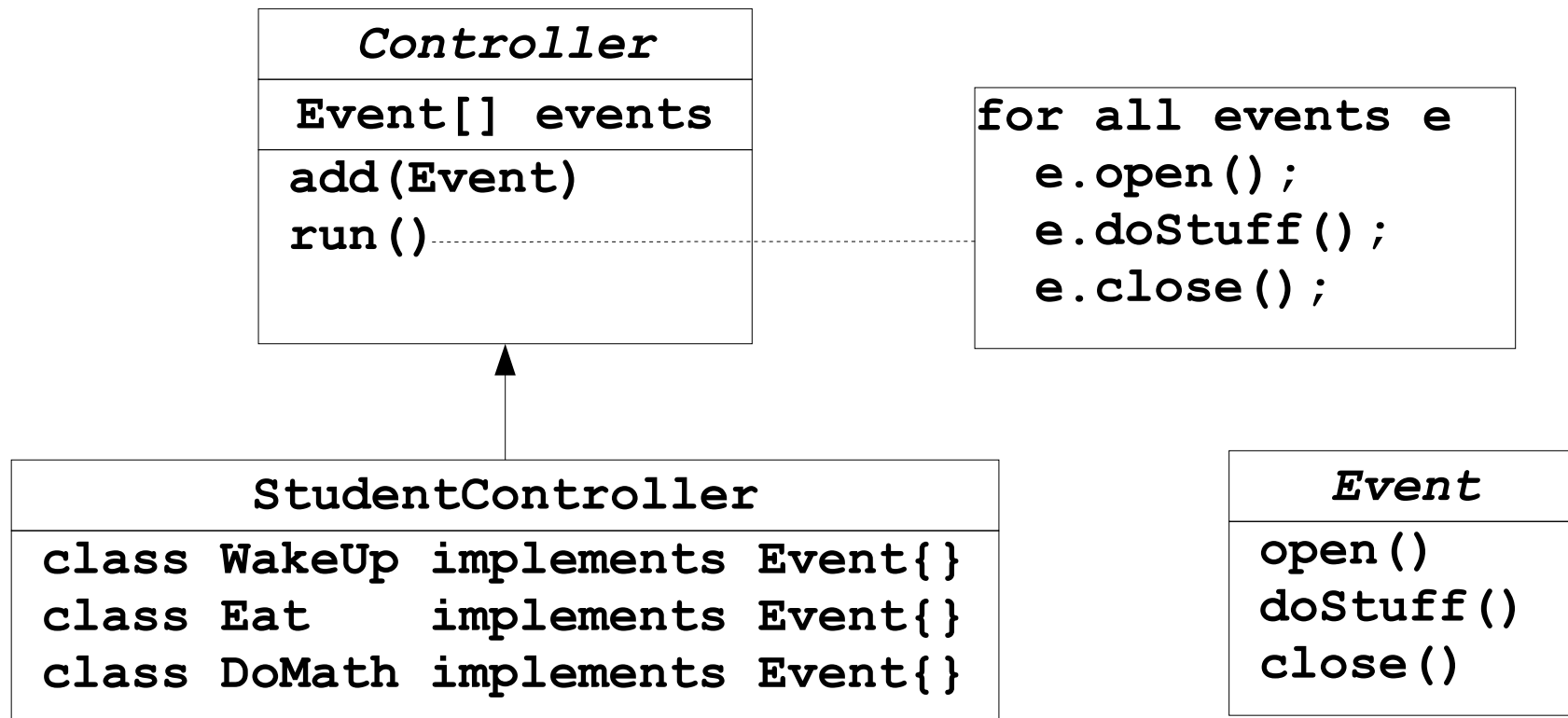

The Template Design Pattern, cont

```
public class StudentController extends Controller{
    public static void main(String[] args){
        StudentController sc = new StudentController();

        // the events for one day for a single student
        sc.add(sc.new WakeUp());
        sc.add(sc.new Eat());
        sc.add(sc.new DoMath());
        sc.add(sc.new GoToSleep());
        sc.run(); // activate template method
    }
}
```

The Template Design Pattern, cont.

- **run** method is final, i.e., cannot be redefined
- All methods used in **run** are defined in **Event** interface
- **Event** interface implemented by inner classes.



Summary

- Very many details for using inner classes
- All four access modifiers can be used on
 - inner classes
 - inner interfaces
- Anonymous inner classes
 - Do not have to invent a name for a class you only need one off
- Template design pattern shows usage of
 - interfaces
 - inner classes