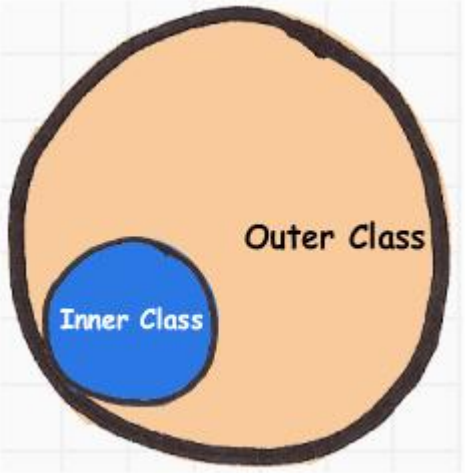
Inner Classes



Introduction

- It's possible to place a class definition within another class definition
  - This is called an inner class
- Code-hiding mechanism
  - Place classes inside other classes
- It knows about and can communicate with the surrounding class
- The kind of code you can write with inner classes is more elegant and clear

Creating inner classes

- You create an inner class just as you'd expect—by placing the class definition inside a surrounding class

```
1  public class Parcel1 {
2    class Contents {
3      private int i = 11;
4      public int value() { return i; }
5    }
6    class Destination {
7      private String label;
8      Destination(String whereTo) {
9        label = whereTo;
10     }
11     String readLabel() { return label; }
12   }
13   // Using inner classes looks just like
14   // using any other class, within Parcel1:
15   public void ship(String dest) {
16     Contents c = new Contents();
17     Destination d = new Destination(dest);
18     System.out.println(d.readLabel());
19   }
20   public static void main(String[] args) {
21     Parcel1 p = new Parcel1();
22     p.ship("Tasmania");
23   }
24 }
```

Creating inner classes (Cont.)

```
1  public class Parcel2 {
2    class Contents {
3      private int i = 11;
4      public int value() { return i; }
5    }
6    class Destination {
7      private String label;
8      Destination(String whereTo) {
9        label = whereTo;
10     }
11     String readLabel() { return label; }
12   }
13   public Destination to(String s) {
14     return new Destination(s);
15   }
16   public Contents contents() {
17     return new Contents();
18   }
19   public void ship(String dest) {
20     Contents c = contents();
21     Destination d = to(dest);
22     System.out.println(d.readLabel());
23   }
24   public static void main(String[] args) {
25     Parcel2 p = new Parcel2();
26     p.ship("Tasmania");
27     Parcel2 q = new Parcel2();
28     // Defining references to inner classes:
29     Parcel2.Contents c = q.contents();
30     Parcel2.Destination d = q.to("Borneo");
31   }
32 }
```

- **If you want to make an object of the inner class anywhere except from within a non-static method of the outer class, you must specify the type of that object as *OuterClassName.InnerClassName***

The link to the outer class

```
interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) { items = new Object[size]; }
  public void add(Object x) {
    if(next < items.length)
      items[next++] = x;
  }
  private class SequenceSelector implements Selector {
    private int i = 0;
    public boolean end() { return i == items.length; }
    public Object current() { return items[i]; }
    public void next() { if(i < items.length) i++; }
  }
  public Selector selector() {
    return new SequenceSelector();
  }
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
      System.out.print(selector.current() + " ");
      selector.next();
    }
  }
}
```

- When you create an inner class, an object of that inner class has *a link* to the *enclosing object* that made it
- It can access the members of that enclosing object
- Inner classes have access rights to all the elements in the enclosing class
- How can this happen?
  - The inner class secretly captures a reference to the particular object of the enclosing class that was responsible for creating it

Using *.this* and *.new*

- **Produce a reference to the outer-class object in an inner class**
  - **OuterClassName.*this***
- **Create an object of its inner classes, you must provide a reference to the other outer-class object in the new expression, using the *.new* syntax**

```
1  public class DotThis {
2    void f() { System.out.println("DotThis.f()"); }
3    public class Inner {
4      public DotThis outer() {
5        return DotThis.this;
6        // A plain "this" would be Inner's "this"
7      }
8    }
9    public Inner inner() { return new Inner(); }
10   public static void main(String[] args) {
11     DotThis dt = new DotThis();
12     DotThis.Inner dti = dt.inner();
13     dti.outer().f();
14   }
15 }
```

```
1  public class DotNew {
2    public class Inner {}
3    public static void main(String[] args) {
4      DotNew dn = new DotNew();
5      DotNew.Inner dni = dn.new Inner();
6    }
7  } ///:~
```

Using *.this* and *.new* (Cont.)

- **It's not possible to create an object of the inner class unless you already have an object of the outer class**
  - **This is because the object of the inner class is quietly connected to the object of the outer class that it was made from**

```
1  public class Parcel3 {
2    class Contents {
3      private int i = 11;
4      public int value() { return i; }
5    }
6    class Destination {
7      private String label;
8      Destination(String whereTo) { label = whereTo; }
9      String readLabel() { return label; }
10   }
11   public static void main(String[] args) {
12     Parcel3 p = new Parcel3();
13     // Must use instance of outer class
14     // to create an instance of the inner class:
15     Parcel3.Contents c = p.new Contents();
16     Parcel3.Destination d = p.new Destination("Tasmania");
17   }
18 } ///:~
```

Inner classes and upcasting

- Inner classes really come into their own when you start upcasting to a base class or an interface
- That's because the inner class—the implementation of the interface—can then be unseen and unavailable
  - Convenient for hiding the implementation

```
1  public interface Destination {
2    String readLabel();
3  } ///:~
```

```
1  public interface Contents {
2    int value();
3  } ///:~
```

```
1  class Parcel4 {
2    private class PContents implements Contents {
3      private int i = 11;
4      public int value() { return i; }
5    }
6    protected class PDestination implements Destination {
7      private String label;
8      private PDestination(String whereTo) {
9        label = whereTo;
10     }
11     public String readLabel() { return label; }
12   }
13   public Destination destination(String s) {
14     return new PDestination(s);
15   }
16   public Contents contents() {
17     return new PContents();
18   }
19 }
20
21 public class TestParcel {
22   public static void main(String[] args) {
23     Parcel4 p = new Parcel4();
24     Contents c = p.contents();
25     Destination d = p.destination("Tasmania");
26     // Illegal -- can't access private class:
27     //! Parcel4.PContents pc = p.new PContents();
28   }
29 } ///:~
```

Inner classes in methods and scopes

- **Inner classes can be created within a method or even an arbitrary scope**
  - **1. As shown previously, you're implementing an interface of some kind so that you can create and return a reference.**
  - **2. You're solving a complicated problem and you want to create a class to aid in your solution, but you don't want it publicly available.**
- **The creation of an entire class within the scope of a method. This is called a *local inner class***

```
1  public class Parcel5 {
2    public Destination destination(String s) {
3      class PDestination implements Destination {
4        private String label;
5        private PDestination(String whereTo) {
6          label = whereTo;
7        }
8        public String readLabel() { return label; }
9      }
10     return new PDestination(s);
11   }
12   public static void main(String[] args) {
13     Parcel5 p = new Parcel5();
14     Destination d = p.destination("Tasmania");
15   }
16 } ///:~
```

Anonymous inner classes

```
1  public class Parcel7 {
2    public Contents contents() {
3      return new Contents() { // Insert a class definition
4        private int i = 11;
5        public int value() { return i; }
6      }; // Semicolon required in this case
7    }
8    public static void main(String[] args) {
9      Parcel7 p = new Parcel7();
10     Contents c = p.contents();
11   }
12 } ///:~
```

```
1  public interface Contents {
2    int value();
3  } ///:~
```

- The class is anonymous -- it has no name
- The anonymous inner-class syntax is a shorthand for

```
1  public class Parcel7b {
2    class MyContents implements Contents {
3      private int i = 11;
4      public int value() { return i; }
5    }
6    public Contents contents() { return new MyContents(); }
7    public static void main(String[] args) {
8      Parcel7b p = new Parcel7b();
9      Contents c = p.contents();
10   }
11 } ///:~
```

Anonymous inner classes (Cont.)

- **The following code shows what to do if your base class needs a constructor with an argument:**

```
1  public class Parcel8 {
2    public Wrapping wrapping(int x) {
3      // Base constructor call:
4      return new Wrapping(x) { // Pass constructor argument.
5        public int value() {
6          return super.value() * 47;
7        }
8      }; // Semicolon required
9    }
10   public static void main(String[] args) {
11     Parcel8 p = new Parcel8();
12     Wrapping w = p.wrapping(10);
13   }
14 } ///:~
```

```
1  public class Wrapping {
2    private int i;
3    public Wrapping(int x) { i = x; }
4    public int value() { return i; }
5  } ///:~
```

Anonymous inner classes (Cont.)

- You can also perform initialization when you define fields in an anonymous class:

```
1  public class Parcel9 {
2    // Argument must be final to use inside
3    // anonymous inner class:
4    public Destination destination(final String dest) {
5      return new Destination() {
6        private String label = dest;
7        public String readLabel() { return label; }
8      };
9    }
10   public static void main(String[] args) {
11     Parcel9 p = new Parcel9();
12     Destination d = p.destination("Tasmania");
13   }
14 } ///:~
```

```
1  public interface Destination {
2    String readLabel();
3  } ///:~
```

- If you're defining an anonymous inner class and want to use an object that's defined outside the anonymous inner class, the compiler requires that the argument reference be ***final***

Anonymous inner classes (Cont.)

- **What if you need to perform some constructor-like activity?**
  - **You can't have a named constructor in an anonymous class (since there's no name!)**
  - **Use instance initialization**

```
1  import static net.mindview.util.Print.*;
2
3  abstract class Base {
4    public Base(int i) {
5      print("Base constructor, i = " + i);
6    }
7    public abstract void f();
8  }
9
10 public class AnonymousConstructor {
11   public static Base getBase(int i) {
12     return new Base(i) {
13       { print("Inside instance initializer"); }
14       public void f() {
15         print("In anonymous f()");
16       }
17     };
18   }
19   public static void main(String[] args) {
20     Base base = getBase(47);
21     base.f();
22   }
23 }
```

Anonymous inner classes (Cont.)

- An *instance initializer* is the constructor for an anonymous inner class

```
1  public interface Destination {
2    String readLabel();
3  } ///:~
```

```
1  public class Parcel10 {
2    public Destination
3    destination(final String dest, final float price) {
4      return new Destination() {
5        private int cost;
6        // Instance initialization for each object:
7        {
8          cost = Math.round(price);
9          if(cost > 100)
10           System.out.println("Over budget!");
11       }
12       private String label = dest;
13       public String readLabel() { return label; }
14     };
15   }
16   public static void main(String[] args) {
17     Parcel10 p = new Parcel10();
18     Destination d = p.destination("Tasmania", 101.395F);
19   }
20 }
```

Factory Method revisited

- **Look at how much nicer the Factories.java example comes out when you use anonymous inner classes**

```
1  import static net.mindview.util.Print.*;
2
3  interface Service {
4    void method1();
5    void method2();
6  }
7
8  interface ServiceFactory {
9    Service getService();
10 }
11
12 class Implementation1 implements Service {
13   private Implementation1() {}
14   public void method1() {print("Implementation1 method1");}
15   public void method2() {print("Implementation1 method2");}
16   public static ServiceFactory factory =
17     new ServiceFactory() {
18       public Service getService() {
19         return new Implementation1();
20       }
21     };
22 }
23
```

```
24 class Implementation2 implements Service {
25   private Implementation2() {}
26   public void method1() {print("Implementation2 method1");}
27   public void method2() {print("Implementation2 method2");}
28   public static ServiceFactory factory =
29     new ServiceFactory() {
30       public Service getService() {
31         return new Implementation2();
32       }
33     };
34 }
35
36 public class Factories {
37   public static void serviceConsumer(ServiceFactory fact) {
38     Service s = fact.getService();
39     s.method1();
40     s.method2();
41   }
42   public static void main(String[] args) {
43     serviceConsumer(Implementation1.factory);
44     // Implementations are completely interchangeable:
45     serviceConsumer(Implementation2.factory);
46   }
47 }
```

Nested classes

- **If you don't need a connection between the inner-class object and the outerclass object, then you can make the inner class *static --- nested class***
  - **You don't need an outer-class object in order to create an object of a nested class**
  - **You can't access a non-static outer-class object from an object of a nested class**
- **Ordinary inner classes cannot have *static* data, *static* fields, or *nested classes***
- **Nested classes can have all of these**

Nested classes (Cont.)

- A nested class does not have a special *this* reference, which makes it analogous to a *static* method

```
1  public class Parcel11 {
2    private static class ParcelContents implements Contents {
3      private int i = 11;
4      public int value() { return i; }
5    }
6    protected static class ParcelDestination
7    implements Destination {
8      private String label;
9      private ParcelDestination(String whereTo) {
10       label = whereTo;
11     }
12     public String readLabel() { return label; }
13     // Nested classes can contain other static elements:
14     public static void f() {}
15     static int x = 10;
16     static class AnotherLevel {
17       public static void f() {}
18       static int x = 10;
19     }
20   }
21   public static Destination destination(String s) {
22     return new ParcelDestination(s);
23   }
24   public static Contents contents() {
25     return new ParcelContents();
26   }
27   public static void main(String[] args) {
28     Contents c = contents();
29     Destination d = destination("Tasmania");
30   }
31 } ///:~
```

```
1  public interface Destination {
2    String readLabel();
3  } ///:~
```

```
1  public interface Contents {
2    int value();
3  } ///:~
```

Classes inside interfaces

- Normally, you can't put any code inside an interface, but a ***nested class*** can be part of an interface
- Any class you put inside an interface is automatically ***public*** and ***static***
- It's convenient to nest a class inside an interface when you want to create some common code to be used with all different implementations of that interface

```
1  public interface ClassInInterface {
2    void howdy();
3    class Test implements ClassInInterface {
4      public void howdy() {
5        System.out.println("Howdy!");
6      }
7      public static void main(String[] args) {
8        new Test().howdy();
9      }
10   }
11 }
```

Why inner classes?

- **Why did the Java designers go to so much trouble to add this fundamental language feature?**

- **The inner class inherits from a class or implements an interface, and the code in the inner class manipulates the outer-class object that it was created within**
  - **So you could say that an inner class provides a kind of window into the outer class**

- ***Each inner class can independently inherit from an implementation. Thus, the inner class is not limited by whether the outer class is already inheriting from an implementation***

- ***Inner classes effectively allow you to inherit from more than one non-interface***

Why inner classes? (Cont.)

- **Consider a situation in which you have two interfaces that must somehow be implemented within a class**
  - **Because of the flexibility of interfaces, you have two choices: a single class or an inner class**

```
1  package innerclasses;
2
3  interface A {}
4  interface B {}
5
6  class X implements A, B {}
7
8  class Y implements A {
9    B makeB() {
10     // Anonymous inner class:
11     return new B() {};
12   }
13 }
14
15 public class MultiInterfaces {
16   static void takesA(A a) {}
17   static void takesB(B b) {}
18   public static void main(String[] args) {
19     X x = new X();
20     Y y = new Y();
21     takesA(x);
22     takesA(y);
23     takesB(x);
24     takesB(y.makeB());
25   }
26 } ///:~
```

Why inner classes? (Cont.)

- **if you have *abstract* or concrete classes instead of interfaces, you are suddenly limited to using inner classes if your class must somehow implement both of the others:**
  - ***Solve the "multiple implementation inheritance" problem***

```
1  package innerclasses;
2
3  class D {}
4  abstract class E {}
5
6  class Z extends D {
7    E makeE() { return new E() {}; }
8  }
9
10 public class MultiImplementation {
11   static void takesD(D d) {}
12   static void takesE(E e) {}
13   public static void main(String[] args) {
14     Z z = new Z();
15     takesD(z);
16     takesE(z.makeE());
17   }
18 } ///:~
```

Why inner classes? (Cont.)

- **If you didn’t need to solve the "multiple implementation inheritance" problem, you could conceivably code around everything else without the need for inner classes**

- **With inner classes you have these additional features:**
  - ***1.The inner class can have multiple instances, each with its own state information that is independent of the information in the outer-class object***
  - ***2.In a single outer class you can have several inner classes, each of which implements the same interface or inherits from the same class in a different way***
  - ***3.The point of creation of the inner-class object is not tied to the creation of the outer-class object***
  - ***4.There is no potentially confusing "is-a" relationship with the inner class; it’s a separate entity***

Inheriting from inner classes

- The inner-class constructor must attach to a reference of the enclosing class object, things are slightly complicated when you inherit from an inner class
- The problem is that the "secret" reference to the enclosing class object must be initialized, and yet in the derived class there's no longer a default object to attach to
  - EnclosingClassReference.super();

```
1  class WithInner {
2    class Inner {}
3  }
4
5  public class InheritInner extends WithInner.Inner {
6    //! InheritInner() {} // Won't compile
7    InheritInner(WithInner wi) {
8      wi.super();
9    }
10   public static void main(String[] args) {
11     WithInner wi = new WithInner();
12     InheritInner ii = new InheritInner(wi);
13   }
14 } ///:~
```

Can inner classes be overridden?

- **What happens when you create an inner class, then inherit from the enclosing class and redefine the inner class?**
  - **This example shows that there isn't any extra inner-class magic going on when you inherit from the outer class. The two inner classes are completely separate entities, each in its own namespace.**

```
1  import static net.mindview.util.Print.*;
2
3  class Egg {
4    private Yolk y;
5    protected class Yolk {
6      public Yolk() { print("Egg.Yolk()"); }
7    }
8    public Egg() {
9      print("New Egg()");
10     y = new Yolk();
11   }
12 }
13
14 public class BigEgg extends Egg {
15   public class Yolk {
16     public Yolk() { print("BigEgg.Yolk()"); }
17   }
18   public static void main(String[] args) {
19     new BigEgg();
20   }
21 }
```

```
/* Output:
New Egg()
Egg.Yolk()
*///:~
```

Can inner classes be overridden? (Cont.)

- It's still possible to explicitly inherit from the inner class

```
1  import static net.mindview.util.Print.*;
2
3  class Egg2 {
4    protected class Yolk {
5      public Yolk() { print("Egg2.Yolk()"); }
6      public void f() { print("Egg2.Yolk.f()");}
7    }
8    private Yolk y = new Yolk();
9    public Egg2() { print("New Egg2()"); }
10   public void insertYolk(Yolk yy) { y = yy; }
11   public void g() { y.f(); }
12 }
13
14 public class BigEgg2 extends Egg2 {
15   public class Yolk extends Egg2.Yolk {
16     public Yolk() { print("BigEgg2.Yolk()"); }
17     public void f() { print("BigEgg2.Yolk.f()"); }
18   }
19   public BigEgg2() { insertYolk(new Yolk()); }
20   public static void main(String[] args) {
21     Egg2 e2 = new BigEgg2();
22     e2.g();
23   }
24 }
```

```
/* Output:
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
*///:~
```

Local inner classes

- **Inner classes can also be created inside code blocks, typically inside the body of a method**
- **A local inner class *cannot* have an *access specifier* because it isn't part of the outer class**
- **It does have access to the final variables in the current code block and all the members of the enclosing class**

Local inner classes (Cont.)

- **The only justification for using a local inner class instead of an anonymous inner class**
  - **If you need a named constructor and/or an overloaded constructor, since an anonymous inner class can only use instance initialization**
  - **Another reason to make a local inner class rather than an anonymous inner class is if you need to make more than one object of that class**

```
1  import static net.mindview.util.Print.*;
2
3  interface Counter {
4    int next();
5  }
6
7  public class LocalInnerClass {
8    private int count = 0;
9    Counter getCounter(final String name) {
10     // A local inner class:
11     class LocalCounter implements Counter {
12       public LocalCounter() {
13         // Local inner class can have a constructor
14         print("LocalCounter()");
15       }
16       public int next() {
17         printnb(name); // Access local final
18         return count++;
19       }
20     }
21     return new LocalCounter();
22   }
```

```
23   // The same thing with an anonymous inner class:
24   Counter getCounter2(final String name) {
25     return new Counter() {
26       // Anonymous inner class cannot have a named
27       // constructor, only an instance initializer:
28       {
29         print("Counter()");
30       }
31       public int next() {
32         printnb(name); // Access local final
33         return count++;
34       }
35     };
36   }
37   public static void main(String[] args) {
38     LocalInnerClass lic = new LocalInnerClass();
39     Counter
40       c1 = lic.getCounter("Local inner "),
41       c2 = lic.getCounter2("Anonymous inner ");
42     for(int i = 0; i < 5; i++)
43       print(c1.next());
44     for(int i = 0; i < 5; i++)
45       print(c2.next());
46   }
47 }
```

Inner-class identifiers

- Every class produces a .class file that holds all the information about how to create objects of this type
- Inner classes must also produce .class files to contain the information for their Class objects
- The name of the enclosing class, followed by a '$', followed by the name of the inner class
  - If inner classes are ***anonymous***, the compiler simply starts generating numbers as inner-class identifiers
  - If inner classes are ***nested within inner classes***, their names are simply appended after a '$' and the outer-class identifier (s)
- For example, the .class files created by LocalInnerClass.java include:

```
Counter.class
LocalInnerClass$1.class
LocalInnerClass$1LocalCounter.class
LocalInnerClass.class
```

Thank you

zhenling@seu.edu.cn