Polymorphism Your Object

Vittayasak Rujivorakul

Lecture 7

Polymorphism Your Object

Vittayasak Rujivorakul

Lecture 7

Polymorphism

- 1 ใน 3 ขององค์ประกอบที่สำคัญของ O-O programming language
- ทำให้สามารถแยกส่วนของ interface จาก implementation
- ทำให้การจัดและอ่าน code ง่ายขึ้น
- ทำให้การขยายต่อไม่เพียงแต่ในระหว่างการพัฒนาแต่รวมไปถึงการเพิ่มในภายหลังเป็นไปได้ง่ายขึ้น
  - เพื่อเลี่ยงการต้องเขียน method ในลักษณะที่เป็น type-specific ซึ่งทำให้การขยาย method หรือสร้าง derived object ชนิดใหม่ทำได้ง่าย

Inheritance & Upcasting

- ในการ upcasting จาก derived object ไปเป็น base object จะยังรักษา interface ของ base class ไว้

```
class Instrument {
   public void play( int n ){
      System.out.println("Instrument.play( )");
   }
}
```

```
class Wind extends Instrument {
   plublic void play( int n ) {
      System.out.println("Wind.play( )");
   }
}
```

```
public class Music {
   public static void tune( Instument i ) {
      i.play(0);
   }
   public static void main(String args[ ]) {
      Wind flute = new Wind( );
      tune(flute);
   }
}
```


Method call binding

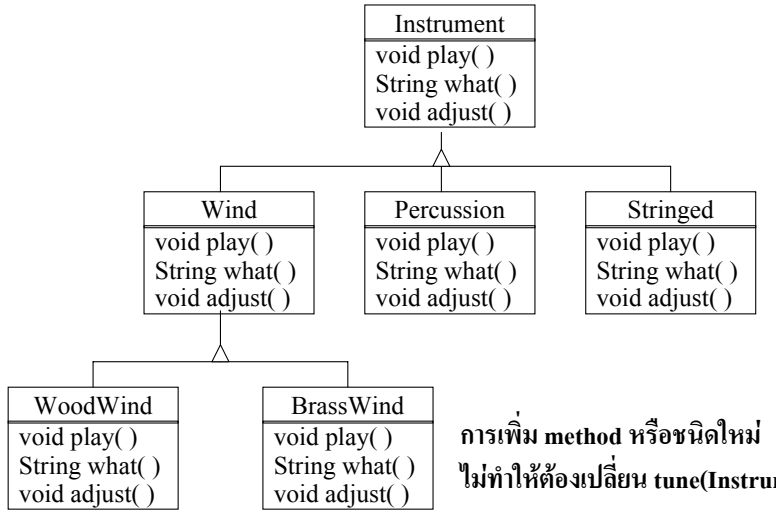
- Binding = การเชื่อม Method call กับ method body
  - early binding : ก่อนที่จะมีการ run compile ทำโดย compiler & linker
  - late binding : เกิดขึ้นในช่วง run-time ซึ่งต้องมีกลไกในการที่จะบอกว่า type ของ object เป็นอะไรในช่วง run-time
- ทุก method binding ในจาวาเป็น late binding ยกเว้น method นั้นจะกำหนดไว้เป็น final
- ดังนั้นเมื่อส่ง message ไปยัง object, late-binding จะทำการเชื่อมโยง method ที่ถูกต้องกับ object นั้นเพื่อทำงานให้เหมาะสมเอง

ตัวอย่าง Late-Binding

```
class Shape {
   void draw( ){ }
   void erase( ) { }
}
class Square extends Shape {
   void draw( ) {
      System.out.println("Square.draw( )");
}
}
class Circle extends Shape {
   void draw( ) {
      System.out.println("Circle.draw( )");
   }
}
```

```
public class Shapes {
   public static Shape randShape( ) {
      switch((int)(Math.random( )*2)) {
         default:
         case 0: return new Circle( );
         case 1: return new Square( );
      }
   }
   public static void main(String a[ ])
   {
       Shape s[] = new Shape[4];
       for (int i=0; i<s.length; i++)
         s[i] = randShape( );
       for (int i=0; i<s.length; i++)
         s[i].draw( );
   }
}
```


Extensibility

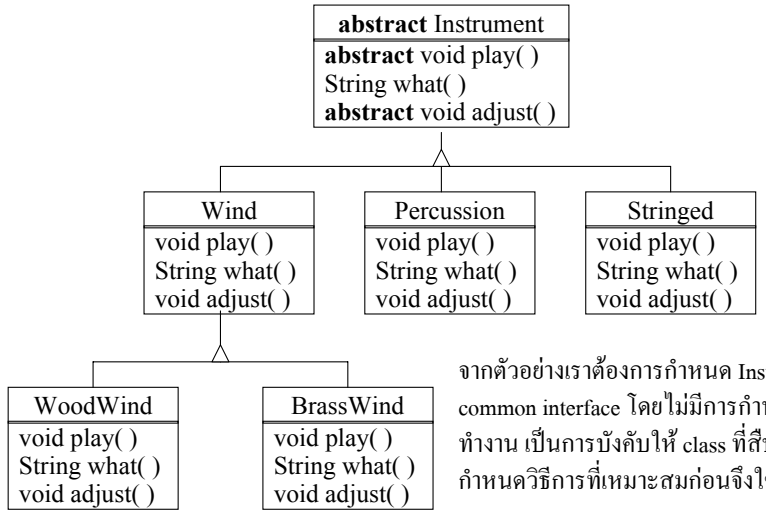


การเพิ่ม method หรือชนิดใหม่ ไม่ทำให้ต้องเปลี่ยน tune(Instrument i) method

Overriding VS Overloading

- การกำหนด method body ใหม่ใน derived class เป็นการทำ overriding
- การเปลี่ยน interface ของ method ใด ๆ เป็นการทำ Overloading
- ดังนั้น ในการเขียนโปรแกรมต้องระวังว่า ต้องการทำ overriding หรือ overloading เพราะ compiler อนุญาตทั้ง 2 วิธีการ การโปรแกรมผิดพลาดในลักษณะทำให้พฤติกรรมของ method อาจต่างกันไป และค้นหาที่ผิดได้ยาก

Abstract classes and Methods

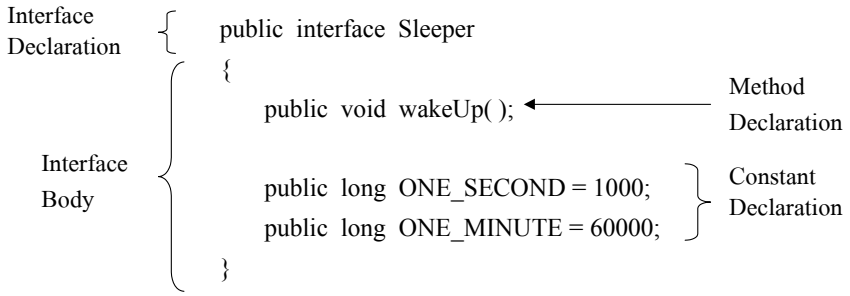


จากตัวอย่างเราต้องการกำหนด Instrument ให้มีเพียง common interface โดยไม่มีการกำหนดขั้นตอนการทำงาน เป็นการบังคับให้ class ที่สืบทอดต่อต้องกำหนดวิธีการที่เหมาะสมก่อนจึงใช้สร้าง object ได้

Interfaces

- คิดได้คล้ายกันกับการทำให้เป็น “pure” abstract class เพื่อใช้สร้าง protocol ระหว่าง classes
- Interface ในจาวา คือ ชุดของชื่อของ methods โดยไม่มีการ implement ซึ่งรวมถึงการประกาศค่าคงที่ในการใช้งานด้วย
  - กำหนดรูปแบบของ class โดยมี method names, argument lists และ return types โดยไม่มี method body
- Interface นั้นไม่ใช่ abstract class
  - abstract class ไม่สามารถสร้าง object จาก class นั้นได้ต้อง inherit

การสร้าง interface



Syntax :

<modifier> **interface** <name> { / * body * / }

- Methods ใน interface นั้น implicitly เป็น “public” และ “abstract”
- ไม่นิยมประกาศ interface หรือ method ใน interface ด้วย modifier “abstract”
- Constant ใน interface นั้น implicitly เป็น “public”, “static” และ “final”

การใช้ interface

- โดยการ implement interface นั้น : implements keyword

```
public class MySleeper implements Sleeper  {
    . . .
    Public void wakeUp ( )  {
        . . .
    }
}
```

  - ต้องทำการ implement ทุก method ที่กำหนดไว้ใน interface
- สามารถใช้ interface เหมือนกับ Type
  - Sleeper aSleeper = new aSleeper ( ) ;

การเลือกใช้การทำ Interface

- เพื่อใช้รวบรวมสิ่งที่เหมือนกันของ classes ที่ไม่เกี่ยวข้องกันโดยที่ไม่ต้องบังคับให้ต้องสร้างความสัมพันธ์ระหว่าง classes นั้น
- เป็นการประกาศ methods ซึ่งคาดว่าจะต้องมี class บาง class ทำการ implement
- เปิดเผย interface โดยที่ไม่ต้องเปิดเผย class (Objects ในลักษณะนี้เรียกว่า anonymous object ซึ่งเป็นประโยชน์เมื่อเวลาส่งมอบ packages ไปยังนักพัฒนาอื่น)

“Multiple Inheritance” ใน Java

- Multiple inheritance คือการสืบทอดจากหลาย base classes (e.g. ล่อ สืบทอดจากม้า และ ลา)
- จาวาอนุญาตให้ทำการสืบทอดได้จากเพียง 1 class (Single inheritance) = = > จาวาไม่ support multiple class inheritance
- จาวาอนุญาตให้ทำการสืบทอดจากหลาย interfaces ได้ = = > จาวา support multiple interface inheritance

Interface VS Multiple Class Inheritance

- Class จะสืบทอดเพียงค่า constants จาก interface
- class ไม่สามารถสืบทอด method implementations จาก interface
- interface hierarchy นั้นไม่ขึ้นกับ class hierarchy
  - Classes ซึ่ง implement interface เดียวกัน อาจจะหรืออาจจะไม่สัมพันธ์กันในระดับ class hierarchy ซึ่งผิดจากแนวคิดของ multiple inheritance
- นั่นคือ จาวายอมให้มี multiple interface inheritance แต่ไม่สนับสนุน multiple class inheritance

การขยาย interface โดยการสืบทอด

- Interface สามารถทำการสืบทอดได้ และ สามารถทำการรวมหลาย interfaces เพื่อสร้างเป็น interface ใหม่ได้
- การสืบทอดของ interface ใช้ extends keyword
- Interface สามารถทำการสืบทอดได้จาก multiple base interfaces

```
interface Monster { void menace ( ) ; }
interface DangerousMonster extends Monster { void destroy ( ) ; }
interface Lethal { void kill ( ) ; }
interface Vampire extends DangerousMonster, Lethal {
        void drinkBlood ( ) ;
}
```


Grouping Constants

- อาจใช้ interface ในการสร้างกลุ่มของค่าคงที่ (คล้ายกับ enum ใน C) เนื่องจาก fields ใน interface จะเป็น public, static และ final โดยอัตโนมัติ

```
public interface Months {
    int
    JAN = 1 , FEB = 2 , MAR = 3 , APR = 4 , MAY = 5 ,
    JUN = 6 , JUL = 7 , AUG = 8 , SEP = 9 , OCT = 10 ,
    NOV = 11 , DEC = 12 ;
}
```

Initializing fields ใน interface

- การกำหนดค่าเริ่มต้นของ fields ใน interface ไม่สามารถเป็น blank finals แต่สามารถทำด้วย non-constant expression ได้

```
public interface RandVals  {
      int  rint  =  (int)  (Math. random ( )  *  10) ;
}
public class TestRandVals  {
      public static void main (String args [ ] )  {
            System. out. println (RandVals. Rint) ;
      }
}
```

- fields นั้นจะถูกกำหนดค่าเริ่มต้นเมื่อ class ถูก loaded และถูกเก็บค่าในส่วนของ static storage area

Inner classes

- Inner Class คือ class ที่ระบุอยู่ภายใน class อื่น โดยที่สามารถอ้างถึง data member ของ object ที่คลอบมันอยู่ได้
- เพื่อใช้ช่วยในการทำ implementation hiding

```
public class Percel  {
      class Contents  {
            private  int  j  =  10 ;
            public  void  value ( )  { return  j ;  }
      }
      public  void  ship  ( )  {
            Contents  c  =  new  Contents  ( ) ;
      }
}
```


Inner classes in methods and scopes

- Inner class สามารถสร้างภายใน method หรือ scope ใด scope หนึ่ง
  - เพื่อสร้าง interface บางอย่างเพื่อการสร้างหรือคืน object
  - เพื่อช่วยแก้ปัญหาที่ซับซ้อน โดยการสร้าง class ภายในที่ซ่อนจากภายนอกมาช่วย
- ตัวอย่าง
  - class ที่กำหนดภายใน method
  - class ที่กำหนดใน scope ภายใน method
  - anonymous class ที่ใช้ implement interface
  - anonymous class ที่ extends จาก class ที่ไม่มี default constructor
  - anonymous class ที่ใช้เพื่อการทำ construction โดยกำหนดค่าเริ่มต้นของ instance

ชนิดของ Inner Classes

- Nested top-level classes & interfaces
  - Class หรือ interface ที่ถูกกำหนดให้เป็น static member ภายใน class หรือ interface อื่นได้โดย static modifier หรือ nested interface ที่โดยนัยเป็น static อยู่แล้ว
  - ลักษณะคล้ายกับ class ปกติซึ่งอยู่ใน package เดียวกัน
  - อ้างถึง inner class หรือ interface โดยเรียก class นอก ตามด้วยจุด และชื่อของ class ใน (I.e., LinkedList.Linkable)
  - สามารถใช้การ import outer class ได้เช่นเดียวกับการ import package
  - โดยทั่วไปใช้เพื่อจัดกลุ่มของ class ที่เกี่ยวข้องกันไว้ด้วยกัน เพื่อให้เกิด ความสะดวก

Nested Top-Level Class

```
public class TestBed  {
    public TestBed  ( )   { }
    void  print ( )   {
          System. Out. Println (“print ( ) “) ;
    }
    / / static inner class
    / / generated as a separated class called TestBed$Tester
    public static class Tester   {
          public static void main (String args [ ] )   {
                TestBed  t  =  new  TestBed ( )  ;
                t. print ( ) ;
          }
    }
```


ชนิดของ Inner Classes (ต่อ)

- Member classes
  - class ที่ถูกกำหนดให้เป็น member ภายใต้ class อื่น โดยที่ไม่ได้กำหนดให้เป็น static
  - class ภายในสามารถที่จะ access members ของ class ภายนอกได้ รวมทั้ง member ที่ประกาศไว้เป็น private ด้วย
  - inner class object จะสัมพันธ์กับ outer class object ที่ครอบมัน
  - ใช้ในกรณีที่ต้องการให้ inner class instance สามารถ access fields ของ instance ของ class ที่ครอบมัน

Member Class

```
public class A {
   public String name = “A”;
   public class B {
      public String name = “B”;
      public class C {
         public String name = “C”;
         public void print_names( ) {
            // C - name of C
            System.out.println(name);
            // C - name of C
            System.out.println(this.name);
           // B - name of B
           System.out.println(B.this.name);
           // A - name of A
           System.out.println(A.this.name);
         }
      }
   }
   public static void main (String args[ ] ) {
      A a = new A();
      // create an A instance
      A.B  b = a.new B();
      // create a B instance
      A.B.C c = b.new C();
      // create a C instance
      c.print_names( );
   }
}
```


ชนิดของ Inner Classes (ต่อ)

- Local Classes
  - inner class ที่ถูกกำหนดภายใต้ block ของ code
  - สามารถถูกเห็นได้เพียงภายใต้ block code นั้น
  - inner class นั้นสามารถที่จะ access members ของ class ที่ครอบมันได้
  - นอกจากนี้ยังสามารถที่จะ access final variables หรือ parameters ที่อยู่ใน scope ของ code block นั้น
  - ใช้เพื่อเป็น “adapter class” หรือใช้ในการจัดการ event (event-handling)

Local Class

```
class X { protected char x = 'x'; }
class Y { protected char y = 'y'; }
class Z extends X {
   private char z = 'z';
   public static char d = 'd';
   public void createLocalObject(final char e){
     final char f = 'f';
     int i = 0;
     class Local extends Y {
        char g = 'g';
        public void printVars( ) {
          // all of these fields &
          // var are accessible
          // to this class this.g
          System.out.println(g);
          // f -- final local
          System.out.println(f);
          // e -- final local
          System.out.println(e);
          // Z.this.d
          System.out.println(d);
          // Z.this.z
          System.out.println(z);
          // b (inherited by this class)
          System.out.println(y);
          // a inherited by the container
          System.out.println(x);
        }
     }
     Local l = this.new Local( );
     l.printVars( );
   }
   public static void main(String args[ ]){
     C c = new C( );
     c.createLocalObject('e');
   }
}
```


ชนิดของ Inner Classes (ต่อ)

- Anonymous classes
  - เป็น class ที่ไม่มีชื่อ
  - ทำการรวมส่วนของการประกาศ และ การ instantiate ไว้ในขั้นตอนเดียว
  - ดังนั้นจึงสามารถถูก instantiate ได้เพียงครั้งเดียว
  - ไม่สามารถ extends หรือ implements class หรือ interface อื่นได้
  - ไม่สามารถกำหนด constructor ให้มันได้
  - มีลักษณะคล้ายกับ local class ทั้งในแง่พฤติกรรมและการใช้งาน

Anonymous Class

```
import java.io. * ;
public class Lister   {
    public static void main (String args [ ] )   {
          File  f = new File (args [0]) ;
          String [ ]  list  =  f.list (new FilenameFilterr ( )    {
                    public boolean accept (File F, String s)    {
                              return s. endWith (".java") ;
                    }
          }) ;
          for (int  i = 0 ;   i < list.length ;   i + +
                    System. out. println (list [i]) ;
    }
```


Anonymous inner classes

- Anonymous classes เป็นรูปแบบอย่างง่ายของ inner class ที่ไม่มีการระบุชื่อ แต่ใช้การ implementation ควบคู่พร้อมกับการสร้าง (new method สำหรับสร้าง object)
  - เพื่อความสะดวกและคล่องตัวในการเขียน class หรือ interface ที่ง่าย ๆ และใช้เพียงครั้งเดียวโดยไม่ต้องสร้าง class ใหม่

```
frame.addWindowListerner (new WindowAdapter ( )  [
      public void windowClosing (WindowEvent e)   {
                frame.dispose ( ) ;
                System.exit (0) ;
      }
} ) ;
```

Inner class กับการใช้จาก outer class

- Outer class สามารถใช้ inner class ได้เหมือนกับ class อื่นทั่วไป
- ในกรณีที่สร้าง object ของ inner class ใน static method ต้องระบุ OuterClassName.InnerClassName

```
public class Percel  {
      class Contents  {
            private  int  j  =  10 ;
            public void value ( )  { return  j ;  }
      }
      public static  void main (String args [ ] )  {
            Percel p  =  new  Percel ( ) ;
            Percel.Contents  c  =  p.new Contents ( ) ;
      }
}
```

- สามารถกำหนด access specifier ให้เป็น private, protected class ได้

Static Inner Classes

- ในบางกรณีที่ต้องกำหนดให้ inner class เป็น static (เช่น outer class ที่ method เป็น static ทำให้เราต้องกำหนด inner class เป็น static), static inner class มีลักษณะ
  - ไม่จำเป็นต้องสร้าง outer object เพื่อที่จะสร้าง static inner class
  - ไม่สามารถจะ access outer object จาก object ของ static inner class
- สามารถที่จะบรรจุ static inner class ใน interface ได้

Inheriting from inner classes

- การสืบทอดจาก inner class นั้นต้องทำ explicit association

```
class WithInner  {
      class Inner  { }
}
public class InheritInner extends  WithInner.Inner  {
      / / InheritInner ( )  { }  / / ไม่สมบูรณ์
      InheritInner (WithInner w)  {
            w.super ( ) ;
      }
      :
}
```


Inner class identifiers

- เมื่อ compile แล้ว inner class จะมีชื่อประกอบด้วย
  - ชื่อของ outer class ตามด้วย $ ตามด้วยชื่อของ inner class
- ในกรณีที่เป็น anonymous class, compiler จะสร้างหมายเลขและใช้หมายเลขตามหลังแทนชื่อ

Constructors and polymorphism : order of calls

- ลำดับของการเรียก constructor
  - เรียก base class constructor ซึ่งจะทำการเรียก base class ในลักษณะ recursive ไปเรื่อย ๆ จนกว่าจะถึง root

    ทำ member initialized ตามลำดับการประกาศ
  - ทำการเรียกส่วน body ของ derived-class constructor

Behavior of polymorphic methods inside constructors

- ในกรณีที่เรียก dynamically-bound method (เช่น abstract method ที่ถูกกำหนด definition ภายหลัง) จากภายใน constructor, method ที่จะถูกเรียกใช้คือ method ใหม่ที่กำหนดทับ (overridden definition)
- ตัวอย่าง PolyConstructors.java

Downcasting and run-time type identification

- ในการ upcast ของ object ทำให้เสียลักษณะเฉพาะของ type นั้น แต่เป็นการ cast ที่ปลอดภัย เพราะ base class จะไม่มีทางที่จะมี interface ที่มากกว่า derived class ดังนั้นทุก message ที่ส่งเข้ามารับประกันว่าจะสามารถทำงานได้
- การทำ downcast ต้องทำแบบ explicit
- จาวาทำการตรวจสอบในช่วง runtime เพื่อที่จะประกันว่าเป็น type ที่ระบุจริง หากไม่จะเกิด ClassCastException ซึ่งการทำการตรวจสอบนี้เรียกว่า runtime type identification (RTTI)