Java语言程序设计

配套教材由清华大学出版社出版发行

第4章 面向对象程序设计之二

中国農業大學 阚道宏

第4章 面向对象程序设计之二

- 面向对象程序设计
  - 分类管理程序代码，即**类与对象**编程
  - 重用类代码
    - **组合**
    - **继承**

- 面向对象程序设计中的**多态**
  - 在字面上可理解为是一种程序代码的**多义性**
  - 进一步提高程序代码的**可重用性**

第4章 面向对象程序设计之二

- 本章内容
  - 4.1 重用类代码
  - 4.2 类的组合
  - 4.3 类的继承与扩展
  - 4.4 对象的替换与多态
  - 4.5 抽象类与接口
  - 4.6 四种特殊的类定义形式

4.1 重用类代码

- 程序=数据+算法
  - 定义变量是与数据相关的代码，即**数据代码**
  - 方法是与算法相关的代码，即**算法代码**

  - 类 = 数据代码+算法代码
  - 重用类代码的三种形式
    - 用类**定义**对象
    - 通过**组合**来定义新类
    - 通过**继承**来定义新类

4.1 重用类代码

- 用类定义对象
  - 类的4大要素

例4-1 一个钟表类Clock的完整定义代码（Clock.java）

```
1  public class Clock {     // 定义钟表类Clock
2     private  int hour, minute, second;  // 字段：保存时分秒数据
3     public  void set(int h, int m, int s)  // 方法：设置钟表对象的时间
4     {  hour = h;  minute = m;  second = s;  }
5     public  void show( )          // 方法：显示时间，显示格式：时:分:秒
6      {  System.out.println( hour +":" +minute +":" +second );  }
7
8     public  Clock( )              // 无参构造方法：将时分秒数据都设为0
9     {  hour = 0;  minute = 0;  second = 0;  }
10    public  Clock(int h, int m, int s)  // 有参构造方法：根据参数设置时间
11    {  hour = h;  minute = m;  second = s;  }
12    public  Clock( Clock oldObj ) {  // 拷贝构造方法：复制已有对象的时分秒数据
13    {  hour = oldObj.hour;  minute = oldObj.minute;  second = oldObj.second;  }
14 }
```

4.1 重用类代码

- 用类定义对象
  - 使用Clock类的典型流程

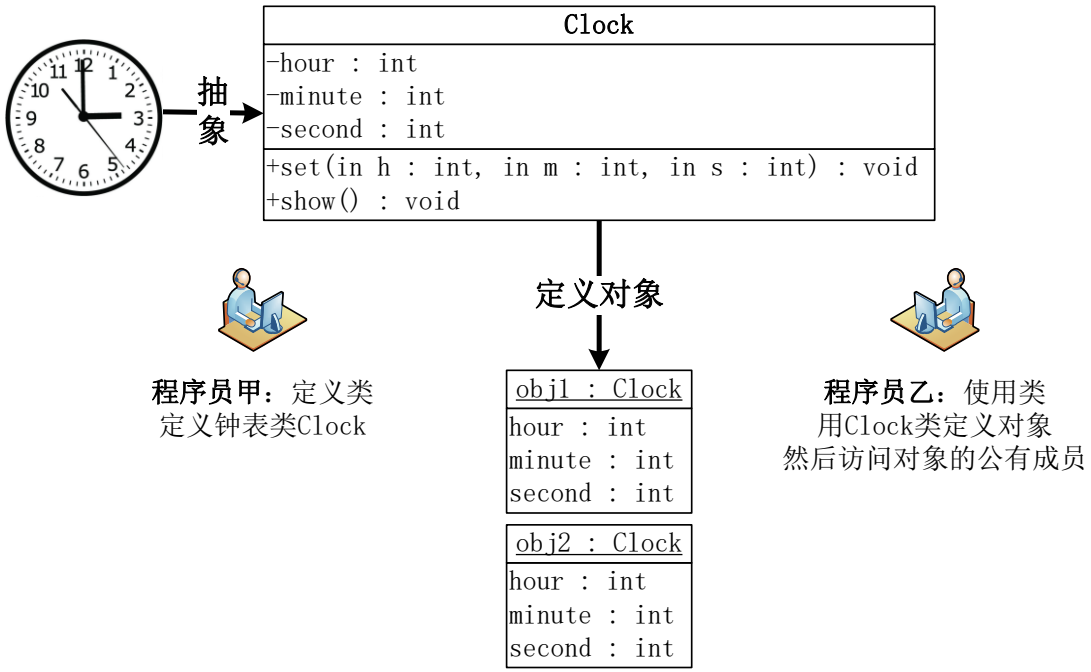
```
Clock obj1 = new Clock( );
obj1.set( 8, 30, 15);
obj1.show( ); // 显示结果：8:30:15
```

  - 用钟表类Clock定义多个对象

```
Clock obj2 = new Clock( 9, 30, 15 );
obj2.show( ); // 显示结果：9:30:15
```

4.1 重用类代码

- 用类定义对象



4.1 重用类代码

- 用类继续定义新类

抽象

DualClock
-hour1 : int -minute1 : int -second1 : int -hour2 : int -minute2 : int -second2 : int
+set1(in h : int, in m : int, in s : int) : void +show1() : void +set2(in h : int, in m : int, in s : int) : void +show2() : void

程序员乙：定义一个双时区钟表类DualClock

  - 双时区钟表类DualClock和钟表类Clock
  - 组合、继承

4.2 类的组合

- 用简单的**零件**组装复杂的**整体**

- 组合的编程原理

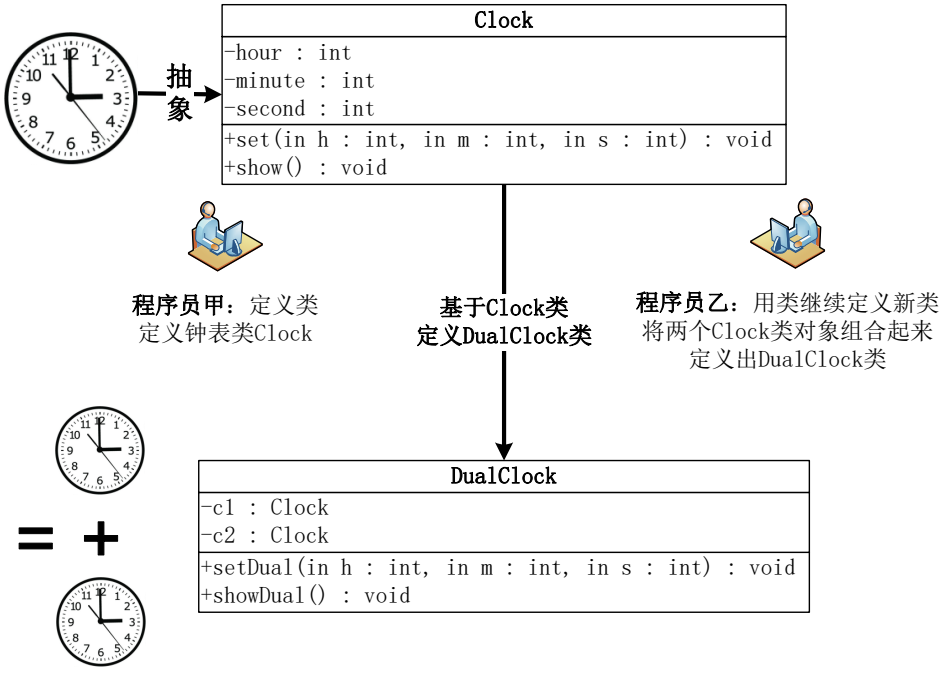
  程序员在定义新类的时候，使用已有的类来定义字段。这些字段是类类型的对象，被称为**对象字段**。Java语言将包含对象字段的类称为**组合类**。

  - 组合类的字段成员
    - **类类型**的对象字段
    - **基本数据类型**的非对象字段

  - 组合类对象
    - 组合类对象名. 非对象字段名
    - 组合类对象名. **对象字段名**. 对象字段的下级成员名

4.2 类的组合

- 组合类的定义
  - 双时区钟表类DualClock：由两个Clock对象组合而成



4.2 类的组合

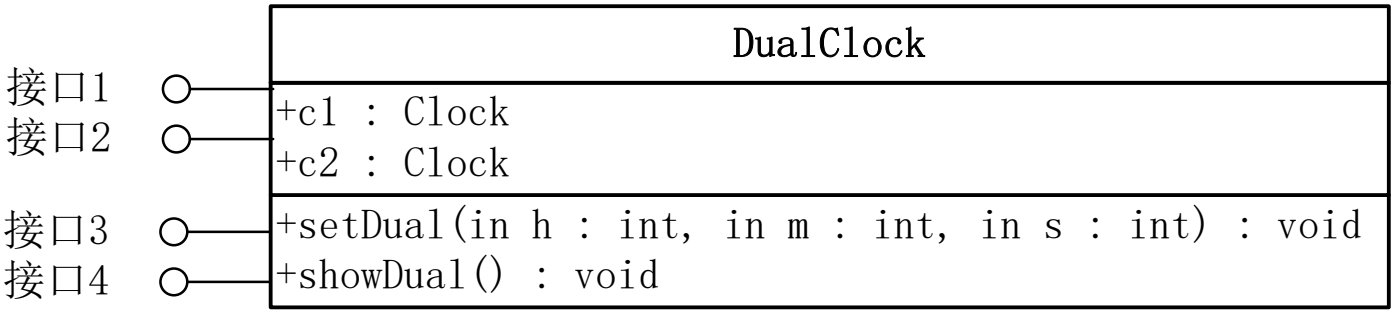
- 组合类的定义
  - 双时区钟表类DualClock

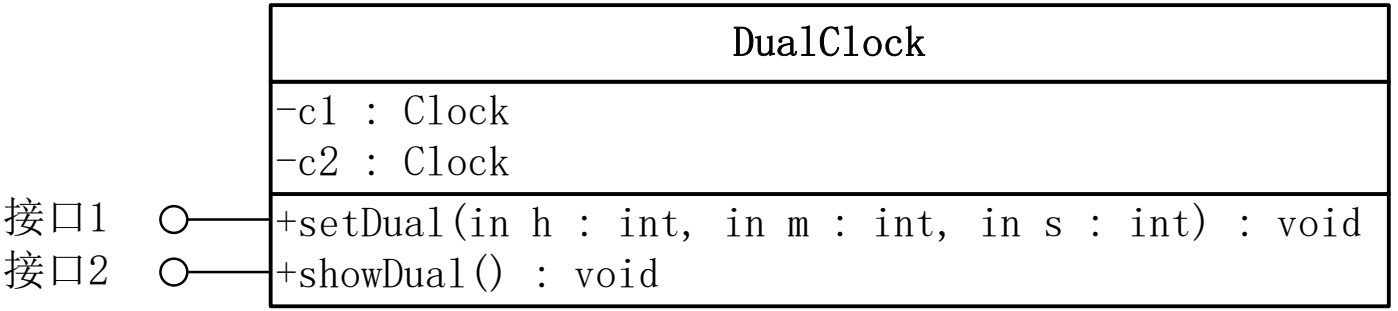
例4-2 一个使用钟表类Clock组合出的双时区钟表类DualClock（DualClock.java）

```
1  public class DualClock {    // 双时区钟表类：含有对象字段，属于组合类
2     public  Clock c1, c2;   // 对象字段：两个Clock类的钟表对象，设为公有成员
3     public  void setDual(int h, int m, int s)    // 设置方法：按参数设置c1、c2的时间
4     { c1.set( h, m, s ); c2.set( h +1, m, s); } // 假设设为两个连续的时区
5     public  void showDual( )              // 显示两个钟表的时间
6     { c1.show( ); c2.show( ); }
7
8     public  DualClock( ) { // 组合类需要定义自己的构造方法
9        c1 = new Clock( ); // 组合类构造方法需使用运算符new创建对象字段
10       c2 = new Clock( );
11    }
12 }
```

4.2 类的组合

- 组合类的定义
  - 对象字段的二次封装





4.2 类的组合

- 组合类对象的定义与访问
  - 定义组合类对象

    DualClock obj = **new** DualClock( );

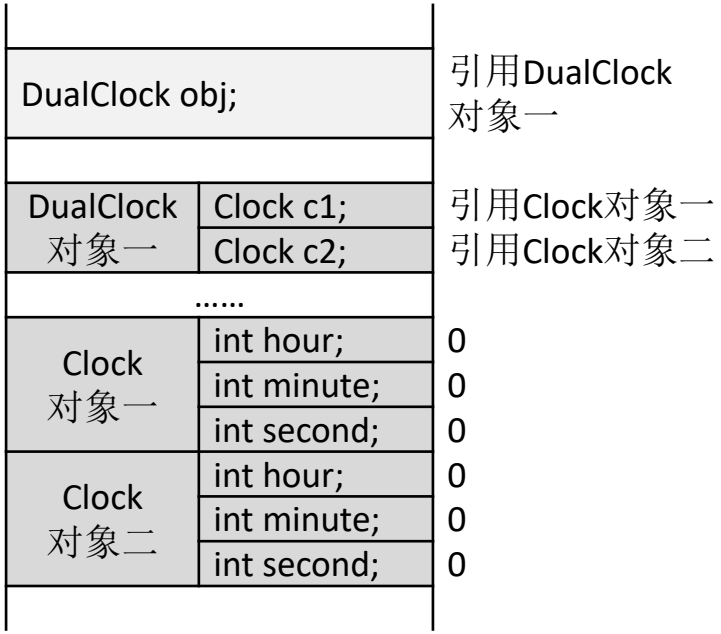
  - 访问组合类对象及其下级成员

    obj.**c1**、obj.**c2**

    obj.**setDual**( )、obj.**showDual**( )

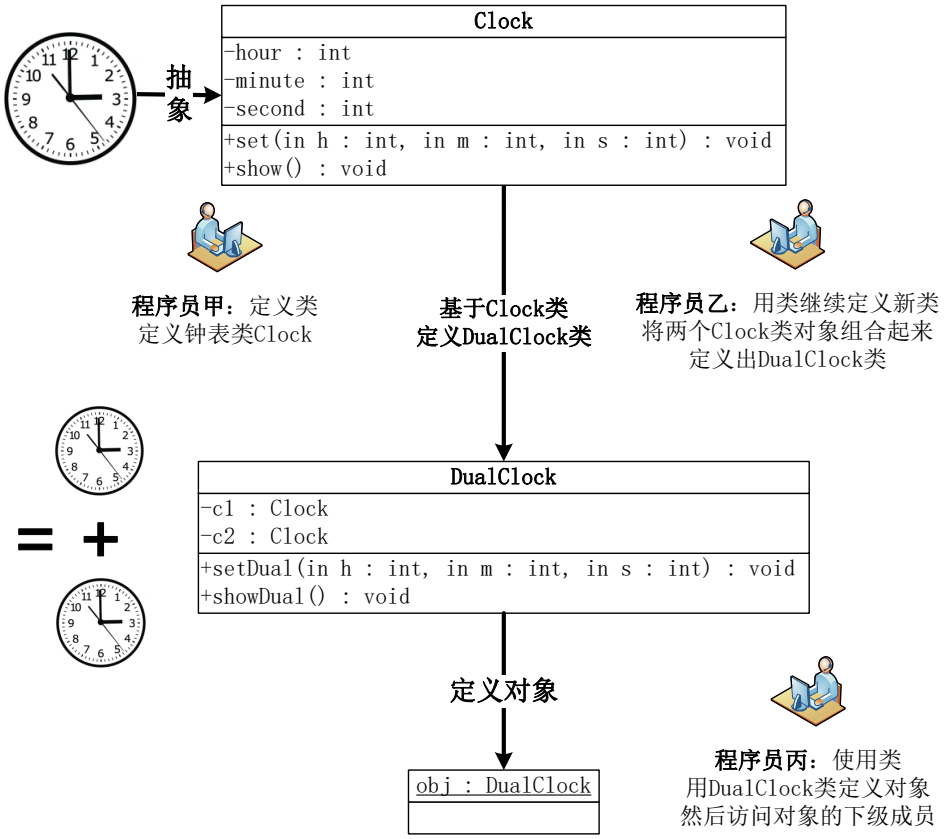
    obj.**c1**.set(10, 15, 30);
    obj.**c1**.show( ); // 显示：10:15:30

    obj.**c1**.hour = 10; // 错误
    obj.**c1**.minute = 15; // 错误
    obj.**c1**.second = 30; // 错误



4.2 类的组合

- 如何设计组合类中对象字段的访问权限？
  - 三个程序员角色
- 多级组合



4.2 类的组合

- 组合类的构造方法
  - 组合类中的两种字段成员
    - 基本数据类型的非对象字段
    - 类类型的对象字段：引用变量
  - 构造方法：如何为**对象字段**创建对象？

4.2 类的组合

- 组合类的构造方法
  - 为对象字段创建对象有四种方法
  - 在构造方法中为对象字段创建对象

```
c1 = new Clock( );
c2 = new Clock( );
```

  - 在定义对象字段时直接创建对象

```
public  Clock c1= new Clock(), c2= new Clock( );
```

4.2 类的组合

- 组合类的构造方法
  - 为对象字段创建对象有四种方法
  - 在构造方法中为对象字段创建对象
  - 在定义对象字段时直接创建对象
  - 向构造方法传递已经创建好的对象

```
public  DualClock( Clock p1, Clock p2 ) {  // 向构造方法传递两个已经创建好的钟表对象
   c1 = p1;  // 对象字段c1将直接引用所传递过来的钟表对象p1
   c2 = p2;  // 对象字段c2将直接引用所传递过来的钟表对象p2
}

Clock cObj1= new Clock( ), cObj2= new Clock();  // 先创建好两个钟表对象cObj1和cObj2
DualClock obj = new DualClock( cObj1, cObj2 );  // 定义组合类对象时再传递给构造方法

DualClock obj = new DualClock( new Clock(), new Clock() );
```

4.2 类的组合

- 组合类的构造方法
  - 为对象字段创建对象有四种方法

  - 在构造方法中为对象字段创建对象
  - 在定义对象字段时直接创建对象
  - 向构造方法传递已经创建好的对象

  - 直接引用其他组合类对象的对象字段

```
public  DualClock( DualClock p ) {  // 向构造方法传递一个已有的组合类对象
   c1 = p.c1;  // 对象字段c1将直接引用所传递过来组合类对象p的c1
   c2 = p.c2;  // 对象字段c2将直接引用所传递过来组合类对象p的c2
}

DualClock obj1 = new DualClock( obj );
```

例4-3 对钟表类Clock重新包装得到一个带日历功能的包装类DateClock

```
1  public class DateClock {  // 包装类（DateClock.java）
2      private  Clock c;   // 对象字段：被包装的原始钟表（Clock）对象c
3     // 以下代码都是为了对钟表对象c进行包装，为其增加日历功能
4      private  int year, month, day;       // 添加字段：保存年月日数据
5      public  void setDate(int y, int m, int d)  // 方法：设置日期
6      {  year = y;  month = m;  day = d;  }
7      public  void show() {            // 方法：显示日期和时间
8          System.out.print(year +"-" +month +"-" +day +" ");  // 先显示日期
9          c.show();                          // 再显示时间
10     }
11     public  Clock getClock()     // 方法：获得包装前的原始钟表对象c
12     {  return c;  }
13     public  DateClock(Clock obj)  // 构造方法：传递被包装的钟表对象
14     {  c = obj;  }           // 对象字段c直接引用传递过来的钟表对象obj
15 }
```

```
1  public class DateClockTest {  // 主类（DateClockTest.java）
2
3      public static  void main(String[] args) {  // 主方法
4          Clock cObj = new Clock( 10, 30, 15 );  // 定义一个钟表对象cObj
5          // 对钟表对象cObj进行包装，得到一个带日历的钟表对象dcObj
6          DateClock dcObj = new DateClock( cObj );
7          dcObj.setDate( 2018, 9, 1 );  // 设置dcObj的日期
8          dcObj.show();  // 显示dcObj的日期和时间，显示结果：2018-9-1 10:30:15
9      }
10 }
```

4.2 类的组合

- 组合类小结
  - 代码重用。组合是一种有效的重用代码形式。程序员在设计新类时应首先了解一下有哪些可以重用的类。这些类可以是自己以前编写的，或是JDK提供的，或是从市场上购买来的。可根据功能需要，采用组合的方法来设计新类
  - 多级组合。用零件类定义组合类，组合类可继续作为零件类去定义更大的组合类，这就是类的多级组合。多级组合是一种“自底向上”的程序设计方法。类越往上组合，其功能就越多
  - 多层封装。多级组合过程中，每一级组合类都会根据自己的功能需要设定对象字段的访问权限。有多少级组合，就会有多少层封装
  - 包装类。定义包装类的目的是增强或调整已有类的功能。包装也可以任意多级，即多级包装。包装类是组合类的一个特例

4.3 类的继承与扩展

- 设计新类时可以**继承**（inherit）已有的类，这个已有的类被称为**超类**（super class）或父类
- 在继承超类的基础上进行**扩展**（extend），或者对从超类继承来的功能进行**重新定义**（override，又称为**覆盖**或重写），这样所得到的新类被称为**子类**（sub class）
- 继承与扩展的编程原理
- 子类成员
  - 一种是从超类继承来的成员，称为**超类成员**
  - 另一种是定义时新添加或重写的新成员，称为**子类成员**

4.3 类的继承与扩展

Java语法：**定义子类**

```
[public] class 子类名 extends 超类名 {
   ……; // 新添加的成员
   ……; // 重新定义的成员
}
```

语法说明：

- 定义**子类**（新类）时，需使用关键字**extends**指定所继承的**超类**（已有类），然后在此基础上进行扩展。一个类只能继承一个超类，即类只能单继承。
- 子类将**继承**超类中的所有字段和方法（静态成员、构造方法除外），子类不用编写任何代码就能拥有与超类相同的字段和方法。子类中从超类继承来的成员被称为是**超类成员**。超类成员会保持其原有的访问权限和功能。
- 超类中的**静态成员**虽然未被子类继承，但可通过子类名或子类对象访问它们。静态成员可认为是被本类及其所有子类对象共享的成员。
- 子类可以**添加**新字段或新方法，这样就能扩展超类中没有的功能。子类新添加的成员被称为是**子类成员**。
- 子类可以**重新定义**（重写）超类成员，即添加与超类成员同名的字段，或具有相同签名的方法。访问这些成员，子类成员将**覆盖**（override)）重名的超类成员。重名的超类成员依然存在，但它们被屏蔽了。实际应用主要是重写超类继承来的方法，重写的目的是**替换**或**增强**原有方法的功能。虽然语法上可以重新定义超类继承来的字段，但没有什么实际用途，而且会造成数据混乱，建议尽量不使用。
- 可以访问被覆盖的超类成员，这时需使用关键字**super**来明确指定超类成员，例如super.字段名、super.方法名( )。关键字super代表超类。**注**：只能在子类新添加的方法成员中使用super关键字访问超类成员。

4.3 类的继承与扩展

- 子类的定义

抽象

Watch
-hour : int -minute : int -second : int -band : int
+set(in h : int, in m : int, in s : int) : void +show() : void +setBand(in b : int) : void

4.3 类的继承与扩展

- 子类的定义

例4-4 通过继承与扩展钟表类Clock所定义出的手表类Watch（Watch.java）

```
1  public class Watch extends Clock { // 继承超类Clock，在此基础上扩展出子类Watch
2     private  int band;        // 新添加的字段：表带类型，1-金属，2-皮革
3     public  void setBand(int b)  // 新添加的方法：设置表带类型
4     {  band = b;  }
5     public  void show( ) { // 重新定义显示时间的方法，显示格式：（表带类型）时:分:秒
6        // 先显示表带类型
7        if (band == 1)  System.out.print("（金属表带）");
8        else        System.out.print("（皮革表带）");
9        // 再显示时间：调用超类的方法show()
10       super.show();  // 关键字super表示超类
11    }
12 }
```

4.3 类的继承与扩展

- 子类的定义
  - 子类**继承**超类的成员

```
private  int hour, minute, second;  // 字段：保存时分秒数据
public   void set(int h, int m, int s); // 方法：设置钟表对象的时间
public   void show( );    // 方法：显示时间，显示格式：时:分:秒
```

  - 子类**添加**新成员或**重写**超类成员

```
private  int band; // 新添加的字段：表带类型，1-金属，2-皮革
public   void setBand(int b); // 新添加的方法：设置表带类型

public  void show( ); // 重写的方法，显示格式：(表带类型)时:分:秒
```

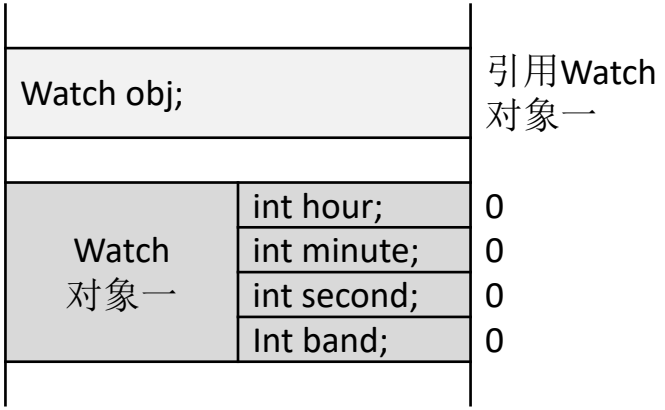
  - 重写超类方法的目的是**替换**或**增强**原有方法的功能

4.3 类的继承与扩展

- 子类的定义
  - 子类成员访问超类成员
    - 子类成员访问超类成员时会受到其访问权限的控制
      - private：hour、minute和second
      - public：show()
    - 两个重名的成员
      - 访问重名成员，访问到的将是重写后的新成员
      - 老成员被新成员“**覆盖**”了
      - 访问被覆盖的老成员：**super**.成员名

```
super.show(); // 关键字super表示超类
```

4.3 类的继承与扩展



- 子类对象的定义与访问
  - 定义子类对象

    Watch obj = **new** Watch( );

  - 子类对象obj包含8个成员
    - 超类成员

      obj.**hour**、obj.**minute**、obj.**second**、obj.**set**( )、obj.**show**( )
    - 子类成员

      obj.**band**、obj.**setBand**( )、obj.**show**( )
  - 访问子类对象中的超类成员
    - 私有的超类成员不可访问：obj.**hour**、obj.**minute**、obj.**second**
    - 访问不到子类对象中被覆盖的超类成员：obj.**show**()
  - 访问子类对象中的子类成员
    - 公有成员可以访问：obj.**setBand**( )、obj.**show**( )
    - 私有成员不可访问：obj.**band**

4.3 类的继承与扩展

- 保护权限protected

表 4-1 四种不同的类成员访问权限

范围 / 权限	在本类中访问	在本包中访问	在子类中访问	任意地方访问
private（私有权限）	可以访问			
未指定（默认权限）	可以访问	可以访问		
protected（保护权限）	可以访问	可以访问	可以访问	
public（公有权限）	可以访问	可以访问	可以访问	可以访问

  - 保护权限是在默认权限基础上，为子类定向开放的一种访问权限

4.3 类的继承与扩展

例4-5 一个关于保护权限的Java演示程序

程序员甲：定义类A（A.java）

```
1  public class A {
2     public    int x; // 公有权限
3     private   int y; // 私有权限
4     protected int z; // 保护权限
5     public  void aFun( ) { // 在本类中访问，不受访问权限控制
6        x = 10; // 访问公有成员，正确
7        y = 10; // 访问私有成员，正确
8        z = 10; // 访问保护成员，正确
9     }
10 }
```

程序员乙：使用类A定义对象（ATest.java）

```
1  public class ATest { // 测试类
2     public static  void main(String[] args) {
3        // 在其他类（非A的子类）中访问
4        A aObj = new A( ); // 先定义对象
5        aObj.x = 10; // 访问公有成员，正确
6        aObj.y = 10; // 访问私有成员，错误
7        // 如果与类A不在同一包中
8        aObj.z = 10; // 访问保护成员，错误
9        // 如果与类A在同一包中
10       aObj.z = 10; // 访问保护成员，正确
11    }
12 } // 场合一：保护权限= 默认权限
```

程序员丙：使用类A定义子类B（B.java）

```
public class B extends A {
   public  void bFun( ) { // 在子类B中访问
      x = 10; // 访问公有超类成员，正确
      y = 10; // 访问私有超类成员，错误
      z = 10; // 访问保护保护成员，正确
      // 子类可以访问保护权限的超类成员
      // 不管子类与超类在不在同一包中
   }
}


// 场合二：保护权限= 为子类定向开放的权限
```

4.3 类的继承与扩展

- 子类的构造方法
  - 子类中的两种成员
    - 从超类继承来的超类成员
    - 定义时新添加或重新定义的子类成员
  - 构造方法：如何初始化继承来的**超类字段**？
    - 调用超类的构造方法

      super(初始值列表);
  - 举例
    - 钟表类Clock：无参构造方法、有参构造方法、拷贝构造方法
    - 如何定义手表类Watch的构造方法？

4.3 类的继承与扩展

例4-6 为子类Watch定义的构造方法示例代码

```
1  public class Clock { // 钟表类Clock（超类）
2      …… // 这里只节选例4-1中的构造方法，其他代码省略
3    public  Clock( )          // 无参构造方法：将时分秒字段都设为0
4    { hour = 0;  minute = 0;  second = 0;  }
5    public  Clock(int h, int m, int s)  // 有参构造方法：根据参数设置时间
6    { hour = h;  minute = m;  second = s;  }
7    public  Clock( Clock oldObj ) {  // 拷贝构造方法：复制已有对象的时分秒字段
8    { hour = oldObj.hour;  minute = oldObj.minute;  second = oldObj.second;  }
9  }
```

```
1  public class Watch extends Clock { // 手表类Watch（子类）
2    …… // 这里列出为子类Watch定义的构造方法，其他代码省略
3    public  Watch( ) { // 无参构造方法
4      super( );    // 先调用超类Clock的无参构造方法，初始化超类字段（时分秒）
5                 // 也可以调用超类Clock的有参构造方法：super( 0, 0, 0 );
6      band = 1;    // 然后再初始化子类字段：表带类型，直接对其赋值
7     }
8     public  Watch( int h, int m, int s, int b ) { // 有参构造方法：初始化时分秒和表带类型
9      super( h, m, s );  // 需调用超类Clock的有参构造方法，初始化超类字段（时分秒）
10     band = b;     // 然后再初始化子类字段：表带类型，直接对其赋值
11    }
12    public  Watch( Watch oldObj ) { // 拷贝构造方法
13     super( oldObj );    // 先调用超类Clock的拷贝构造方法，初始化超类字段
14     band = oldObj.band;  // 然后再初始化子类字段：表带类型，直接对其赋值
15    }
16 }
```

4.3 类的继承与扩展

- 子类构造方法的语法细则
  - 在子类构造方法中可以使用关键字super调用超类的构造方法，其目的是初始化超类字段。因为超类字段可能是私有的，在子类中不能访问，因此必须通过超类的构造方法才能进行初始化
  - 如果编写super调用语句，则该语句必须是构造方法的第一条语句
  - 如果没有编写super调用语句，则编译器会自动在构造方法的第一行增加如下的调用语句：

```
super( ); // 调用超类的无参构造方法
```

4.3 类的继承与扩展

- 子类字段成员的初始化过程
  - 子类可以在三个地方对字段成员进行初始化
    - ① 在构造方法的最开头编写super调用语句，调用超类的构造方法来初始化**超类字段**
    - ② 在添加新的**子类字段**时，可在定义时做初始化赋值
    - ③ 在构造方法的方法体中使用赋值语句进行初始化。可以在这里初始化新添加的**子类字段**，也可以初始化从超类继承来并且是**可访问**的**超类字段**
  - 创建子类对象时，上述初始化代码的执行顺序依次是①②③

4.3 类的继承与扩展

例4-7 一个初始化子类中字段成员的Java演示程序

超类Sup（Sup.java）

```
1   class Sup { // 定义超类Sup
2     public    int x; // 公有成员
3     private   int y; // 私有成员
4     protected int z; // 保护成员
5     public  Sup( ) {  // 构造方法
6       // 在构造方法中显示创建过程
7       System.out.println("Sup enter: " +x +y +z);
8       x = 1;  y = 1;  z = 1;
9       System.out.println("Sup exit: " +x +y +z);
10    }
11  }
12
13
```

子类Sub（Sub.java）

```
class Sub extends Sup { // 定义子类Sub
  private int a = 2; // 新添加的成员，初始化②
  public  Sub( ) {   // 子类的构造方法
    super( );      // 初始化①
    // 在构造方法中显示创建过程
    System.out.println("Sub enter: " +x +"?" +z +a);
    a = 3;         // 初始化③
    x = 3;
    // y = 3; // 子类不能访问私有的超类成员
    z = 3;
    System.out.println("Sub exit: " +x +"?" +z +a);
  }
```

```
1   public class FieldInitDemo { // 测试类（Fiel
2     public static void main(String[] args) { // 主
3       Sub obj = new Sub( );        // 创建子类S
4     }
5   }
```

Problems @ Javadoc Declaration Console

<terminated> FieldInitDemo [Java Application] C:\Java

```
Sup enter: 000
Sup exit: 111
Sub enter: 1?12
Sub exit: 3?33
```

4.3 类的继承与扩展

- 关键字final
  - final：最终的、不可更改的
  - final局部变量：只读**变量**

```
final  double PI = 3.14;

final  double PI;
PI = 3.14;

final  Clock c = new Clock(8, 30, 15);
c = new Clock(9, 30, 15);  // 错误：不能改变引用
```

4.3 类的继承与扩展

- 关键字final
  - final：最终的、不可更改的
  - final局部变量：只读变量

  - final字段：只读**字段**

```
public class A {
    public final  int a = 10;
    ……; // 其他代码省略
}

A obj = new A( );
System.out.println( obj.a );  // 显示结果：10
obj.a = 20;                   // 错误：不能再修改字段a的值
```

4.3 类的继承与扩展

- 关键字final
  - final：最终的、不可更改的
  - final局部变量：只读变量
  - final字段：只读字段

  - final方法：最终**方法**

    [访问权限] **final** 返回值类型 **方法名**(形式参数列表)

    { …… }

  - final类：最终**类**

    [访问权限] **final** class 类名 {

    { …… }

4.4 对象的替换与多态

- 子类是超类这个大类下细分的小类，因此一个子类对象可以被**当作**超类对象使用
- 面向对象程序设计利用子类和超类之间的这种特殊关系，提出了对象的**替换**与**多态**，其目的是为了提高程序中**算法代码**的重用性
- 程序中最常见的算法代码形式是**方法**（即函数）
- 钟表类Clock

4.4 对象的替换与多态

- 算法代码的重用性
  - 什么是算法代码的重用性？

```
void fun( int x ) { System.out.println(x*x); }
fun( 5 ); // 正确

fun( 5.8 ); // 错误
```

  - **结论1**：Java语言对数据类型一致性的要求比较严格，属于**强类型检查**的计算机语言

```
void fun( double x ) { System.out.println(x*x); }
```

4.4 对象的替换与多态

- 算法代码的重用性
  - 什么是算法代码的重用性？

```
class A { … }
void aFun( A x ) { … }

A  aObj = new A();
aFun( aObj ); // 正确

class B { … }
B  bObj = new B();
aFun( bObj ); // 错误
```

  - **结论2**：不能调用处理A类对象的方法aFun()来处理B类的对象数据

  - 对象的**替换**与**多态**：如果类B是类A的子类

```
class B  extends A { … } // 类B继承类A，是类A的子类
```

4.4 对象的替换与多态

- 算法代码的重用性
  - 钟表类Clock及其处理算法举例

例4-8 一个处理钟表类Clock对象的方法setGMT()及其测试类GMTTest（GMTTest.java）

```
1  public class GMTTest { // 测试类
2      public static  void main(String[] args) { // 主
3          Clock cObj = new Clock( );      // 创建一
4          // 给定GMT时间8:30:15，调用方法setG
5          setGMT(cObj, 8, 30, 15);
6      }
7
8      // 处理钟表类Clock对象的方法setGMT()：
9      // 给定GMT时间，先将其转换成北京时间，然后再设置给钟表对象obj
10     public static  void setGMT(Clock obj, int hGMT, int mGMT, int sGMT) {
11         int h, m, s;   // 先定义3个保存北京时间的变量
12         h = hGMT +8;  // 北京时间比GMT时间早8个小时，即小时数加8
13         m = mGMT;  s = sGMT;
14         obj.set(h, m, s);  // 将转换后的北京时间设置给钟表对象obj
15         obj.show();    // 显示时间：GMT时间8:30:15所对应的北京时间是16:30:15
16 } }
```

类代码：钟表类Clock
算法代码：setGMT()

4.4 对象的替换与多态

- 类族及其处理算法
  - 钟表类Clock

例4-9 从钟表类Clock扩展出的的3个子类

手表类**Watch**

```
1  class Watch  extends Clock { // 手表类
2    public int band = 1;   // 新添加表带类型
3    public void show( ) { // 重写show方法
4      if (band == 1)  // 金属表带
5        System.out.print( “(金属表带)” );
6      else       // 皮革表带
7        System.out.print( “(皮革表带)” );
8      super.show();
9  } }
```

潜水表类**DivingWatch**

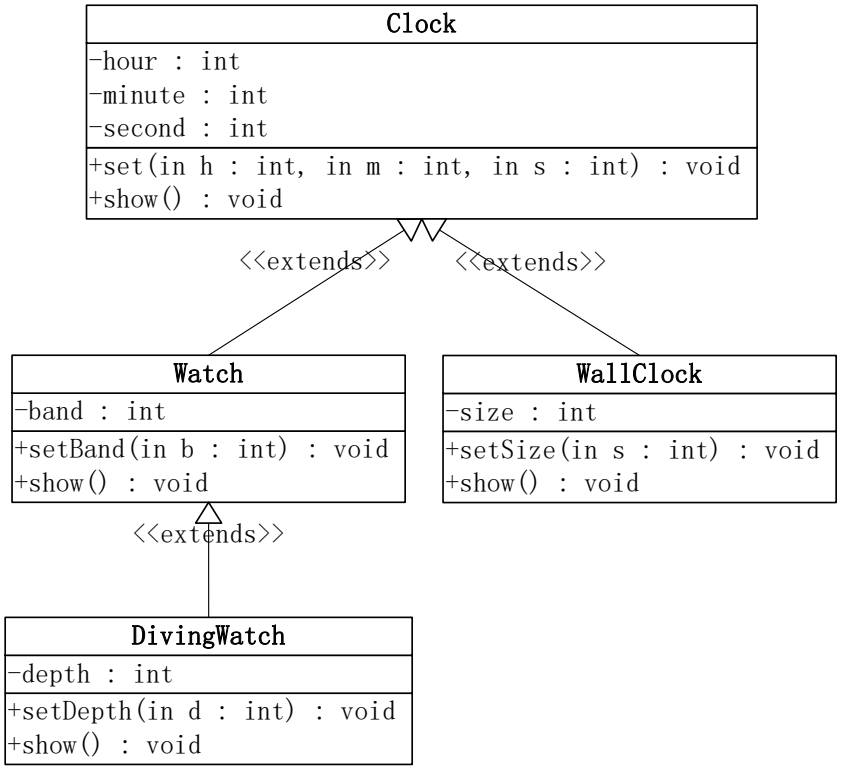
```
1  class DivingWatch extends Watch { // 潜水表类
2    public int depth = 10;  // 新添加最大深度
3      public void show( ) { // 重写show方法
4        System.out.print( “(“+depth+”米)” );
5        super.show();
6  } }
```

挂钟类**WallClock**

```
class WallClock  extends Clock { // 挂钟类
  public int size = 12;   // 新添加表盘尺寸
  public void show( ) { // 重写show方法
    System.out.print( “(“ +size +”英寸)” );
    super.show();
} }
```

4.4 对象的替换与多态

- 类族及其处理算法
  - 多级继承与扩展
  - 类族
  - 钟表类族
  - 继承与扩展
    - 重用钟表类Clock的类代码



类代码：钟表类Clock
算法代码：setGMT()

4.4 对象的替换与多态

- 类族及其处理算法
  - 同一类族中对象的**多态性**
    - 显示时间的方法**Show**()
    - 钟表类Clock

```
Clock  cObj = new Clock( );  // 钟表类Clock的对象
cObj.set( 8, 30, 15 );       // 将时间设置为8:30:15
cObj.show( );                // 显示时间：8:30:15
```

    - 手表类Watch

```
Watch  wObj = new Watch( );  // 手表类Watch的对象
wObj.set( 8, 30, 15 );       // 将时间设置为8:30:15
wObj.show( );                // 显示时间：(金属表带)8:30:15
```

    - 挂钟类WallClock

```
WallClock  wcObj = new WallClock( );  // 挂钟类WallClock的对象
wcObj.set( 8, 30, 15 );               // 将时间设置为8:30:15
wcObj.show( );                        // 显示时间：(12英寸)8:30:15
```

    - 潜水表类DivingWatch

```
DivingWatch  dwObj = new DivingWatch( );  // 潜水表类DivingWatch的对象
dwObj.set( 8, 30, 15 );                   // 将时间设置为8:30:15
dwObj.show( );                            // 显示时间：(10米)(金属表带)8:30:15
```

  - 同一类族中不同钟表对象会显示出不同的时间格式，我们称这些钟表对象表现出了**多态性**（Polymorphism）

4.4 对象的替换与多态

- 米老鼠和唐老鸭
  - 下达指令**Go**
- 类A的对象aObj和类B的对象bObj
  - 调用对象的方法成员**fun()**：**aObj.fun( )**；**bObj.fun( )**;
  - 执行方法fun()
  - 不同fun()方法完成不同的功能：对象具有多态性
- 消息
  - 调用对象的某个方法成员：向对象发送一条消息
  - 执行方法成员完成某种程序功能：对象响应该消息
- 对象多态性
  - 调用不同对象的同名方法成员，但所执行的方法不同，完成的程序功能也不同
  - 面向对象程序设计只关注子类与超类之间存在的多态性问题

4.4 对象的替换与多态

- 类族及其处理算法
  - 子类与超类共用算法代码

```
public static  void setGMT(Clock obj, int hGMT, int mGMT, int sGMT) {
   int h, m, s;      // 先定义3个保存北京时间的变量
   h = hGMT +8;  // 北京时间比GMT时间早8个小时，即小时数加8
   m = mGMT;  s = sGMT;
   obj.set(h, m, s);  // 将转换后的北京时间设置给钟表对象
   obj.show();        // 显示转换后的北京时间
}
Clock cObj = new Clock( );  // 创建一个Clock类的钟表对象cObj
setGMT(cObj, 8, 30, 15);   // 传递一个GMT时间8:30:15
```

  - **问题**：能否重用处理超类Clock对象的方法setGMT()，为子类对象设置GMT时间？

```
Watch wObj = new Watch( );  // 创建一个Watch类的手表对象wObj
setGMT(wObj, 8, 30, 15);       // 传递一个GMT时间8:30:15
```

  - **讨论1**：形参**obj**和实参**wObj**类型不一致怎么办？
  - **讨论2**：通过超类Clock的引用变量obj调用子类Watch对象的显示时间方法show()，会调用哪个**show**()？**注**：子类中有多个同名的方法show()

4.4 对象的替换与多态

- 对象的替换与多态
  - 超类引用变量引用子类对象

    对象替换语法规则：可以将子类对象的引用赋值给超类的引用变量，即超类引用变量可以引用子类对象。

    ```
    Clock cObj1 = new Watch( );          // 类Watch是类Clock的一级子类
    Clock cObj2 = new WallClock( );      // 类WallClock是类Clock的一级子类
    Clock cObj3 = new DivingWatch( );  // 类DivingWatch是类Clock的二级子类
    ```

  - 对象替换：将子类对象当作一个超类对象来使用

  - 将超类的引用变量赋值给子类的引用变量

    ```
    Watch wObj = (Watch) cObj1;  // cObj1必须确实引用了一个类Watch的对象
    WallClock wcObj = (WallClock) cObj2;
    DivingWatch dwObj = (DivingWatch) cObj3;
    ```

4.4 对象的替换与多态

- 对象的替换与多态
  - 通过超类引用变量访问子类对象的成员
    - 子类对象中的超类成员（**老成员**）和子类成员（**新成员**）
    - 对象多态语法规则
      - 通过超类引用变量访问子类对象的成员，只能访问从超类继承来的**老成员**，不能访问子类新添加的**新成员**

```
Clock cObj = new Watch( );
cObj.set( 8, 30, 15 );  // 正确
cObj.band = 1;          // 错误
```

      - 如果子类重新定义了超类成员，则通过超类引用变量访问子类对象中的同名成员，将自动调用子类重写的**新成员**

```
Clock cObj = new Watch( );
cObj.set( 8, 30, 15 );
cObj.show( );  // 显示结果：(金属表带)8:30:15
```

4.4 对象的替换与多态

- 对象的替换与多态
  - 类族共用算法代码：设置GMT时间的算法代码**setGMT()**
    - 钟表类Clock（超类）

      Clock cObj = new Clock( ); // 钟表类Clock的对象

      setGMT(cObj, 8, 30, 15); // 设置GMT时间，显示结果：**16:30:15**
    - 手表类Watch（一级子类）

      Watch wObj = new Watch( ); // 手表类Watch的对象

      setGMT(wObj, 8, 30, 15); // 设置GMT时间，显示结果：**(金属表带)16:30:15**
    - 挂钟类WallClock（一级子类）

      WallClock wcObj = new WallClock( ); // 挂钟类WallClock的对象

      setGMT(wcObj, 8, 30, 15); // 设置GMT时间，显示结果：**(12英寸)16:30:15**
    - 潜水表类DivingWatch（二级子类）

      DivingWatch dwObj = new DivingWatch( ); // 潜水表类DivingWatch的对象

      setGMT(dwObj, 8, 30, 15); // 设置GMT时间，显示结果：**(10米)(金属表带)16:30:15**
  - 从各钟表对象所显示的时间格式来看，类族不仅可以**共用**算法代码，而且共用时还能继续**保持**对象的多态性

4.4 对象的替换与多态

- 运算符 instanceof
  - 一个超类引用变量所引用的可能是超类对象，也可能是子类对象
  - 如何确定引用变量到底引用的是哪个类的对象呢？

    引用变量名 **instanceof** 类名

    Clock cObj = new **Watch**( );

    表达式“cObj **instanceof** Watch”的计算结果为**true**
    表达式“cObj **instanceof** Clock”的计算结果为**true**
    表达式“cObj **instanceof** DivingWatch”的计算结果为**false**

  - 子类与超类之间的关系
    - 子类可被认为是一种超类。例如，手表是一种钟表，手表类是钟表这个大类下一个细分的小类。注：反过来不成立，例如钟表不能被认为是一种手表
    - 超类可以代表子类。例如，钟表是手表、挂钟和潜水表等经泛化、抽象后得到的上层概念，可代表各种不同形式的钟表

4.5 抽象类与接口

- 凝练类代码
  - 可以继续将多个不同类中的共性抽象出来形成超类
  - 编码时再从超类继承，扩展出各个不同的类

例4-10 一个本科生类和研究生类的Java示意代码

	Undergraduate：本科生类	Graduate：研究生类
1	public class **Undergraduate** { // 本科生类	public class **Graduate** { // 研究生类
2	public char **Name**[], **ID**[]; // 姓名、学号	public char **Name**[], **ID**[]; // 姓名、学号
3	public int **Age**; // 年龄	public int **Age**; // 年龄
4	public float **Score**; // 课堂成绩	public float **Score**; // 课堂成绩
5	public float DesignScore; // 毕业设计成绩	public float PaperScore; // 毕业论文成绩
6		public int Thesis; // 发表论文数量
7	public void **Input**() { … } // 输入学生信息	public void **Input**() { … } // 输入学生信息
8	public void **ShowInfo**() { … } // 显示学生信息	public void **ShowInfo**() { … } // 显示学生信息
9	public float TotalScore() { … } // 计算总成绩	public float TotalScore() { … } // 计算总成绩
10	}	}

4.5 抽象类与接口

- 凝练类代码

例4-11 抽象超类（学生类）后再扩展本科生类和研究生类的Java示意代码

Student：学生类（抽象出的超类）

```
1  public class Student {      // 超类：学生类
2    public char Name[], ID[];   // 姓名、学号
3    public int Age;          // 年龄
4    public float Score;       // 课堂成绩
5    public void Input() { … }    // 输入学生基本信息
6    public void ShowInfo() { … } // 显示学生基本信息
7  }
```

Undergraduate：本科生类（子类）

```
1  public class Undergraduate extends Student {
2    public float PracticeScore;  // 毕业设计成绩
3
4    public float TotalScore() { … } // 计算总成绩
5    public void Input() { … }    // 重写输入方法
6    public void ShowInfo() { … } // 重写显示方法
7  }
```

Graduate：研究生类（子类）

```
class Graduate extends Student {
  public double PaperScore;  // 毕业论文成绩
  public int Thesis;         // 发表论文数量
  public float TotalScore() { … } // 计算总成绩
  public void Input() { … }    // 重写输入方法
  public void ShowInfo() { … } // 重写显示方法
}
```

4.5 抽象类与接口

- 抽象方法与抽象类

例4-12 一个圆形类和长方形类的Java示意代码

	Circle：圆形类	Rectangle：长方形类
1	public class **Circle** {　　// 圆形类	public class **Rectangle** {　　// 长方形类
2	public double r;　　// 半径	public double a, b;　　// 长、宽
3	public double **area**() { … } // 求面积	public double **area**() { … } // 求面积
4	public double **len**() { … }　// 求周长	public double **len**() { … }　// 求周长
5	}	}

```
public abstract  class Shape {      // 形状类
   public abstract  double area();  // 求面积方法的签名
   public abstract  double len();   // 求周长方法的签名
}
```

– Java语言将只给出方法签名，但没有方法体的方法称为**抽象**（abstract）**方法**，定义时需使用关键字abstract进行修饰
– 将含有抽象方法的类称为**抽象类**，定义时也必须使用关键字abstract

4.5 抽象类与接口

- 抽象方法与抽象类
  - 抽象类的语法细则
    - 抽象类不能实例化
    - 抽象类可以作为超类定义子类
      - 抽象类可以作为超类定义子类。子类将继承抽象类中除静态成员和构造方法之外的所有成员，包括其中的抽象方法
      - 抽象方法只设计了方法的调用接口，即只定义了方法签名，但没有提供方法体代码。子类继承了抽象方法的调用接口，还需要为抽象方法编写具体的算法代码，这被称为是实现（implement）抽象方法
      - 子类如果实现了所有的抽象方法，那么它就变成了一个普通的类，可以实例化；否则子类仍然是一个抽象类，定义时继续使用关键字abstract
      - 可以定义抽象类的引用变量，所定义的引用变量可以引用其子类的实例化对象
    - 抽象类可以包含字段和非抽象的方法

4.5 抽象类与接口

- 抽象方法与抽象类
  - 几何形状类Shape

例4-13 继承几何形状类Shape的圆形类和长方形类的Java示例代码

	Circle：圆形类	Rectangle：长方形类
1	public class **Circle** extends **Shape** { // 圆形类	public class **Rectangle** extends **Shape** { // 长方形类
2	public double r; // 添加字段半径	public double a, b; // 添加字段长、宽
3	public double **area**() // 实现抽象方法	public double **area**() // 实现抽象方法
4	{ return (3.14*r*r); }	{ return (a*b); }
5	public double **len**() // 实现抽象方法	public double **len**() // 实现抽象方法
6	{ return (3.14*2*r); }	{ return (2*(a+b)); }
7	public Circle(double x) // 构造方法	public Rectangle(double x, double y) // 构造方法
8	{ r = x; }	{ a = x; b = y; }
9	}	}

  - 圆形类Circle和长方形类Rectangle都是抽象类Shape的子类，它们共同组成了一个以抽象类Shape为超类的类族，可称为**几何形状类族**

4.5 抽象类与接口

- 抽象方法与抽象类
  - 抽象类的应用
    - 抽象类及其子类共同组成一个类族
      - 统一类族对外的使用接口
      - 让类族共用算法代码
    - 统一类族的使用接口

```
Circle  cObj = new Circle( 10 );
Rectangle  rObj = new Rectangle( 5, 10 );

System.out.println( cObj.area() +", " +cObj.len() );
System.out.println( rObj.area() +", " +rObj.len() );
```

4.5 抽象类与接口

- 抽象方法与抽象类
  - 抽象类的应用
    - 抽象类及其子类共同组成一个类族
      - 统一类族对外的使用接口
      - 让类族共用算法代码

    - 让类族共用算法代码

```
public  void shapeInfo( Shape sObj ) {  // 显示面积、周长等形状信息
   System.out.println( sObj.area() +", " +sObj.len() );
}

Circle  cObj = new Circle( 10 );         // 定义一个圆形类对象cObj
Rectangle  rObj = new Rectangle( 5, 10 );  // 定义一个长方形类对象rObj

shapeInfo( cObj );  // 共用方法shapeInfo，显示圆形对象的形状信息
shapeInfo( rObj );  // 共用方法shapeInfo，显示长方形形对象的形状信息
```

4.5 抽象类与接口

- 接口
  - **接口**（interface）是一种特殊的抽象类，其中只包含抽象方法、静态方法或静态只读字段等特殊成员

```
public abstract  class Shape {      // 形状类
    public abstract  double area();  // 求面积方法的签名
    public abstract  double len();   // 求周长方法的签名
}
```

  - 几何形状类Shape只能说是一种“**可以计算面积和周长的**”接口
  - Java语言将接口从抽象类中独立出来，单独作为一种**引用数据类型**。接口在Java语言中占有非常重要的地位

Java语法：**定义接口**和**实现接口**

```
[public] interface 接口名 [extends 父接口名列表] {
   [public static final] 数据类型 公有静态只读字段名 = 初始值;
   ……
   [public static]   返回值类型 公有静态方法名( 形式参数列表 ) { …… }
   ……
   [public abstract] 返回值类型 公有抽象方法名( 形式参数列表 );
   ……
}

class 类名 implements 接口名列表 {
   实现从接口继承的抽象方法
   定义类的其他成员
}
```

语法说明：

- 定义接口时使用关键字**interface**。
- **接口名**需符合标识符的命名规则，习惯上以大写字母开头；通常也会以“-able”或“-ible”结尾，表示“可做什么操作的”接口。
- 定义接口时可以使用关键字**extends**继承其他接口（称为**父接口**），然后在此基础上进行扩展。接口可以继承多个父接口，多个父接口之间用逗号隔开。接口可以多继承。
- 接口的成员只能有三种，分别是**公有静态只读字段**（public static final，可省略）、**公有静态方法**（public static，可省略）和**公有抽象方法**（public abstract，可省略），它们都是公有权限。接口中的抽象方法只有方法签名，但没有方法体，其目的是为了向外界提供一种统一的操作接口标准。某些接口仅包含一个抽象方法成员，这被称为是**功能接口**。某些接口不包含任何成员，是一个空接口，这被称为是**标记接口**。
- 接口可以被类**实现**。定义类时可使用关键字**implements**实现某个接口，然后在类中对接口里的抽象方法进行重新定义，并编写完整的方法体代码，这个过程就被称为是类对接口的实现。类实现接口的目的是为了继承其中的接口设计，然后按照统一的接口标准（即方法具有统一的调用接口）向外界提供服务（即方法可以被外界调用）。实现了某个接口的类被称为是该接口的子类。
- 一个类只能继承一个超类（即**单继承**），但可以实现多个接口（即**多实现**）。实现多个接口时，多个接口之间用逗号隔开。
- 接口是一种特殊的**引用数据类型**。可以用接口定义引用变量，但不能创建对象。接口类型的引用变量可以引用其子类的对象。

4.5 抽象类与接口

- 接口
  - 一个儿童手表类ChildrenWatch举例
  - 儿童手表类**ChildrenWatch** = 手表Watch +**打电话** +**定位**
    - 定义一个“**可以打电话的**”接口Callable

```
public interface  Callable {
   int pNumber = 1234;
   void call(int number);
   void answer( );
}
```

    - 定义一个“**可以定位的**”接口Positionable

```
public interface  Positionable {
   void showPosition( ); // 显示定位的操作接口标准：公有抽象方法
}
```

    - 定义儿童手表类ChildrenWatch

4.5 抽象类与接口

例4-14 一个完整的儿童手表类ChildrenWatch定义（ChildrenWatch.java）及测试代码

```
1  public class ChildrenWatch  extends Watch  implements Callable, Positionable {
2     // 继承手表类Watch，同时实现接口Callable和Positionable
3     public  void call(int number)  // 实现接口Callable所规定的打电话方法
4     { System.out.println("Call " +number); } // 显示一个模拟打电话的提示信息
5     public  void answer( )      // 实现接口Callable所规定的接电话方法
6     { System.out.println("Answer a call"); } // 显示一个模拟接电话的提示信息
7
8     public  void showPosition( ) // 实现接口Positionable所规定的显示定位方法
9     { System.out.println("Show position"); } // 显示一个模拟定位的提示信息
10 }
```

```
1  public class CWatchTest { // 测试类（CWatchTest.java）
2     public static  void main(String[] args) {       // 主方法
3        ChildrenWatch cw = new ChildrenWatch( ); // 使用儿童手表类定义对象
4        cw.set(8, 30, 15);   // 设置手表时间
5        cw.show( );       // 显示手表信息：（金属表带）8:30:15
6        cw.call(6789);      // 打电话。模拟显示结果：Call 6789
7        cw.answer( );      // 接电话。模拟显示结果：Answer a call
8        cw.showPosition( ); // 显示定位。模拟显示结果：Show position
9     }
10 }
```

4.5 抽象类与接口

- 接口
  - 抽象方法的默认方法体

```
public interface Positionable {
   // void showPosition( ) ;      // 无默认方法
   default  void showPosition( )  // 有默认方法
   {  System.out.println( "Show position" );  }
}
```

  - 继承超类与实现接口
    - 子类**继承超类**：子类是超类的扩展，但仍属于超类，与超类具有类属关系。超类的功能是子类的主要功能
    - 子类**实现接口**：为接口实现具体功能，但这些功能只是子类的次要功能
    - 类只能**单继承**，接口可以**多实现**

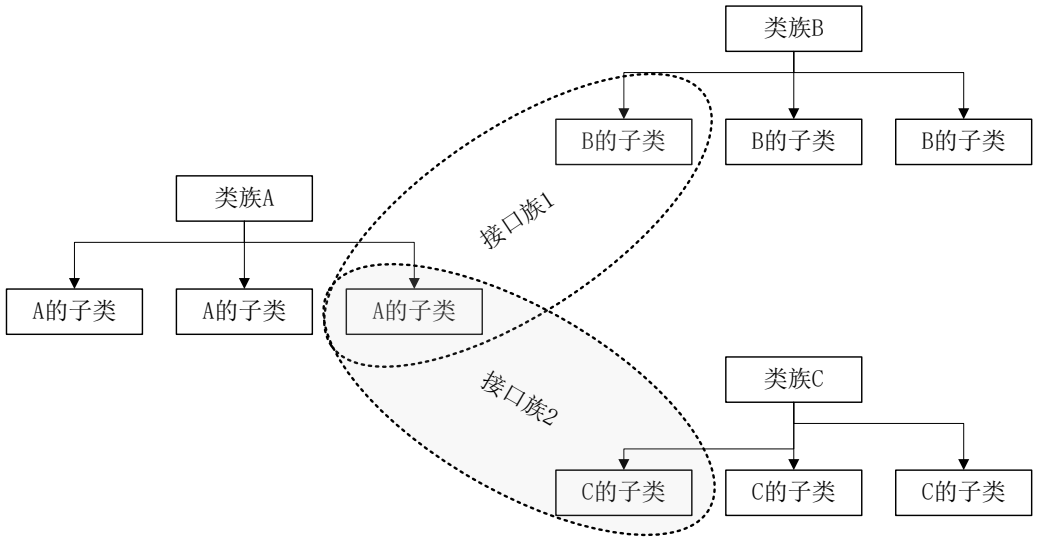
4.5 抽象类与接口

- 接口
  - 接口的应用
    - 使用接口定义引用变量

```
ChildrenWatch cw = new ChildrenWatch( );

// 通过超类引用变量访问子类对象
Watch  r1 = cw;        // 超类Watch的引用变量r1，引用子类对象cw
r1.show( );            // 通过r1只能访问超类Watch曾经定义过的成员

// 通过Callable接口引用变量访问子类对象
Callable  r2 = cw;              // 接口Callable的引用变量r2，引用子类对象cw
r2.call(6789);  r2.answer( );  // 通过r2只能访问接口Callable曾经定义过的成员

// 通过Positionable接口引用变量访问子类对象
Positionable  r3 = cw;  // 接口Positionable的引用变量r3，引用子类对象cw
r3.showPosition( );     // 通过r3只能访问接口Positionable曾经定义过的成员
```

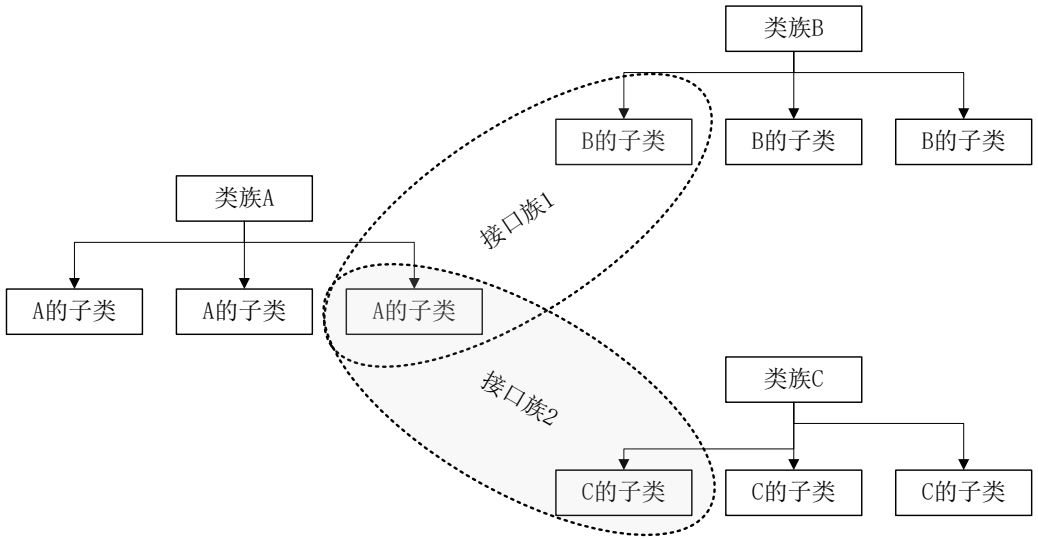
4.5 抽象类与接口

- 接口
  - 接口的应用
    - 让同一接口族共用算法代码
    - 一个接口可以被多个类实现，这些类就构成一个具有统一接口的类族，称为**接口族**



4.5 抽象类与接口

- 类族与接口族
  - 类族：类族中，各个类之间有紧密的类属关系。利用对象替换与多态机制，同一类族中的超类和各子类之间可以共用算法代码
  - 接口族：接口为类的组织管理又提供了一种新的形式。接口族中，各个类都实现了同样的接口，但它们之间的关系是松散的，而且没有类属关系。利用对象替换与多态机制，同一接口族中的各个类之间也可以共用算法代码
  - 类族与接口族：一个类只能在一个类族中，因为类只能单继承。但一个类可以在多个接口族中，因为它能同时实现多个接口



4.6 四种特殊的类定义形式

- 内部类

  可以将一个类定义在另一个类的内部。定义在其他类内部的类被称为**内部类**（inner class）。反过来，包含其他类的类被称为**外部类**（outer class）。

  - 内部类是外部类的一个成员，可以访问所在外部类的私有成员
  - 内部类可以使用类成员所特有的权限（private、protected）进行管理
  - 在外部类之外的地方使用内部类，需通过外部类对象才能使用

4.6 四种特殊的类定义形式

- 内部类

例4-15 一个包含内部类B的外部类AwithInner示例代码（AwithInner.java）

```
1  public class AwithInner { // 外部类
2      public  int x = 10;
3      private  int y = 20; // 私有成员y
4
5      public class B { // 内部类
6          public  int z = 30;
7          public  void bShow() {
8              System.out.println( x ); // 内部类可以直接访问外部类的成员x
9              System.out.println( y ); // 内部类可以访问外部类的私有成员y
10             System.out.println( z );
11     } }
12
13     public void aShow() {  // 在外部类中使用内部类B
14         B bObj = new B(); // 与使用普通的类一样
15         bObj.bShow();
16 } }
```

```
AwithInner aObj = new AwithInner();
AwithInner.B bObj = aObj.new B();
bObj.bShow();
```

4.6 四种特殊的类定义形式

- 内部类
  - 内部类可以应用于如下的场合
    - 当只被某一个类使用的时候，可以将类定义成该类的内部类，并将内部类的访问权限设为private（私有权限）
    - 当希望访问某个类的私有成员时，可以将类定义成该类的内部类
    - 当希望将若干个具有关联关系的类分成一组进行管理时，可以将这些类集中定义到某个外部类中

4.6 四种特殊的类定义形式

例4-16 一个包含局部类B的外部类AwithLocal示例代码（AwithLocal.java）

```
1  public class AwithLocal { // 外部类，其方法aMethod中包含一个局部类B
2     private int a = 10;   // 外部类的私有成员a
3     public void aMethod( int x) { // 方法中的形参x
4        final int y = 30;       // 方法中的局部只读变量（或称常量）y
5
6        class B {           // 定义在方法aMethod中的局部类
7           int b = 40;       // 局部类的成员b
8           void showA( )    // 访问外部类的成员a
9           { System.out.println( a ); }
10          void showXY( )   // 访问方法中的形参x和局部只读变量y
11          { System.out.println( x +" and " +y ); }
12          void showB( )    // 访问本类的成员b
13          { System.out.println( b ); }
14       }
15
16       B obj = new B( ); // 创建局部类B的对象obj
17       obj.showA( ); obj.showXY( ); obj.showB( ); // 调用对象obj的方法
18 } }
```

4.6 四种特殊的类定义形式

- 局部类
  - 局部类是一种特殊的类
    - 局部类访问外部类的成员时不受权限控制
    - 局部类可以访问所在方法的形参和局部变量，但要求被访问的形参和局部变量是常量（final）或事实常量（effectively final，数值在方法执行过程中不会改变）
    - 方法中的局部类和局部变量一样，不能（也不需要）设定访问权限
    - 局部类主要用于在某个方法内部使用，其他地方不需要使用这个类

4.6 四种特殊的类定义形式

- 局部类
  - 内部类、局部类可以继承超类或实现接口

例4-17 一个实现接口的局部类B示例代码（在4-16的AwithLocal.java基础上修改而来）

```
1  public class AwithLocal { // 外部类，其方法aMethod中包含一个局部类B
2    private int a = 10;   // 外部类的私有成员a
3    public void aMethod( int x) { // 方法中的形参x
4      final int y = 30;      // 方法中的局部只读变量（或称常量）y
5
6      class B implements IShow { // 实现接口IShow的局部类B
7        int b = 40;          // 局部类的成员b
8        public void show( )    // 实现接口IShow里的抽象方法show
9        { System.out.println( a +" and " +b ); }
10     }
11
12     B obj = new B( ); // 创建局部类B的对象obj
13     obj.show( );     // 调用对象obj的方法show
14 } }
15
16 public interface IShow {  // 接口IShow
17   public void show( ) ; // 定义了一个抽象方法show
18 }
```

4.6 四种特殊的类定义形式

- 匿名类
  - 当只被某个类中的某个方法使用的时候，可以将类定义成该方法中的**局部类**
  - 如果这个局部类继承某个超类或实现某个接口，并且在方法中仅仅被使用一次，则可以省略局部类的类名，这就是匿名类（anonymous class）
  - 匿名类必须继承某个超类或实现某个接口，在定义匿名类引用变量或创建匿名类对象时需使用这个超类或父接口的名字

4.6 四种特殊的类定义形式

- 匿名类

例4-18 一个实现接口的匿名类示例代码（在4-17的AwithLocal.java基础上修改而来）

```
1  public class AwithLocal { // 外部类，其方法aMethod中包含一个局部类B
2    private int a = 10;   // 外部类的私有成员a
3    public void aMethod( int x) { // 方法中的形参x
4      final int y = 30;      // 方法中的局部只读变量（或称常量）y
5
6        IShow obj = new IShow( ) { // 用一条语句完成定义匿名类和创建对象的工作
7          int b = 40;         // 匿名类的成员b
8          public void show( )    // 实现接口IShow里的抽象方法show
9          { System.out.println( a +" and " +b ); }
10       };
11     obj.show( );    // 调用对象obj的方法show
12 } }
13
14 public interface IShow {  // 接口IShow
15   public void show( ) ; // 定义了一个抽象方法sho
16 }
```

```
( new IShow( ) { // 同时省略类名和对象名
    int b = 40;
    public void show( )
    { System.out.println( b ); }
} ) . show( ) ;
```

4.6 四种特殊的类定义形式

- 匿名类
  - 匿名类是一种特殊的类
    - 匿名类必须继承某个超类或接口，创建匿名类对象时使用其超类名或父接口名

      new 超类名或父接口名( ) **{** 匿名类定义代码 **}**;

      IShow obj = **new IShow**( ) { …… };

    - 匿名类只能继承一个超类，或只能实现一个接口

    - 匿名类没有类名，无法定义构造方法

4.6 四种特殊的类定义形式

- 匿名方法
  - 如果一个接口只包含一个抽象方法，则这样的接口被称为**功能接口**（functional interface）
  - 如果一个类只实现了某个功能接口，并且没有再定义任何其他成员，则该类将只包含一个方法成员，这样的类被称为**功能类**（functional class）
  - 使用功能类所创建的对象，其中只包含一个方法成员。一个**功能类对象**，其本质就是一个完成某种程序功能的方法（即函数）
  - 如果一个功能对象**只被**使用一次，则可以使用**匿名类**来简化代码

4.6 四种特殊的类定义形式

- 匿名方法

例4-19 一个功能接口IOperator和功能类AClass的示例代码

	IOperator：功能接口	AClass：实现接口IOperator的功能类
1	public interface **IOperator** { // 功能接口	class **AClass** implements **IOperator** { // 功能类
2	int **operation**(int x, int y); // 抽象方法	int **operation**(int x, int y) // 实现接口里的方法
3	}	{ return x +y; } // 加法运算
4		}

– 创建功能类AClass的对象，可以使用如下三种方法

- 正常方法

```
AClass r = new AClass( );                  // 使用类AClass创建一个功能类对象
System.out.println( r.operation(3, 8) ); // 调用对象的方法operation，显示结果：11
```

- 改用匿名类简化代码（用匿名类取代AClass）

```
IOperator r = new IOperator( ) {           // 用一条语句完成定义类和创建对象的工作
                int operation(int x, int y)  // 实现接口里的方法
                { return x +y; }             // 加法运算
              };
System.out.println( r.operation(3, 8) ); // 调用对象的方法，显示结果：11
```

- 改用匿名方法进一步简化代码（用匿名方法取代匿名类）

```
IOperator r = (int x, int y) ->   // 省略了匿名类中的运算符new、接口名、方法名及其类型
              { return x +y; } // 只保留方法的形参列表和方法体
System.out.println( r.operation(3, 8) ); // 调用对象的方法，显示结果：11
```

4.6 四种特殊的类定义形式

- 匿名方法
  - Java语言将省略掉**方法名**和**返回值类型**的方法称为**匿名方法**（anonymous method）
  - 匿名方法只保留了方法的**形参列表**和**方法体**，并在形参列表和方法体之间插入一个指向运算符“**->**”
  - 匿名方法看起来像是一个表达式，术语称为“**Lambda**表达式”
- 实际Java编程中，程序员在创建**一次性**使用的对象时，常常喜欢以**匿名类**或**匿名方法**的形式来简化程序代码

第4章 面向对象程序设计之二

- 本章学习要点
  - 学会使用**组合**和**继承**的方法来定义新类，这样可以提高类代码的开发效率
  - 应从提高算法代码重用性的角度去理解对象的**替换**与**多态**机制
  - 熟练掌握**接口**的定义和实现方法，并充分理解接口与超类的区别
  - 熟练掌握**匿名类**和**匿名方法**的简写形式

学习预告

- Java语言经过二十多年的发展，已经积累了大量编写好的、可实现各种不同功能的类。这些类是以类库的形式提供给我们使用的，例如，
  - Java语言本身提供的**基本类库**（类似于C语言的系统函数）
  - Java语言提供的**扩展类库**（更多高级功能）
  - 第三方提供的**开源类库**

- 类库相当于是已经编写好的**程序零件**。重用类库中的类，相当于是用现成的零件来**组装**程序，这样就能快速开发出功能强大的软件，这就是程序的应用开发

即将进入程序**应用开发**