面向对象(类)

看待问题的方式:
1．里面有什么东西?
2．每个东西看上去是什么样的?
3．每个东西能做点什么用?
4．这些东西待在什么地方?
5．这些东西之间有什么关系?
6．这些东西是怎么完成任务的?
事物关系:

1. 整体-部分的关系(聚集关系)
2. 笼统-具体关系(继承关系)
3. 伙伴关系(关联关系)

类

- 把具有*相同数据和相同操作*的一组相似对象归为一"类"
- 在面向对象的软件技术中，把具有相同数据和相同操作的一组相似对象也归为一类
- 实例就是由*某个特定的类*所描述的*具体的对象*(instance = object)

类是一类事物的抽象

- 先注意问题的本质与描述，其次是实现过程或细节
- 数据抽象：描述某类对象的属性或状态
- 代码抽象：描述某类对象的共有的行为特征或具有的功能
- 抽象的实现：通过类的声明

类是一种封装(encapsulation)

将抽象出的数据成员、代码成员(实现操作的代码)相结合，将它们视为一个整体，放在对象内部
目的是*增强安全性和简化编程*，使用者不必了解具体的实现细节，只需要通过外部接口，以特定的访问权限，来使用类的成员

条件:
1．有清晰的边界
2．有确定的接口
3．受保护的内部实现

Java类的定义

类是组成Java程序的基本要素。它封装了一类对象的状态和方法，是这一类对象的原型

```
[类的修饰字] class 类名称 [extends 父类名称][implements 接口名称列表]
{
    变量定义及初始化;
    方法定义及方法体;
}
```

```
public class Circle{
   private int x, y, r;
   public void setCenter(int x, int y) { this.x = x; this.y=y;}
   public void SetRadius(int r) {this.r = r; }
   public int getRadius() { return r; }
   public void draw() {System. Out.println("Circle:(" + x + ","+y+")");
}
```

类的修饰符

访问控制修饰符：`public`，`default`
抽象类修饰符：`abstract`
最终类修饰：`final`(不能继承)

类名称

- 以字母、字符"-" 或"$"开头
- 只能含有大于十六进制00C0以上的Unicode字符
- 不能使用与Java关键字相同的类名

- 类名通常以大写字母开头，如果类名由多个单词组成，则每一个单词的开头字母也大写

完整类名

- 如果多个包中包含相同的类名，如果引起混淆，需要包括包名进行区分，这就叫做完整限定类名
  - edu.bnu.zh.bookstore.Book
  - edu.bnu.bj.hotel.Book
- 如果类位于默认包中，完整限定类名就是类名
- 包名有助于划分和组织Java应用中的各个类
  - edu.bnu.bj.courses
  - edu.bnu.bj.lib
  - edu.bnu.bj.life
- 不在同一个包中的类，需要先import再使用
  - import edu.bnu.zh.bookstore.Book效率高于import edu.bnu.zh.bookstore.*
  - 虽然“.”表示包的层次结构，但import父包不会自动引入其子包
    - import edu.bnu.*;
    - import edu.bnu.bj.*;
    - import edu.bnu.bj.lib.*;

类体

类体中定义了该类所有的变量（属性）和该类所支持的方法，通常变量在方法前定义（不一定要求）

```
类声明 {
   成员变量定义;
   构造函数定义;
   成员方法定义;
}
```

```
public class Circle{
   private int x, y, r;
   public void setCenter(int x, int y) { this.x = x; this.y=y;}
   public void SetRadius(int r) {this.r = r; }
   public int getRadius() { return r; }
   public void draw() {System. Out.println(“Circle:”);
}
```

域的声明

域，即类的属性或变量

◈ 域，也就是类的属性或变量。声明域的格式为：

```
[域修饰符]类型 变量[= 初始值][,变量];
```

◈ 其中，"[]"内的内容为可选项，当初始值可能是一个表达式，也可能是一个对象。

```
public class Circle{
   private int x=0, y=0, r=1;

   public void setCenter(int x, int y) { this.x = x; this.y=y;}
   public void SetRadius(int r) {this.r = r; }
   public int getRadius() { return r; }
   public void draw() {System. Out.println("Circle:");
}
```

变量的作用域

- 作用域首先指的是**变量的存在范围**，只有在这个代码块内的程序能访问到它
- 作用域决定了变量的**生命周期**，即变量什么时候创建并分配空间开始，到变量销毁并清除内存的过程
- 当**变量被定义**时，它的作用域就被确定了，按作用域划分
  - 成员（全局）变量，可以方法在前，声明在后
  - *局部变量*
    - 方法或构造器参数，能且只能在*整个方法或整个构造器*内使用
    - 方法或构造器*内部声明的变量*，能且只能在整个方法或整个构造器内使用
    - 方法或构造器代码块中声明的变量，只能在代码块内部使用，如`for`循环中定义的控制变量
    - 必须**先声明再使用**
  - *变量可以重名，但作用域必须不同*。如成员变量和方法参数同名，不同方法中的参数同名

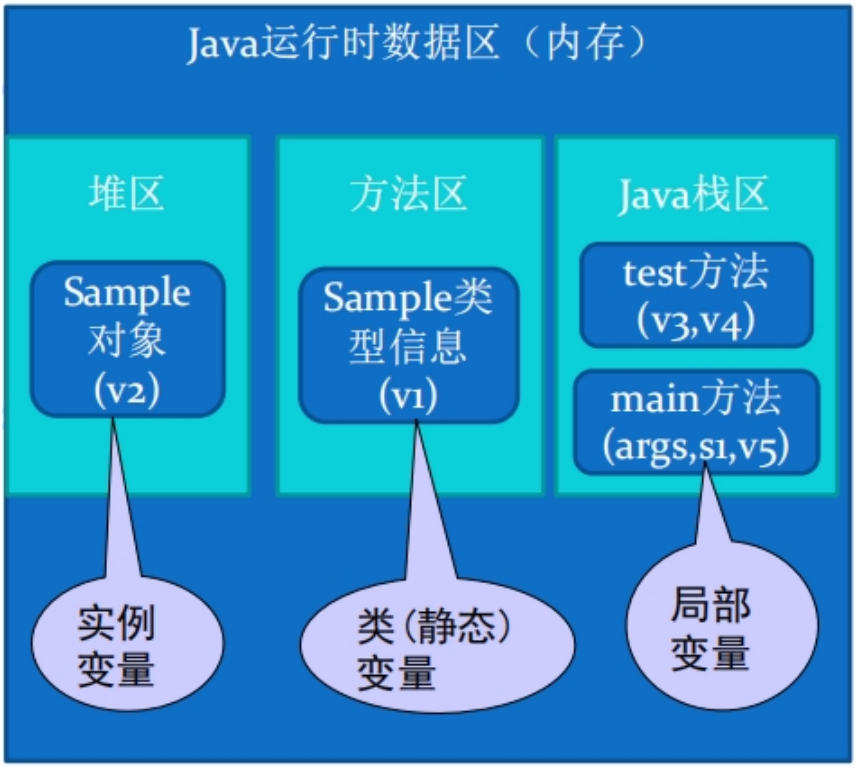
变量的生命周期

- 方法的**参数**和**局部变量**只有*当方法被调用时才存在*
- 实例变量的生命周期依附于实例变量
- **静态变量**的生命周期依附于**类的生命周期**

变量的类型与缓存区

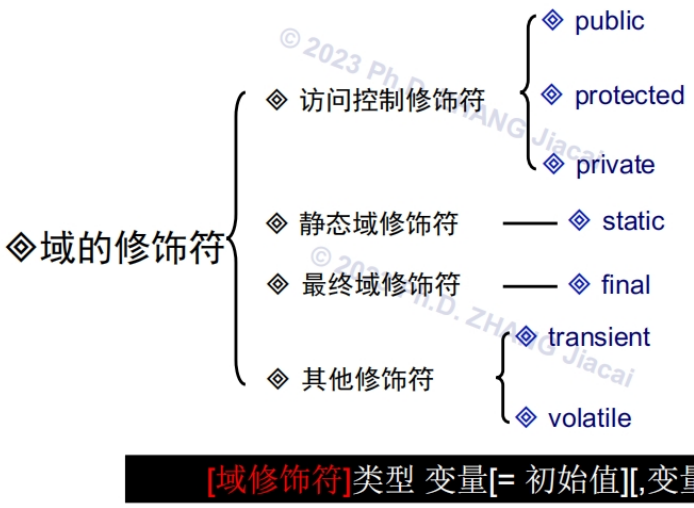
```java
public class Sample{
        static int v1; // 静态变量
        int v2; // 实例变量

        public void test(int v3){
                int v4 = v3 + 1;
                v1 = v4;
        }
        public static void main(String args[]){
                int v5 = 2;
                Sample s1 = new Sample();
                s1.test(v5); // 调用完test方法后，v3,v4即被释放
                Syatem.out.println(v1 + "," s1.v2); // print后main方法结束，args,s1,v5被释放
        } // 触发垃圾回收机制，v2无访问，被释放
} // JVM卸载Sample类，静态变量v1随之释放
```



域修饰符

域的声明——域修饰符



[域修饰符]类型 变量[= 初始值][,变量];

方法声明

◈ 方法声明了可以被调用的代码，传递固定数量的参数。方法声明的格式为：

方法修饰符 结果类型 方法名([参数列表]) throws子句{方法的体}

这里还没学哎

- 例：

```
final void move(int dx, int dy) throws IOException{
    x += dx;
    y += dy;
}
```

Java允许相同方法名但参数列表不同的方法存在

声明命名

同名方法要求返回类型一样，但参数类型可以不同

◈ 方法声明了可以被调用的代码，传递固定数量的参数。方法声明的格式为：

方法修饰符 结果类型 方法名([参数列表]) throws子句{方法的体}

这里还没学哎

- 例：

```
final void move(int dx, int dy) throws IOException{
    x += dx;
    y += dy;
}
```

方法修饰符

方法的体

- 方法体是对方法的实现。它包括局部变量的声明以及所有合法的Java指令。局部变量的作用域只在该方法内部
- 如果方法提供实现，但是实现部分可以不要求任何可执行的代码，方法的体还是应该当作一个语句块写出，即“{}”
  - 如果方法被声明为void，那么，方法的体中就可以含有return语句，但return后面不能有表达式
  - 如果方法的声明中返回类型不是void，则方法的体中必须含有return expression 语句
- main方法在Java应用程序中表示程序执行的起
- main方法是Java应用程序中惟一与用户直接交换信息的方法，必须声明为public static和void，不能返回任何值。它的定义格式如下：

```
public static void main(String args[]) {方法体}
```

面向对象的封装

- 大部分情况下，类中的成员变量定义为私有属性（private）
  - 如果该属性允许外界访问，一般也不会改变属性的修饰符，而是提供公共方法接口来操作属性
  - 封装就是透明

    这里的透明就是你知道它存在，但无法检视它，看不见不是被遮挡，而是透明

```
private int volume;

public void setVolume(int volume){this.volume = volume;}
public int getVolume(){return volume;}
```

外部只能读、不能写的属性

```
private int powerMeter;

private void setPowerMeeter(double powerMeter){
    this.powerMeter = powerMeter;
}
public double getPowerMeeter (){return powerMeter;}
```

```
public void setPassword(String passwird){
    if (password==null || password.length != 6)
        System.out.println("Error password");
    } else{
        this.password = password;
    }
}
```

防止使用者不当修改属性，或者属性修改有连锁反应要处理

面向对象的思维

对象创建

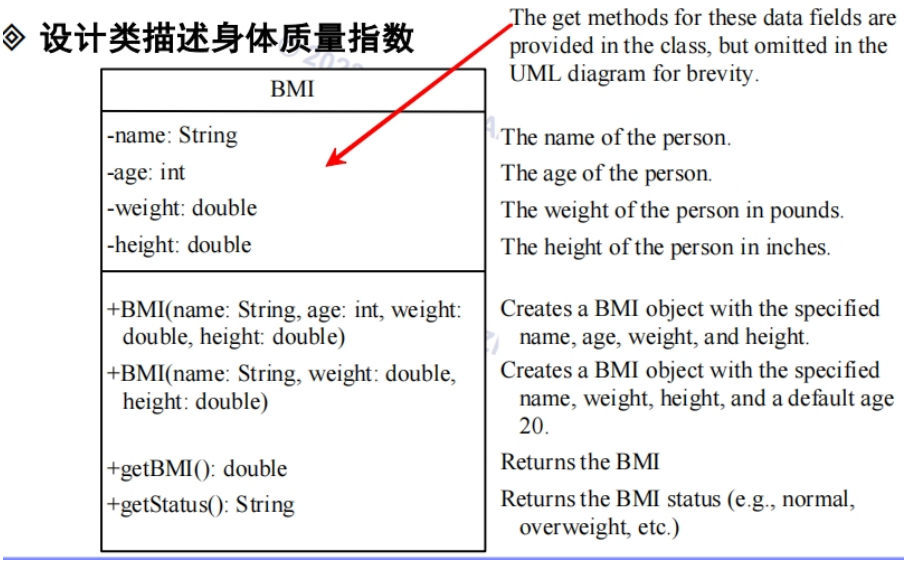
使用new方法

类名 实例名 [=new 类的构造器]

```
Point origin_one = new Point(23,94);
Rectangle rect_one= new Rectangle(origin_one,100,200);
Rectangle rect_two=new Rectangle(50,100);
```

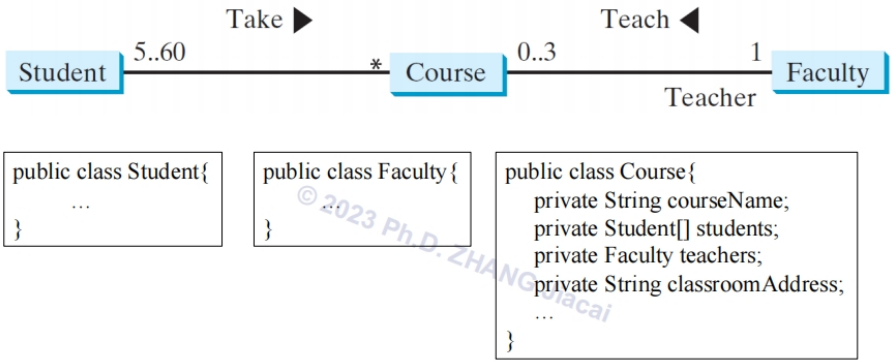
UML图

◈ 设计类描述身体质量指数



+表示public，-表示private，上部分放属性，下部分放方法

◈ 考虑类描述课程（Course）



`5...60`表示5-60学生选课，且`*`代表至少有1位
`0..3`表示有0-3门课会由1位老师教授

Random类

- 随机数生成：
  - `Math.random():[0.0 - 1.0)`
  - `java.util.Random`

  随机数实际上是伪随机，给定特定的种子，生成的随机数就是一样的

JAVA

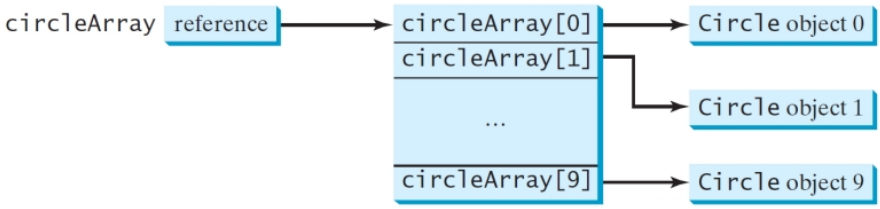
```java
import java.util.Random;
public class CreateRandom {
        public static void main(String[] args) {
                int a[] = new int[10];

                Random random1 = new Random(3);
                for (int i = 0; i < a.length; i++) {
                        a[i] = random1.nextInt(1000); // 生成1000以内的随机数
                }
                for (int i = 0; i < a.length; i++) {
                        System.out.print(a[i] + " ");
                }
                System.out.println("");

                Random random2 = new Random(3);
                for (int i = 0; i < a.length; i++) {
                        a[i] = random2.nextInt(1000);
                }
                for (int i = 0; i < a.length; i++) {
                        System.out.print(a[i] + " ");
                }
        }
}
```

对象数组

- 对象数组实质上是**引用变量的数组**
- 对象数组至少有两重引用



JAVA

```java
Circle[] circleArray = new Circle[10];
circleArray[0] = new Circle();
circleArray[0] = new Circle();
```

- `circleArray`引用整个数组对象，其中存储*10个引用地址*
- 这10个地址`circleArray[0]... circleArray[9]`：初始化为`null`
- `circleArray[0].getArea()`，将会调用一个`Circle`对象的方法

创建不被修改的变量

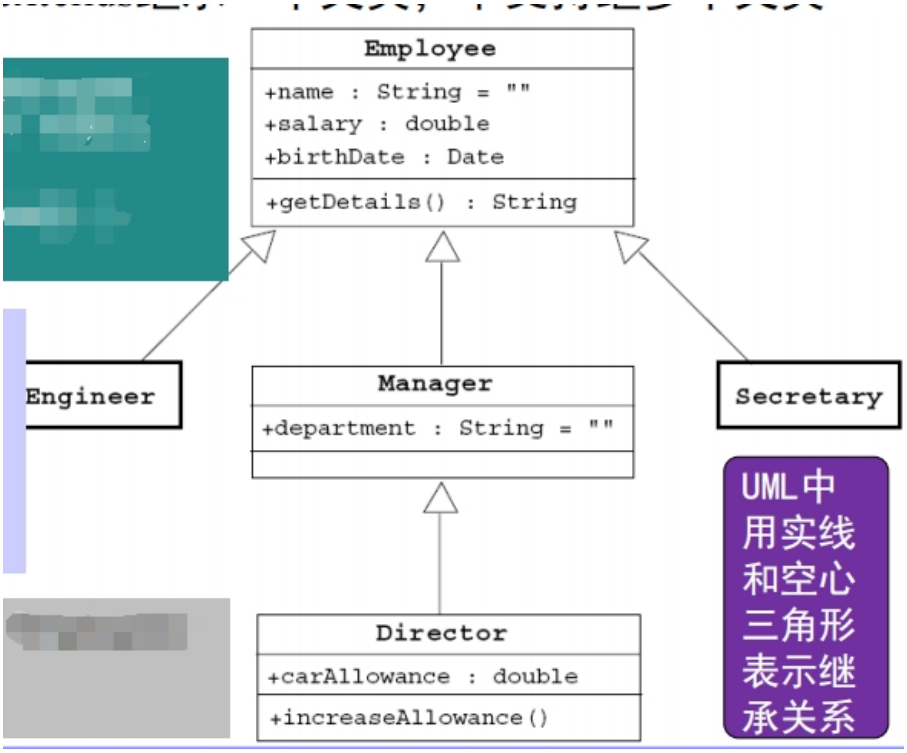
- 有些对象一旦创建就不想修改其属性

子类

`class Sub extends Base`

Java语言允许使用`extends`继承一个父类，**不支持继承多个父类**

在UML中**用实线和空心三角形**表示继承关系



JAVA

```
public class Employee{
        public String name = "";
        public double salary;
        public Date birthday;

        public String getDetail(){}
}
public class Manager extends Employee{
        public String department;
}
public class Director estends Manager{
        public double carAllowance = 1000.0;
        ...
        public void increaseAllowance(){...}
}
```

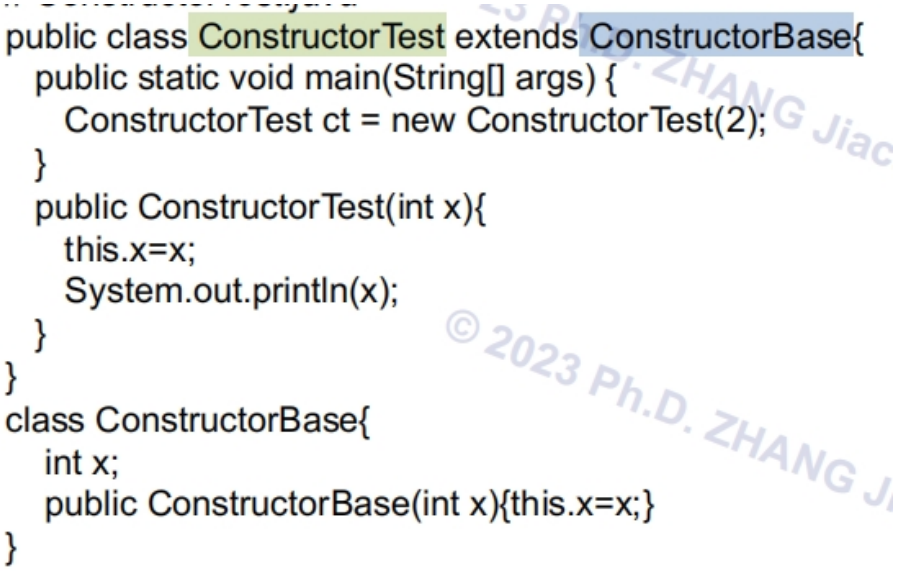
使用继承

- 子类与父类关系：is a relationship
- 定义方法：`extends`
- **构造器不会继承**
  - 子类构造器中虽然没有继承父类的构造器，但**可以使用**父类中*非private的构造器*，具体语法是构造器第一行使用`super()`语句
  - 构造器中**首行**中的`super()`*调用父类的构造器*，调用时可以**指定参数**，如果父类中没有参数相同的构造器，则编译错误

- 如果子类构造器中首行**没有调用**父类的构造器，系统**隐含插入没有参数的**`super()`**语句**，如果父类中没有空参数的构造器，则编译错误
- `this()`调同类构造器，`super()` 调父类构造器

> 在Java中，子类继承父类的属性和方法。当创建一个子类对象时，该对象包含父类和子类的所有属性和方法。构造函数用于初始化对象的状态。
> 如果子类没有显式地调用父类的构造函数，Java编译器会默认尝试**调用父类的无参构造函数**。如果父类**没有无参构造函数**，并且子类的构造函数**没有显式地调用**父类的有参构造函数，那么**编译器会报错**，因为父类的状态没有被正确地初始化。
> 通过显式地调用父类的构造函数，你可以确保父类的状态被正确地初始化，从而确保子类对象的正确性和完整性。

错误例子，由于子类构造器会隐式调用父类构造器，而父类没有空构造器，所以会编译错误：



JAVA

```java
// 改正后
public class Constructor entends ConstructorBase{
        public static void main(String[] args){
                ConstructorTest ct = new ConstructorTest(2);
        }
        public ConstructorTest(int x){
                // super(); 隐式调用了父类的空构造器
                this.x = x;
                // super(x); 不用this的写法
                System.out.println(this.x);
        }
}

class ConstructorBase{
        int x;
        public ConstructorBase(int x){this.x = x;}
        public COnstructorBase(){} // 创建一个空构造器便于子类调用
}
```

- 方法和变量可以继承，但要看权限，**父类私有成员不继承**

构造器不继承

- 关键字super
  - 父类的变量或方法
    - super.memberVariable
    - super.memberMethod()
  - this()调同类构造器
  - super()调父类构造器
    - 如果子类构造器中首行没有调用同类this()或父类的构造器super()语句，系统隐含插入没有参数的super()语句
    - 如果父类中没有空参数的构造器，则编译错误。

```
public class Manager extends Employee{
  private String department;
  public Manager(String name, double salary, String dept){
    super(name, salary);  this.department = dept;
  }
  public Manager(String name , double sal, String dept) {
    super(name);  this.department = dept; this.salary=sal;
  }
  //以上两个构造器实现同样功能，代码中保留1个
  public Manager(String name, String dept){
    this(name,10000, dept);
    this.department = dept;
  }
  public Manager(String dept){
    this.department = dept;
  }
}
```

```
Employee(String Name, Bouble salary){}
Employee(String Name){}
Employee(){}
```

例子

JAVA

```java
package test1;

public class InheritanceTest {
        public static void main(String[] args) {
                new SubA(1);
                SubA sa = new SubA();
                sa.print();
                new SubB();
        }
}

class SuperA{
        private int j = 10;
        public SuperA(int i) {
                System.out.println("constructor SuperA(int)");
        }
        public SuperA() {
                System.out.println("constructor SuperA()");
        }
        void print() {
                System.out.println("print() in SuperA");
        }
}

class SubA extends SuperA{
        private int k = 2;
        public SubA(int i) {
                super(i);
                System.out.println("constructor SubA(int)");
        }

        public SubA() {
        // 系统会自动插入super();
                System.out.println("constructor SubA()");
        }

        void print() {
                super.print(); // 方法重写后，可以通过这种方式访问父类的方法
                System.out.println("print() in SubA");
        }
}

class SubB extends SuperB{
        public SubB() {
                super("Hello");
                System.out.println("constructor subB()");
        }
}
```

```
class SuperB{
        public SuperB(String s) {
                System.out.println("constructor superB(string):" + s);
        }
}
```

```
constructor SuperA(int)
constructor SubA(int)
constructor SuperA()
constructor SubA()
print() in SuperA
print() in SubA
constructor superB(string):Hello
constructor subB()
```

访问修饰符的控制范围

访问修饰符	本类	同一包中的类	不同包中的类
public	☺	☺●	☺●
protected	☺	☺●	●
缺省	☺	☺●	
private	☺		

☺: 可访问

● 可继承

注意：

```
class A{
  private int x;
  public A(int x){this.x = x;}
  public void accessPrivate(A object){
    System.out.println(object.x);
  }
}


public class Test {
  public void accessPrivate(A object){
    System.out.println(object.x);
  }

  public static void main(String args[]){
    A o1 = new A(1); A o2 = new A(2);
    o1.accessPrivate(o2);  //输出2
    o2.accessPrivate(o1);  //输出1
    Test t = new Test();
    t.accessPrivate(o2); // 无法访问，因为不是同一类
  } // 说明即使设置了private也可能被访问
}
```