

访问修饰符

java的四个关键字：

public、protected、default、private

（他们决定了紧跟其后被定义的东西可以被谁使用）

适用范围<访问权限范围越小，安全性越高>

访问权限	类	包	子类	其他包	
public	√	√	√	√	（对任何人都是可用的）
protected	√	√	√	×	（继承的类可以访问以及和 private 一样的权限）
default	√	√	×	×	（包访问权限，即在整个包内均可被访问）
private	√	×	×	×	（除类型创建者和类型的内部方法之外的任何人都不能访问的元素）

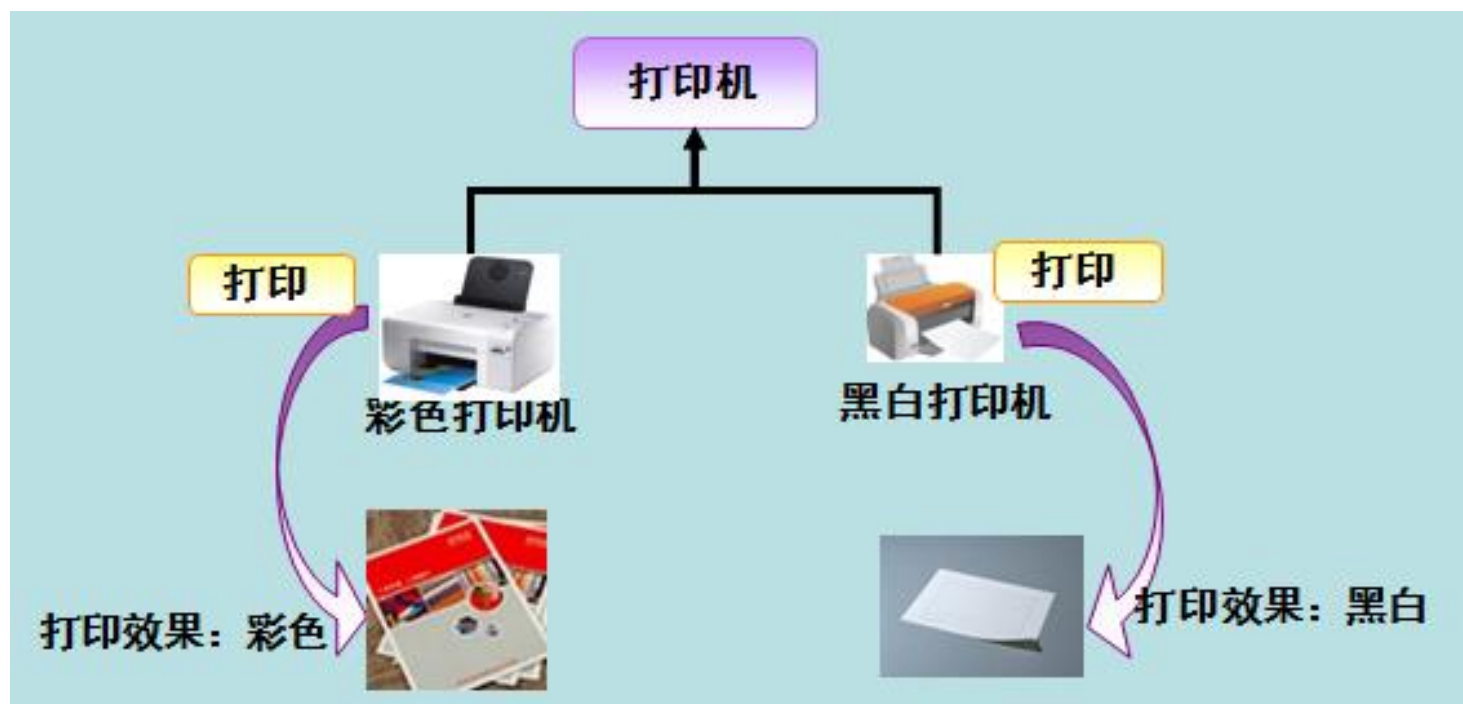
给出如下代码:

```
Class Test{  
    private int m;  
    public static void fun() {  
  
        // some code...  
  
    }  
}
```

如何使成员变量m 被函数fun()直接访问? (C)

- A、将private int m 改为protected int m
- B、将private int m 改为 public int m
- C、将private int m 改为 static int m
- D、将private int m 改为 int m

多态是同一个行为具有多个不同表现形式或形态的能力。
多态就是同一个接口，使用不同的实例而执行不同操作，如图所示：



多态存在的三个必要条件

- 继承
- 重写
- 父类引用指向子类对象

比如：

```
Parent p = new Child();
```

阅读下面的代码：

```
public class Test {  
    public static void main(String[] args) {  
        show(new Cat()); // 以 Cat 对象调用 show 方法  
        show(new Dog()); // 以 Dog 对象调用 show 方法  
  
        Animal a = new Cat(); // 向上转型  
        a.eat();                // 调用的是 Cat 的 eat  
        Cat c = (Cat)a;        // 向下转型  
        c.work();              // 调用的是 Cat 的 work  
    }  
  
    public static void show(Animal a) {  
        a.eat();  
        // 类型判断  
        if (a instanceof Cat) { // 猫做的事情  
            Cat c = (Cat)a;  
            c.work();  
        } else if (a instanceof Dog) {  
            Dog c = (Dog)a;  
            c.work();  
        }  
    }  
}
```

```
abstract class Animal {  
    abstract void eat();  
}  
  
class Cat extends Animal {  
    public void eat() {  
        System.out.println("吃鱼");  
    }  
    public void work() {  
        System.out.println("抓老鼠");  
    }  
}  
  
class Dog extends Animal {  
    public void eat() {  
        System.out.println("吃骨头");  
    }  
    public void work() {  
        System.out.println("看家");  
    }  
}
```

吃鱼

抓老鼠

吃骨头

看家

吃鱼

抓老鼠

```
public class Animal{  
    public String name;//名称  
    public int age;//年龄  
  
    public Animal(){  
        this.age=100;  
        System.out.println("动物无参构造函数！");  
    }  
    //展示动物属性方法  
    public void showInfo(){  
        this.name="动物名";  
        System.out.println("Animal.name:"+this.name);  
    }  
}
```

```
public class Person extends Animal {  
    public String name;//名称  
    public Person() {  
        System.out.println("人无参构造函数！");  
    }  
    //展示（动物）人类属性方法  
    @Override  
    public void showInfo(){  
        super.showInfo();//调用父类的方法，给Animal的name赋值。  
        this.name="人名";  
        System.out.println("Animal.age:"+super.age);//输出父类  
        System.out.println("Person.name:"+this.name);//输出Person  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Person p=new Person();  
        p.showInfo();  
    }  
}
```

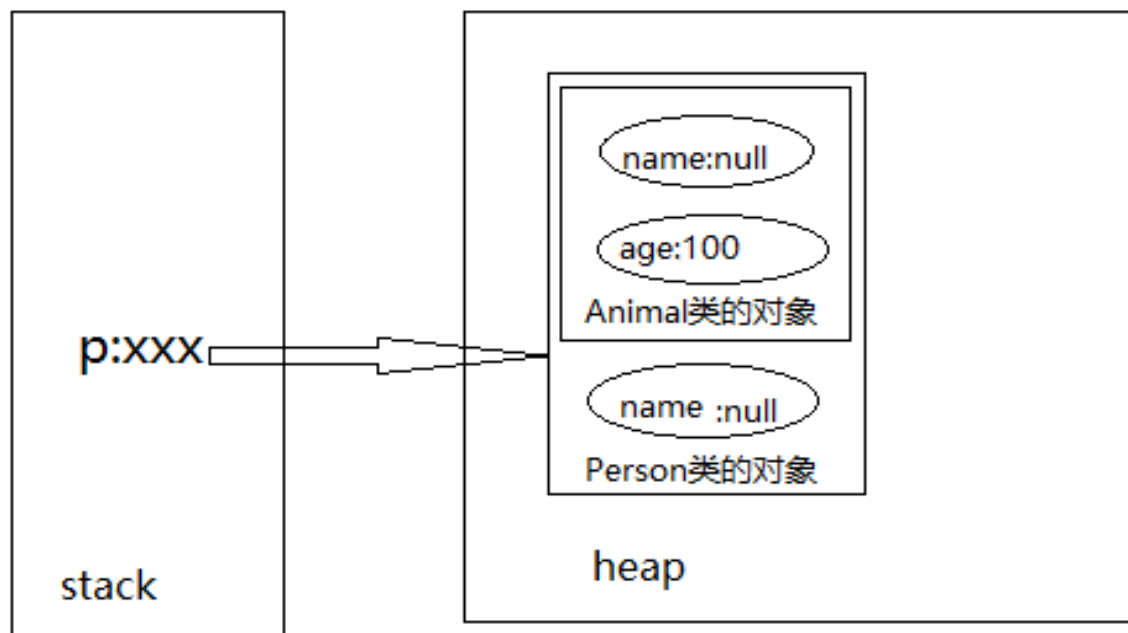
运行结果：

```
动物无参构造函数！  
人无参构造函数！  
Animal.name:动物名  
Animal.age:100  
Person.name:人名
```

```
Person p=new Person();
```

先在栈（stack）空间中产生一个变量p，变量p里面存放的值是，Person类new的实例化对象的堆地址，通过这个值就可以找到堆中new对象的值，因为Person类是继承Animal父类的子类，所以在new Person对象时，这个对象里包含了Animal对象，Animal的name属性声明是没有初始化值，在程序编译时，会默认赋值null（String类型为null,int类型是0），Animal的age属性，在调用Person类的构造方法时，会先调用父类的无参构造函数，并给age赋值为100，同理Person类对象自己的属性name值为null，因此在执行Person p=new Person();之后，虚拟机中的内存分配

如下：



•父类引用指向子类对象---- 成员变量的实际取值

```
package test.xing;

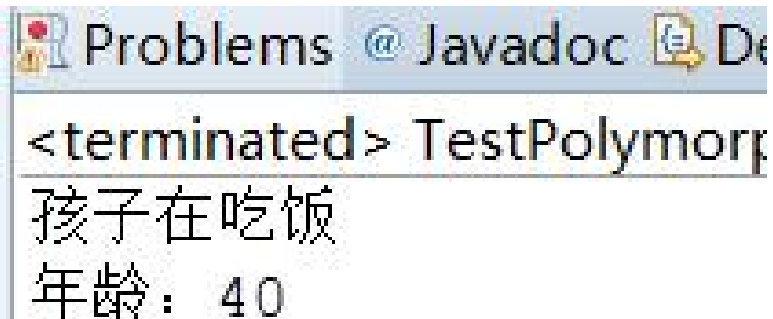
class Father{
    protected int age;
    public Father(){
        age = 40;
    }

    void eat(){
        System.out.println("父亲在吃饭");
    }
}

class Child extends Father{
    protected int age;
    public Child(){
        age = 18;
    }

    void eat(){
        System.out.println("孩子在吃饭");
    }
    void play(){
        System.out.println("孩子在打CS");
    }
}
```

```
public class TestPolymorphic {
    public static void main(String[] args) {
        Father c = new Child();
        c.eat();
        //c.play();
        System.out.println("年龄: "+c.age );
    }
}
```



Problems @ Javadoc De

<terminated> TestPolymorp

孩子在吃饭

年龄: 40

属性/变量不存在多态!

部分作业评讲：

```
3. import java.io.FileInputStream;
import java.io.FileOutputStream;
public class FileTest {
    public static void main(String[] args) throws Exception {
        int data1 = -2;
        FileOutputStream fout = new FileOutputStream("f1.txt");
        fout.write(data1);
        fout.write(new byte[] { -1, -2 });
        fout.close();
        FileInputStream fin = new FileInputStream("f1.txt");
        int data2 = fin.read();
        byte[] b = new byte[4];
        int count = fin.read(b);
        fin.close();
        for (int i : b) {
            System.out.println(i);
        }
        System.out.println(count);
        System.out.printf("data1=%d, data2=%d", data1, data2);
    }
}
```

-1
-2
0
0
2

data1=-2, data2=254



读第一个字节“11111110”，赋值给data2，高24位使用0补足（此时data2为254）

```
byte[] b = new byte[4];  
int count = fin.read(b);
```



依次读出f1.txt后面2个字节的内容，-1，-2分别赋值给b[0],b[1].
Count值为2

```
for (int i : b) {  
    System.out.println(i);  
}  
System.out.println(count);  
System.out.printf("data1=%d, data2=%d", data1, data2);
```

-1
-2
0
0
2

data1=-2, data2=254

8.

```
class OverrideTest {
    void show() {
        System.out.println("super show");
        System.out.println(this.getClass().getName());
        this.getName();
    }
    void getName() {
        System.out.println("OverrideTest");
    }
}

public class SubOverride extends OverrideTest {
    void show() {
        System.out.println("sub show");
        super.show();
        this.getName();
    }
    void getName() {
        System.out.println(this.getClass().getSuperclass().getName());
    }
    public static void main(String args[]) {
        OverrideTest s = new SubOverride();
        s.show();
    }
}
```

```
sub show
super show
SubOverride
OverrideTest
OverrideTest
```

```
public static void main(String args[]) {  
    OverrideTest s = new SubOverride();  
    s.show();  
}
```



s.show() 实质调用的是子类的show() 方法



```
public class SubOverride extends OverrideTest {  
    void show() {  
        System.out.println("sub show");  
        super.show();  
        this.getName();  
    }  
}
```



输出 “sub show”，然后执行：super.show()，执行父类的show()



```
class OverrideTest {  
    void show() {  
        System.out.println("super show");  
        System.out.println(this.getClass().getName());  
        this.getName();  
    }  
}
```

输出 “super show”，然后执行System.out.println (this.getClass().getName())



this.getClass().getName(): 这里是this实际上就是子类对象，获得子类的类名SubOverride (为什么不是调用子类的getName()方法?)



输出 “SubOverride”，然后执行this.getName(); (调用子类的getName()方法)



```
void getName() {  
    System.out.println(this.getClass().getSuperclass().getName());  
}
```

输出 “OverrideTest ”，此时super.show() 执行完毕，继续子类中的下一条语句this.getName()



输出 “OverrideTest ”

异常处理

try...catch...finally语句块

```
1  public static void main(String[] args){
2      try {
3          foo();
4      }catch(ArithmeticException ae) {
5          System.out.println("处理异常");
6      }
7  }
8  public static void foo(){
9      int a = 5/0; //异常抛出点
10     System.out.println("为什么还不给我涨工资!!!"); ///////////////
11 }
```

throws是另一种处理异常的方式

它不同于try...catch...finally，throws仅仅是将函数中可能出现的异常向调用者声明，而自己则不具体处理。

采取这种异常处理的原因可能是：方法本身不知道如何处理这样的异常，或者说让调用者处理更好，调用者需要为可能发生的异常负责。

```
1 public void foo() throws ExceptionType1 , ExceptionType2 ,ExceptionTypeN
2 {
3     //foo内部可以抛出 ExceptionType1 , ExceptionType2 ,ExceptionTypeN 类的异常，或者他们的子
4     类的异常对象。
    }
```


throw 异常抛出语句

throw exceptionObject

程序员也可以通过throw语句手动显式的抛出一个异常。throw语句的后面必须是一个异常对象。

throw 语句必须写在函数中，执行throw 语句的地方就是一个异常抛出点，它和由JRE自动形成的异常抛出点没有任何差别。

```
1  public void save(User user)
2  {
3      if(user == null)
4          throw new IllegalArgumentException("User对象为空");
5      //.....
6
7  }
```

异常调用栈

```
package person.ismallboy.console;

import java.io.IOException;

public class TestEx {
    private void fun1() throws IOException {
        throw new IOException("level 1 exception");
    }

    private void fun2() throws IOException {
        try {
            fun1();
            System.out.println("2");
        } catch (IOException e) {
            throw new IOException("level 2 exception", e);
        }
    }
}
```

```
private void fun3() {
    try {
        fun2();
        System.out.println("3");
    } catch (IOException e) {
        throw new RuntimeException("level 3 exception", e);
    }
}

public static void main(String[] args) {
    try {
        new TestEx().fun3();
        System.out.println("0");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
java.lang.RuntimeException: level 3 exception
    at person.ismallboy.console.TestEx.fun3(TestEx.java:24)
    at person.ismallboy.console.TestEx.main(TestEx.java:30)
Caused by: java.io.IOException: level 2 exception
    at person.ismallboy.console.TestEx.fun2(TestEx.java:15)
    at person.ismallboy.console.TestEx.fun3(TestEx.java:21)
    ... 1 more
Caused by: java.io.IOException: level 1 exception
    at person.ismallboy.console.TestEx.fun1(TestEx.java:7)
    at person.ismallboy.console.TestEx.fun2(TestEx.java:12)
    ... 2 more
```

异常沿着调用栈查找，直到找到合适的catch语句捕获为止：

