章节21-章节30

章节21 课时 83 03064_综合实战：数组操作（基础父类）

//定义一个数组操作类Array{},在这个类里面可以进行整型数组的操作，由外部传入数组的大小，并且要求实现数据的保存和输出。随后在这个类上派生出2个子类：
1、排序类，通过此类取得的数据可以进行排序；
2、反转类，通过此类取得的数组数据要求采用倒序的方式输出。

```
class Array{
  private int data[];//定义一个数组
  private int foot;//表示数组的操作脚标
  public Array(int len){//构造本类对象时需要设置大小
    if(len>0){//至少有元素
      this.data=new int [len];//开辟一个数组
    }else{
      this.data=new int[1];//维持一个元素的大小
    }
  }
  public boolean add(int num){
    if(this.foot<this.data.length){
      this.data[this.foot++]=num;
      return true;//保存成功
    }
    return false;//保存失败
  }
  public int[] getData(){//取得所有的数组内容
    return this.data;
  }
}
public class TestDemo{
  public static void main(String args[]){
    Array arr=new Array(3);
    System.out.println(arr.add(10));
    System.out.println(arr.add(20));
    System.out.println(arr.add(30));
    System.out.println(arr.add(100));
    System.out.println(arr.add(200));
    int [] temp=arr.getData();
    for(int x=0;x<temp.length;x++){
      System.out.println(temp[x]);
    }
  }
}
```

章节21 课时 84 03065_综合实战：数组操作（排序子类）

```
class Array{
  private int data[];//定义一个数组
  private int foot;//表示数组的操作脚标
  public Array(int len){//构造本类对象时需要设置大小
    if(len>0){//至少有元素
      this.data=new int [len];//开辟一个数组
    }else{
      this.data=new int[1];//维持一个元素的大小
    }
  }
  public boolean add(int num){
    if(this.foot<this.data.length){
      this.data[this.foot++]=num;
      return true;//保存成功
    }
    return false;//保存失败
```

```
  }
  public int[] getData(){//取得所有的数组内容
    return this.data;
  }
}
class SortArray extends Array{//Array类里面现在没有无参构造方法
  public SortArray(int len){//这样父类中的data数组就可以初始化
    super(len);//明确的调用父类的有参构造
  }
  //因为父类中getData()方法不够当前类使用的
  //使用覆写的概念来将此方法的功能扩充
  public int [] getData(){
    java.util.Arrays.sort(super.getData());
    return super.getData();
  }
}
public class TestDemo{
  public static void main(String args[]){
    SortArray arr=new SortArray(3);
    System.out.println(arr.add(100));
    System.out.println(arr.add(20));
    System.out.println(arr.add(19));
    System.out.println(arr.add(100));
    System.out.println(arr.add(200));
    int [] temp=arr.getData();
    for(int x=0;x<temp.length;x++){
      System.out.println(temp[x]);
    }
  }
}
```

章节21 课时 85 03066_综合实战：数组操作（反转子类）

```
class Array{
  private int data [];
  private int foot;
  public Array(int len){
    if(len>0){
      this.data=new int[len];
    }else{
      this.data=new int[1];
    }
  }
  public boolean add(int num){
    if(this.foot<this.data.length){
      this.data[this.foot++]=num;
      return true;
    }
    return false;
  }
  public int[] getData(){
    return this.data;
  }
}
class SortArray extends Array{
  public SortArray(int len){
    super(len);
  }
  public int [] getData(){
    java.util.Arrays.sort(super.getData());
    return super.getData();
  }
}
class ReverseArray extends Array{
  public ReverseArray(int len){
    super(len);
  }
  public int [] getData(){
```

```
    int center=super.getData().length/2;
    int head=0;
    int tail=super.getData().length-1;
    for(int x=0;x<center;x++){
      int temp=super.getData()[head];
      super.getData()[head]=super.getData()[tail];
      super.getData()[tail]=temp;
    }
    return super.getData();
  }
}
public class TestDemo{
  public static void main(String args[]){
    SortArray arr=new SortArray(3);
    System.out.println(arr.add(101));
    System.out.println(arr.add(120));
    System.out.println(arr.add(30));
    int [] temp=arr.getData();
    for(int x=0;x<temp.length;x++){
      System.out.println(temp[x]);
    }
  }
}
```

章节22 课时 86 03067_final关键字

```
//在以后查看文档的时候，如果发现了final定义的类或方法时千万不要继承或覆写。
//使用public static final定义的是全局常量，全局常量每一个字母都要求大写。
//举例：public static final MSG="MLDN";
//举例：public static final MSG=100;
```

章节22 课时 87 03068_对象多态性

多态的依赖：方法的覆写

多态性严格来讲有两种描述形式：

1、方法的重载：同一个方法名称，会根据传入的参数类型和个数不同执行不同的方法体

2、方法的覆写：同一个方法会根据不同子类的覆写实现不同的功能。

对象的多态性：指的是发生在继承关系中，子类和父类之间的转换问题。

1、向上转型(自动完成)：父类 父类对象=子类实例;

2、向下转型(强制完成)：子类 子类对象=(子类) 父类实例;

范例1：

```
class A{
  public void print(){
    System.out.println("A、public void print(){}");
  }
}
class B extends A{
  public void print(){
    System.out.println("B、public void print(){}");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=new B();//向上转型
    B b=(B) a;//向下转型
    b.print();//看new的是哪个类，看方法有没有被子类覆写，调的永远是被覆写过的方法
  }
}
```

//向下转型指的是父类要调用子类自己定义的特殊方法

范例2：

```
class A{
  public void print(){
    System.out.println("A、public void print(){}");
  }
}
```

```
class B extends A{
  public void print(){
    System.out.println("B、public void print(){}");
  }
  public void funB(){
    System.out.println("B、扩充B的funB()方法");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=new B();//向上转型
    B b=(B) a;//向下转型主要是为了父类能够访问子类自己定义的方法
    a.funB();
  //以上三行代码和直接实例化子类进行调用 new B().fun(); 执行效果是一样的，那为什么还要转型折腾？
  }
}//答案请看↓
```

//以前一直强调，对于数据的操作分两步：
1、设置数据（保存数据），最需要的是参数统一功能；而转型恰好满足了这一需求。
2、取出数据

范例3：

```
class A{
  public void print(){
    System.out.println("A、public void print(){}");
  }
}
class B extends A{
  public void print(){
    System.out.println("B、public void print(){}");
  }
  public void funB(){
    System.out.println("B、扩充B的funB()方法");
  }
}
public class TestDemo{
  public static void main(String args[]){
    fun(new B());//向上转型，完整的向上转型是A a=new B();但此处和fun(A a)方法里的合起来就是。
  }
  public static void fun(A a){//定义一个特殊的方法，只允许使用一个类型的类对象，即统一参数。
    B b=(B) a;//想要调用funB()发现功能不足，所以需设置此步向下转型。调用个性化的特征
    b.funB();
  }
}
```

对于对象的转型，给出以下的经验总结：
80%的情况下都只会使用向上转型，因为可以得到参数类型的统一，方便与我们的设计；子类定义的方法大部分情况下轻易父类的方法名称为标准进行覆写，不要过多的扩充方法。例如上一个例子里的funB()方法尽量不要有。
5%的情况下会使用向下转型，目的是调用子类的特殊方法；
15%的情况下是不转型的。例如 String
个性化的操作在一个标准的开发之中应该尽量少出现，因为对象的转型操作里面毕竟有了强制问题，容易带来安全隐患。

范例4：

```
class A{
  public void print(){
    System.out.println("A、public void print(){}");
  }
}
class B extends A{
  public void print(){
    System.out.println("B、public void print(){}");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=new A();
    B b=(B) a;
    b.print();
  }
}
```

```
//编译正常通过，但是执行时出现了ClassCastException异常
//ClassCastException指的是类转换异常，两个没有关系的类对象强制发生类向下转型时所带来的异常。所以来讲向下转型是有风险的。
解决办法看↓
-------------------------------------------------------------------------------
//为了保证转型的顺利进行，在Java里面有个关键字instanceof,  此关键字的使用：对象 instanceof 类 返回boolean型
//如果某个对象是某个类的实例，就返回true，否则返回false。
范例5：
class A{
  public void print(){
    System.out.println("A、public void print(){}");
  }
}
class B extends A{
  public void print(){
    System.out.println("B、public void print(){}");
  }
}
public class TestDemo{
  public static void main(String args[]){
  A a=new A();//执行结果为 true false，此语句换成A a=new B();向上转型(即a也转为B的实例)，结果就会变为true true。
  System.out.println(a instanceof A );//对a和A类有什么关系进行判断
  System.out.println(a instanceof B );//对a和B类有什么关系进行判断
  }
}
-------------------------------------------------------------------------------
范例6：
class A{
  public void print(){
    System.out.println("A、public void print(){}");
  }
}
class B extends A{
  public void print(){
    System.out.println("B、public void print(){}");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=new B();
    System.out.println(a instanceof A );//对a和A类有什么关系进行判断
    System.out.println(a instanceof B );//对a和B类有什么关系进行判断
    if(a instanceof B){//加此句是为了避免范例4中向下转型的风险
      B b=(B) a;//向下转型
      b.print();
      //上述向下转只是实例化了一个对象b,也可以用实例化匿名对象new B().print();
    }
  }
}
总结：
//1、开发之中尽量使用向上转型，向上转型的主要功能是实现参数统一，同时只有发生了向上转型之后，才可以发生向下转型。
//2、子类中尽量不要过多扩充和父类无关的操作方法；
//3、90%的情况下，开发之中子类的方法尽量与父类的方法功能保持一致。
```

章节23 课时 88 03069_抽象类的定义及使用（基本概念）

```
abstract class A{
  public void fun(){
    System.out.println("A、public void print(){}");
  }
  public abstract void print();
}
class B extends A{
  public void print(){
    System.out.println("Hello World!");
  }
}
```

```
public class TestDemo{
  public static void main(String args[]){
  A a=new B();
  a.print();
  }
}

//抽象类继承子类里面会有明确的方法覆写要求，而普通类没有；
//抽象类只比普通类多了一些抽象方法的定义，其他组成部分与普通类完全一样；
//普通类对象可以直接实例化，但是抽象类对象必须经过向上转型之后才可以得到实例化对象。
//虽然一个子类可以继承任意一个普通类，但是从开发实际要求来讲，普通类不要去继承另外一个普通类，而只能继承抽象类。
```

章节23 课时 89 03070_抽象类的定义及使用（使用限制）

```
//1、抽象类里面由于存在一些属性，那么在抽象类中一定会存在构造方法，目的：为属性初始化，并且子类对象实例化的时候依然满足于先执行父类构造，再调用子类构造的情况。
//2、抽象类不能使用final定义：因为抽象类必须有子类，而final定义的类不能有子类。
//3、外部抽象类不允许使用static声明，而内部的抽象类允许使用static声明，使用static声明的内部抽象类就相当于是一个外部抽象类，继承时使用“外部类.内部类”的形式表示类名称。
abstract class A{
   static abstract class B{
    public void print(){
      System.out.println();
    }
  }
}
class X extends A.B{
  public void print(){
    System.out.println("************");
  }
}
public class TestDemo{
  public static void main(String args[]){
  A.B ab=new X();
  ab.print();
  }
}



//4、任何情况下，如果要执行类中的static方法时，都可以在没有对象的时候直接调用。对于抽象类也是一样。
abstract class A{
  public static void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A.print();
  }
}

//5、有时候 ，由于抽象类只需要一个特定的系统子类操作，所以可以忽略 掉外部子类。
abstract class A{//定义一个抽象类
  public abstract void print();
  private static class B extends A{//内部抽象类子类
    public void print(){//覆写抽象类的方法
      System.out.println("Hello World!");
    }
  }
  public static A getInstance(){//这个方法不受实例化对象的控制
    return new B();
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=A.getInstance();
```

```
    a.print();//此句和上句可以被A.getInstance().print();代替
  }
}

//6、在任何一个类的构造执行完之前，所有属性的内容都是其对应数据类型的默认值，而子类对构造执行之前，一定先执行父类构造，那么此时子类构造未执行，所以下例num=0；
abstract class A{
  public A(){//2、父类构造方法
    this.print();//3、调用print()方法
  }
  public abstract void print();
}
class B extends A{
  private int num=100;
  public B(int num){
    this.num=num;
  }
  public void print(){//4、调用复写后的方法。
    System.out.println("num="+num);
  }
}
public class TestDemo{
  public static void main(String args[]){
    new B(30);//1、执行构造//如果此步改为new B(30).print();则num为30
  }
}
```

章节23 课时 90 03071_抽象类的定义及使用（模板设计模式）

```
abstract class Action{//父类定义一个行为类
  public static final int EAT=1;
  public static final int SLEEP=5;
  public static final int WORK=7;
  public void command(int flag){
    switch(flag){
      case EAT:
        this.eat();
        break;
      case SLEEP:
        this.sleep();
        break;
      case WORK:
        this.work();
        break;
      case EAT+WORK:
        this.eat();
        this.work();
        break;
    }
  }
//因为现在不确定子类的实现是什么样的，所以定义为抽象方法。
  public abstract void eat();
  public abstract void sleep();
  public abstract void work();
}
class Robot extends Action{
  public void eat(){
    System.out.println("机器人补充能量");
  }
  public void sleep(){
  }
  public void work(){
    System.out.println("机器人在努力工作");
  }
}
class Human extends Action{
  public void eat(){
    System.out.println("人类正在吃饭");
```

```
  }
  public void sleep(){
    System.out.println("人类正在睡觉休息");
  }
  public void work(){
    System.out.println("人为了梦想在努力工作");
  }
}
class Pig extends Action{
  public void eat(){
    System.out.println("猪正在啃食槽");
  }
  public void sleep(){
    System.out.println("猪正在睡觉养膘");
  }
  public void work(){

  }
}
public class TestDemo{
  public static void main(String args[]){
    fun (new Robot());
    fun (new Human());
  }
  public static void fun(Action act){//定义一个Action的属性act
    act.command(Action.EAT);
    act.command(Action.SLEEP);
    act.command(Action.WORK);
  }
}
```

章节24 课时 91 03072_接口的定义及使用（接口基本定义）

```
//如果不会接口，别说你会Java，其他语言也无法理解
//如果一个类之中只是抽象方法和全局常量所组成，那么在这种情况下不会将其定义为一个抽象类（因为抽象类里面会有构造方法，构造方法是消除不掉的），而只会将其定位为我们的接口，所以接口严格来讲就属于一个特殊的类，而且这个类里面只有抽象方法与全局常量（连构造都没有）。
interface A{
  public static final String MSG="Hello";//全局常量
  public abstract void print();//抽象方法
}
//由于接口里面存在抽象方法，所以接口对象不可能直接使用关键字new进行实例化的操作。下面是接口的使用要求：
    1、接口必须要有子类，但是此时一个子类可以使用 implements关键字实现多个接口；
    2、接口的子类如果不是抽象类，那么必须要覆写接口中的全部抽象方法；
    3、接口的对象可以利用子类对象的向上转型进行实例化操作。
范例：
interface A{
  public static final String MSG="Hello";
  public abstract void print();
}
interface B{
  public abstract void get();
}
class X implements A,B{
  public void print(){
    System.out.println("A接口的抽象方法");
  }
  public void get(){
    System.out.println("B接口的抽象方法");
  }
}
public class TestDemo{
  public static void main(String args[]){
    X x=new X();//实例化X子类对象
    A a=x;//向上转型
    B b=x;//向上转型
```

```
    a.print();
    b.get();
  }
}

interface A{
  public static final String MSG="Hello";
  public abstract void print();
}
interface B{
  public abstract void get();
}
class X implements A,B{
  public void print(){
    System.out.println("A接口的抽象方法");
  }
  public void get(){
    System.out.println("B接口的抽象方法");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=new X();//实例化X子类对象
    B b=(B) a;//类型转换
    a.print();
    b.get();
  }
}

//对于子类而言，除了接口之外，还可能会去继承抽象类， 所以说一个子类又要继承抽象类又要去实现接口的话，请先使用extends继承，而后再使用inplements实现。
//对于接口而言，发现里面的组成就是抽象方法和全局常量，所以很多时候也有一些人为了省略编写，可以不用协商abstracta或public static final,并且在方法上是否编写public结果都是一样的，因为在接口里面只能够使用一种访问权限就是public。以下两个接口的定义最终效果是完全相同的：
interface A{
  public static final String MSG="Hello";
  public abstract void print();
}
interface A{
  String MSG="Hello";
  void print();
}


interface A{
  public void funA();
  abstract class B{
    public abstract void funB();
  }
}
class X implements A{//X实现了A接口
  public void funA(){
    System.out.println("Hello World!!");
  }
  class Y extends B{//内部抽象类的子类
    public void funB(){
      System.out.println();
    }
  }
}
public class TestDemo{
  public static void main(String args[]){
  }
}


//范例：在一个接口内部如果使用了static去定义一个内部接口，表示是一个外部接口。
interface A{
```

```
  public void funA();
  static interface B{//外部接口
    public void funB();
  }
}
class X implements A.B{//形成了一个外部接口
  public void funB(){
  }
}
public class TestDemo{
  public static void main(String args[]){
  }
}
```

接口在实际开发之中有三大核心作用：

1、定义不同层之间的操作标准；

2、表示一种操作能力；

3、表示将服务器端的远程方法试图暴露给客户端。

章节24 课时 92 03073_接口的定义及使用（标准定义）

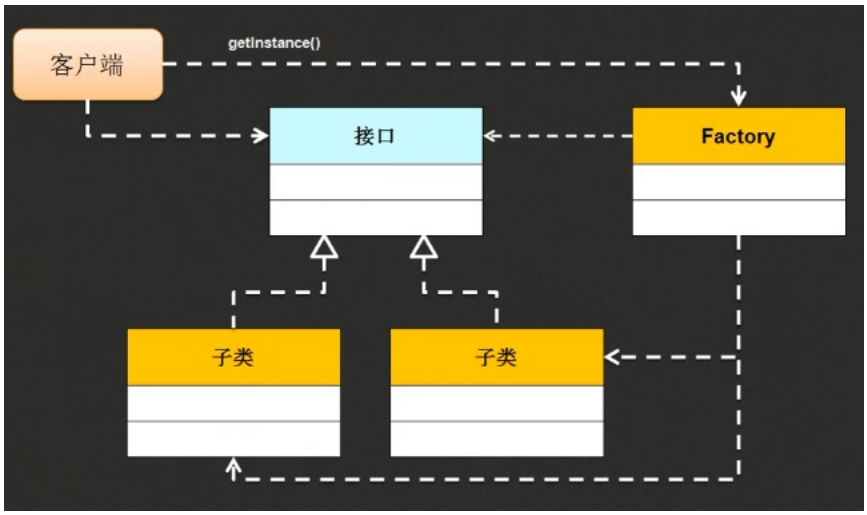
```
interface USB{
  public void start();
  public void stop();
}
class Computer{
  public void plugin(USB usb){//插入
    usb.start();//固定操作
    usb.stop();
  }
}
class Flash implements USB{
  public void start(){
    System.out.println("U盘开始使用");
  }
  public void stop(){
    System.out.println("U盘停止使用");
  }
}
class Print implements USB{
  public void start(){
    System.out.println("打印机开始使用");
  }
  public void stop(){
    System.out.println("打印机停止使用");
  }
}
public class TestDemo{
  public static void main(String args[]){
    Computer com=new Computer();
    com.plugin(new Flash());
    com.plugin(new Print());
  }
}
```

章节24 课时 93 03074_接口的定义及使用（工厂设计模式）熟练默写代码

```
interface Fruit{
  public void eat();
}
class Apple implements Fruit{
  public void eat(){
    System.out.println("***吃苹果");
  }
}
```

```
public class TestDemo{
  public static void main(String args[]){
    Fruit f=new Apple();
    f.eat();
  }
}
//如果要想确认一个代码是否真的好，有这么几个标准：
1、客户端调用简单，不需要关注具体的细节；
2、客户端之外的代码修改，不影响用户的使用。
以上代码，如果加一个橘子，那么得改Fruit a=new Orange;但是由于客户并不需要关心这些，所以存在问题。
经分析发现，最大的问题就在于关键字new，而这一问题就可以理解为耦合度太高，导致了代码维护的不方便，就相当于A一直要与B绑定在一起。可以完全参考Java虚拟机的思想：程序→JVM→适应不同的操作系统

interface Fruit{
  public void eat();
}
class Apple implements Fruit{
  public void eat(){
    System.out.println("***吃苹果");
  }
}
class Orange implements Fruit{
  public void eat(){
    System.out.println("***吃橘子");
  }
}
class Factory{
  public static Fruit getInstance(String className){
    if("apple".equals(className)){
      return new Apple();
    }else if("orange".equals(className)){
      return new Orange();
    }else{
      return null;
    }
  }
}
public class TestDemo{
  public static void main(String args[]){
    Fruit f=Factory.getInstance("apple");
    f.eat();
  }
}
```



//面试题：请编写一个Factory程序，答案如上。

章节24 课时 94 03075_接口的定义及使用（代理设计模式）熟练默写代码

```
interface Subject{//整个操作的核心主题
```

```
  public void make();//整个主题的核心功能
}
class RealSubject implements Subject{
  public void make(){
    System.out.println("皇帝陛下正在XX");
  }
}
class ProxySubject implements Subject{
  private Subject subject;
  public ProxySubject(Subject subject){
    this.subject=subject;
  }
  public void prepare(){
    System.out.println("为临幸做准备");
  }
  public void make(){
    this.prepare();
    this.subject.make();//告诉皇帝可以开始了
    this.destroy();
  }
  public void destroy(){
    System.out.println("把娘娘搬走了，皇帝伺候睡觉了");
  }
}
public class TestDemo{
  public static void main(String args[]){
    Subject sub=new ProxySubject(new RealSubject());
    sub.make();//调用的是代理主题操作
  }
}
```

章节24 课时 95 03076_接口的定义及使用（抽象类与接口区别）

NO	区别	抽象类	接口
1	关键字	abstract class	interface
2	组成	构造方法，普通方法，抽象方法，static方法，常量，变量	抽象方法，全局常量
3	子类使用	class 子类 extends 父类	class 子类 implements 接口
4	关系	抽象类可以实现多个接口	接口不能够继承抽象类，却可以继承多个父接口
5	权限	可以使用各种权限	只能够使用public权限
6	限制	单继承局限	没有单继承限制
7	子类	抽象类和接口都必须有子类，子类必须要覆写全部的抽象方法	抽象类和接口都必须有子类，子类必须要覆写全部的抽象方法
8	实例化对象	依靠子类对象的向上转型进行对象的实例化	依靠子类对象的向上转型进行对象的实例化

通过以上比较可以发现，抽象类支持的功能绝对要比接口更多，但只有一点不好，那就是单继承局限，所以这重要的一点就掩盖了所有抽象类的优点。即：当抽象类和接口都可以使用的时候，优先考虑接口。
一个不成文的参考：
在进行某些公共操作的时候一定要定义接口；
有了接口就需要利用子类完善方法；
如果是自己定义的接口，那么绝对不要去使用关键字new直接实例化接口子类，用工厂类完成。
开发总结：
接口与抽象类定义的不同
接口作为标准用于解耦合以及不同层之间的连接桥梁
一定要将工厂设计模式与代理设计模式的结构记下来。

章节25 课时 96 03077_Object类（基本概念）
Object是所有类的父类，也就是说任何一个类在定义的时候如果没有明确的继承一个父类的话，那么它就是Object类的子类。

也就是说以下2种类定义的最终效果是完全相同的。
```
class Book extends Object{}
class Book{}
```
整个Java里面类的继承关系一直都存在（除了Object类）。
既然Object类是所有类的父类，那么最大的一个好处就是利用Object类可以接收全部类的对象，因为可以向上自动转型。
范例：利用Object类来接收对象
```
class Book extends Object{
}
public class TestDemo{
  public static void main(String args[]){
    Object obja=new Book();
    Object objb="Hello";
    Book b=(Book)obja;
    String s=(String)objb;
  }
}
```
问题：为什么在Object类里面要定义一个无参构造方法呢？
既然Object类是所有类的父类，那么所有类对象实例化的时候，子类构造方法一定要默认调用父类的无参构造。
从严格意义上来讲（一般不遵守），任何一个简单Java类都应该覆写Object类中的三个方法：
取得对象信息：public String toString();
对象比较：public boolean equals(Object obj)
取得对象HASH码：public int hashCode();

章节25 课时 97 03078_Object类（toString()方法）
```
class Book extends Object{
}
public class TestDemo{
  public static void main(String args[]){
   Book b=new Book();
   String s="Hello";
   System.out.println(b);
   System.out.println(s);
   System.out.println(b.toString());
  }
}
```
//发现现在如果直接输出对象与调用toString()方法后输出对象的功能是完全一样的。输出操作会自动调用类对象中的toString方法将对象变为字符串后再输出。
//Object类中的toString()为了适应于所有对象的输出，所以只输出了对象的编码。
//如果现在有需要，也可以根据实际情况来覆写此方法。
//范例：
```
class Book extends Object{
  private String title;
  private double price;
  public Book(String title,double price){
    this.title=title;
    this.price=price;
  }
  public String toString(){//getInfo()
    return "书名："+this.title+",价格："+this.price;
  }
}
public class TestDemo{
  public static void main(String args[]){
   Book b=new Book("Java开发",79.8);
   System.out.println(b);
  }
}
```
//直接输出对象就调用了toString(),等于输出时候的代码又节约了。

章节25 课时 98 03079_Object类（equals()方法）

//之前我们使用了一个 compare()方法进行比较，这不标准，在开发之中标准的做法是使用用equals()方法完成。范例：
```
class Book extends Object{
  private String title;
  private double price;
  public Book(String title,double price){
    this.title=title;
```

```
    this.price=price;
  }
  public String toString(){//getInfo()
    return "书名：" +this.title+",价格：" +this.price;
  }
  public boolean equals(Object obj){
    if(this==obj){//地址相同
      return true;
    }
    if(obj==null){
      return false;
    }
    if(!(obj instanceof Book)){//不是本类对象
      return false;
    }
    Book book=(Book) obj;//此步要比较内容了，但是title和price属性是子类新增的类型,父类Object中并没有，需要向下转型，调用子类自己的特色。
    if(this.title.equals(book.title)&&this.price==book.price){
      return true;
    }
    return false;
  }
}
public class TestDemo{
  public static void main(String args[]){
    Book b1=new Book("Java开发",79.8);
    Book b2=new Book("Java开发",79.8);
    System.out.println(b1.equals(b2));
    System.out.println(b1.equals("Hello"));
  }
}
```

章节25 课时 99 03080_Object类（接收引用类型）

```
//数组和Object之间的转换
public class TestDemo{
  public static void main(String args[]){
    Object obj=new int[]{1,2,3};//向上转型
    System.out.println(obj);
    if(obj instanceof int[]){//是否是int数组
      int data[]=(int[]) obj;//向下转型
      for(int x=0;x<data.length;x++){
        System.out.println(data[x]);
      }
    }
  }
}


interface A{
  public void fun();
}
class B implements A{
  public void fun(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    A a=new B();//接口对象//实例化一个匿名对象，然后向上转型。完整代码可为：B b=new B(); A a=b;
    Object obj=a;//接收接口对象//a再向上转型为Object类对象。
    A t=(A) obj;//向下转型//强制类型转换
    t.fun();
  }
}//整个程序的类就统一在了Object类上了
```

章节25 课时 100 03081_Object类（完善链表）熟练默写代码

1、Object可以接受一切的数据类型（引用数据类型），包括了数组和接口，解决了数据统一问题。
2、toString()在对象直接输出时使用，equals()在对象比较时调用。
熟练默写下列代码，将以前的String 用Object全部代替（除了主方法中的String）后就成了一个标准的可用链表。

```
class Link{
  private class Node{
    private Object data;
    private Node next;
    public Node(Object data){
      this.data=data;
    }
    public void addNode(Node newNode){
      if(this.next==null){
        this.next=newNode;
      }else{
        this.next.addNode(newNode);
      }
    }
    public boolean containsNode(Object data){
      if(this.data.equals(data)){
        return true;
      }else{
        if(this.next!=null){
          return this.next.containsNode(data);
        }else{
          return false;
        }
      }
    }
    public Object getNode(int index){
      if(Link.this.foot++==index){
        return this.data;
      }else{
        return this.next.getNode(index);
      }
    }
    public void setNode(int index,Object data){
      if(Link.this.foot++==index){
        this.data=data;
      }else{
        this.next.setNode(index,data);
      }
    }
    public void removeNode(Node previous,Object data){
      if(data.equals(this.data)){
        previous.next=this.next;
      }else{
        this.next.removeNode(this,data);
      }
    }
    public void toArrayNode(){
      Link.this.retArray[Link.this.foot++]=this.data;
      if(this.next!=null){//有后续元素
        this.next.toArrayNode();
      }
    }
  }
  private Node root;
  private int count=0;
  private int foot=0;
  private Object [] retArray;//返回的数组
  public void add(Object data){
    Node newNode=new Node(data);
    if(this.root==null){
      this.root=newNode;
    }else{
```

```
      this.root.addNode(newNode);
    }
    count++;
  }
  public int size(){
    return this.count;
  }
  public boolean isEmpty(){
   return this.count==0;
  }
  public boolean contains(Object data){
    if(data==null||this.root==null){
      return false;
    }else{
      return this.root.containsNode(data);
    }
  }
  public Object get(int index){
    if(this.count<=index){
      return null;
    }
    this.foot=0;
    return this.root.getNode(index);
  }
  public void set(int index,Object data){
    if(this.count<=index){
      return;
    }
    this.foot=0;
    this.root.setNode(index,data);
  }
  public Object [] toArray(){
    if(this.root==null){
      return null;
    }
    this.foot=0;//需要脚标控制
    this.retArray=new Object[this.count];//根据保存内容开辟数组
    this.root.toArrayNode();//交给Node类处理
    return this.retArray;
  }
  public void remove(Object data){
    if(this.contains(data)){
      if(data.equals(this.root.data)){
        this.root=this.root.next;
      }else{
        this.root.next.removeNode(this.root,data);
      }
      this.count--;
    }
  }
}
public class TestDemo{
  public static void main(String args[]){
    Link all=new Link();
    all.add("A");//String转为Object
    all.add("B");//String转为Object
    all.add("C");//String转为Object
    all.remove("A");//String覆写了equals()
    Object data[]=all.toArray();
    for(int x=0;x<data.length;x++){
      String str=(String) data[x];//Object变为String
      System.out.println(str);
    }
  }
}
```

章节25 课时 101 03082_综合实战：宠物商店

```
//到目前为止，有4个代码模型；1、简单Java类的单独编写；2、类的转换；3、对象比较；4、接口应用。
class Link{//第四个代码模型
  private class Node{
    private Object data;
    private Node next;
    public Node(Object data){
      this.data=data;
    }
    public void addNode(Node newNode){
      if(this.next==null){
        this.next=newNode;
      }else{
        this.next.addNode(newNode);
      }
    }
    public boolean containsNode(Object data){
      if(this.data.equals(data)){
        return true;
      }else{
        if(this.next!=null){
          return this.next.containsNode(data);
        }else{
          return false;
        }
      }
    }
    public Object getNode(int index){
      if(Link.this.foot++==index){
        return this.data;
      }else{
        return this.next.getNode(index);
      }
    }
    public void setNode(int index,Object data){
      if(Link.this.foot++==index){
        this.data=data;
      }else{
        this.next.setNode(index,data);
      }
    }
    public void removeNode(Node previous,Object data){
      if(data.equals(this.data)){
        previous.next=this.next;
      }else{
        this.next.removeNode(this,data);
      }
    }
    public void toArrayNode(){
      Link.this.retArray[Link.this.foot++]=this.data;
      if(this.next!=null){//有后续元素
        this.next.toArrayNode();
      }
    }
  }
  private Node root;
  private int count=0;
  private int foot=0;
  private Object [] retArray;//返回的数组
  public void add(Object data){
    Node newNode=new Node(data);
    if(this.root==null){
      this.root=newNode;
    }else{
      this.root.addNode(newNode);
    }
    count++;
  }
```

```
  public int size(){
    return this.count;
  }
  public boolean isEmpty(){
   return this.count==0;
  }
  public boolean contains(Object data){
    if(data==null||this.root==null){
      return false;
    }else{
      return this.root.containsNode(data);
    }
  }
  public Object get(int index){
    if(this.count<=index){
      return null;
    }
    this.foot=0;
    return this.root.getNode(index);
  }
  public void set(int index,Object data){
    if(this.count<=index){
      return;
    }
    this.foot=0;
    this.root.setNode(index,data);
  }
  public Object [] toArray(){
    if(this.root==null){
      return null;
    }
    this.foot=0;//需要脚标控制
    this.retArray=new Object[this.count];//根据保存内容开辟数组
    this.root.toArrayNode();//交给Node类处理
    return this.retArray;
  }
  public void remove(Object data){
    if(this.contains(data)){
      if(data.equals(this.root.data)){
        this.root=this.root.next;
      }else{
        this.root.next.removeNode(this.root,data);
      }
      this.count--;
    }
  }
}
interface Pet{
  public String getName();//得到宠物的名字
  public int getAge();//得到宠物的年龄
}
class PetShop{//一个宠物商店要保存多个宠物信息
  private Link pets=new Link();//保存宠物信息,因为Link是链表，要想使用先实例化。
  public void add(Pet pet){//上架
    this.pets.add(pet);//向集合里添加数据
  }
  public void delete(Pet pet){//下架，删除在链表里面需要equals方法支持。
    this.pets.remove(pet);
  }
  public Link search(String keyWord){//模糊查询一定是返回多个类型，不知道多少个。返回Link即可。
    Link result=new Link();//保存结果
    //将集合变为对象数组的形式返回，因为集合保存的是Object,但是真正要查询的数据是在Pet接口对象的getName()方法的返回值
    Object obj[]=this.pets.toArray();
    for(int x=0;x<obj.length;x++){
      Pet p=(Pet) obj[x];
      if(p.getName().contains(keyWord)){
        result.add(p);//保存满足条件的结果。
      }
```

```
    }
    return result;
  }
}
class Cat implements Pet{//如果不实现接口无法接收宠物信息
  private String name;
  private int age;
  public Cat(String name,int age){
    this.name=name;
    this.age=age;
  }
  public boolean equals(Object obj){
    if(this==obj){
      return true;
    }
    if(obj==null){
      return false;
    }
    if(!(obj instanceof Cat)){
      return false;
    }
    Cat c=(Cat) obj;//向下转型
    if(this.name.equals(c.name)&&this.age==c.age){
      return true;
    }
    return false;
  }
  public String getName(){//覆写
    return this.name;
  }
  public int getAge(){//覆写
    return this.age;
  }
  public String toString(){
    return "猫的名字："+this.name+"，猫的年龄："+this.age;
  }
}
class Dog implements Pet{//可以发现，有了接口之后，子类的形式都非常类似，这属于接口的实现特点。
  private String name;
  private int age;
  public Dog(String name,int age){
    this.name=name;
    this.age=age;
  }
  public boolean equals(Object obj){
    if(this==obj){
      return true;
    }
    if(obj==null){
      return false;
    }
    if(!(obj instanceof Dog)){
      return false;
    }
    Dog c=(Dog) obj;//向下转型
    if(this.name.equals(c.name)&&this.age==c.age){
      return true;
    }
    return false;
  }
  public String getName(){//覆写
    return this.name;
  }
  public int getAge(){//覆写
    return this.age;
  }
  public String toString(){
    return "狗的名字："+this.name+"，狗的年龄："+this.age;
```

```
  }
}
public class TestDemo{
  public static void main(String args[]){
    PetShop shop=new PetShop();
    shop.add(new Cat("王猫",20));
    shop.add(new Cat("惊猫",10));
    shop.add(new Cat("雷猫",9));
    shop.add(new Dog("王狗",9));
    shop.add(new Dog("惊狗",20));
    shop.add(new Dog("雷狗",20));
    shop.delete(new Dog("王狗",9));
    Link all=shop.search("惊");
    Object obj []=all.toArray();
    for(int x=0;x<obj.length;x++){
      System.out.println(obj[x]);
    }
  }
}
```

章节26 课时 102 03083_匿名内部类

```
interface Message{
  public void print();
}
class MessageImpl implements Message{
  public void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    fun(new MessageImpl());//传了一个Message接口的子类对象到fun()方法之中
  }
  public static void fun(Message msg){
    msg.print();//此处调用的一定是被覆写过的方法。msg则接收了接口的子类匿名对象（new MessageImpl）
  }
}
```

```
interface Message{
  public void print();
}
public class TestDemo{
  public static void main(String args[]){
    fun(new Message(){
      public void print(){
        System.out.println("Hello World!");
      }
    });
  }
  public static void fun(Message msg){
    msg.print();
  }
}

//使用匿名内部类的时候有一个前提：必须要基于接口或抽象类的应用。
//但是需要强烈强调的是，如果匿名内部类定义在了方法里面，方法的参数或者是变量要被匿名内部类所访问，必须加上final关键字（JDK1.8之后改变了）
//匿名内部类是在抽象类和接口的基础上发展的，最大的好处是帮助用户减少了类的定义。
```

章节26 课时 103 03084_基本数据类型的包装类（定义简介）

```
class MyInt{//一个类
  private int num;//这个类包装的基本数据类型
  public MyInt(int num){//构造的目的是为了将基本数据类型传递给对象
    this.num=num;
```

```
  }
  public int intValue(){//将包装的数据内容返回
    return this.num;
  }
}
public class TestDemo{
  public static void main(String args[]){
    MyInt mi=new MyInt(10);//将int包装为对象
    int temp=mi.intValue();//将对象中包装的基本数据类型取出
    System.out.println(temp*2);//只有取出包装数据之后才可以进行计算
  }
}
```

因为这样的实现是比较容易的，所以在Java里面为了方便用户使用，所以专门提供了一组包装类，来包装基本类型：byte(Byte)、short（Short）、int（Integer）、long（Long）、float（Float）、double（Double）、char(Character)、boolean（Boolean）。但是以上的包装类又分为两种子类型：

对象型包装类（Object直接子类）：Character、Boolean。

数值型包装类（Number直接子类）：Byte、Short、Integer、Float、Double、Long。

Number是一个抽象类，里面定义了6个方法：intValue()、byteValue()、doubleValue()、floatValue()、longValue()、shortValue()。

章节26 课时 104 03085_基本数据类型的包装类（装箱与拆箱）

范例：使用int Integer

```
public class TestDemo{
  public static void main(String args[]){
    Integer obj=new Integer(10);//将基本数据类型装箱
    int temp=obj.intValue();//将基本数据类型拆箱
    System.out.println(temp*2);
  }
}//之前编写的MyInt类，现在换成了Integer这个系统类
```

范例：使用double Double

```
public class TestDemo{
  public static void main(String args[]){
    Double obj=new Double(10.2);
    double temp=obj.doubleValue();
    System.out.println(temp*2);
  }
}//之前编写的MyInt类，现在换成了Integer这个系统类
```

范例：使用Boolean Boolean

```
public class TestDemo{
  public static void main(String args[]){
    Boolean obj=new Boolean(true);
    boolean temp=obj.booleanValue();
    System.out.println(temp);
  }
}
```

在JDK1.5之前能够使用的操作都是以上形式的代码，但是从JDK1.5之后，Java为了方便开发，提供了自动装箱和自动拆箱的机制，并且可以直接利用包装类的对象进行数学计算。

范例：观察自动装箱与拆箱

```
public class TestDemo{
  public static void main(String args[]){
    Integer obj=10;//自动装箱
    int temp=obj;//自动拆箱
    obj++;//包装类直接进行数学计算
    System.out.println(temp*obj);
  }
}
```

```
public class TestDemo{
  public static void main(String args[]){
    Integer obja=10;//自动装箱，存的不是10，而是对象。
    Integer objb=10;
    Integer objc=new Integer(10);
    System.out.println(obja==objb);
```

```
    System.out.println(obja==objc);
    System.out.println(objb==objc);
    System.out.println(obja.equals(objc));
  }
}//在使用包装类时，很少用构造方法赋值，几乎都是直接赋值。这一点与String相同，在判断内容是否相等时，一定要使用equals()方法。

提示：Object此时可以统一天下了。
public class TestDemo{
  public static void main(String args[]){
    Object obj=10;//先包装再转换
    int temp=(Integer) obj;//向下变为Integer类型
    System.out.println(temp*2);
  }
}

Object可以接收一切的引用数据类型，但是由于存在有自动的装箱机制，那么Object也可以存放基本类型了。
流程：基本数据类型→自动装箱（变成对象）→向上转型为Object
public class TestDemo{
  public static void main(String args[]){
    Object obj=10;//先包装再转换
    int temp=(Integer) obj;//向下变为Integer类型
    System.out.println(temp*2);
  }
}


public class TestDemo{
  public static void main(String args[]){
    Boolean flag=true;//自动装箱
    if(flag){//直接拆箱判断
      System.out.println("Hello World!");
    }
  }
}
有了这一系列的自动装箱和拆箱的支持之后，在数据类型的选择上就方便了许多。
```

章节26 课时 105 03086_基本数据类型的包装类（数据类型转换）

```
使用包装类最多的情况实际上是它的数据类型转换功能上，在包装类里面提供有将String方法，使用几个代表的类作说明：
Integer类：public static int parseInt(String s);
Double类：public static double parseDouble(String s);
Boolean类：public static boolean parseBoolean(String s);
特别需要注意的是Character类里面并不存在有字符串变为字符的方法。因为String类里面有一个charAt()的方法可以根据索引取出字符内容。
范例：将字符串变为int型数据
public class TestDemo{
  public static void main(String args[]){
    String str="123";//此处字符串必须为数字，否则出错
    int temp=Integer.parseInt(str);
    System.out.println(temp*2);
  }
}
范例：
public class TestDemo{
  public static void main(String args[]){
    String str="1.3";//此处字符串没有小数点也可以，转换之后有.0
    double temp=Double.parseDouble(str);
    System.out.println(temp*2);
  }
}
范例：
public class TestDemo{
  public static void main(String args[]){
    String str="true";//如果此处为其他任何字符串，结果返回均为false
    boolean flag= Boolean.parseBoolean(str);
    if(flag){
```

```
      System.out.println("满足条件！");
    }else{
      System.out.println("不满足条件！");
    }
  }
}
```

范例：将其他类型转为String

```
public class TestDemo{
  public static void main(String args[]){
    int num=100;
    String str=num+"";//任何数字和String类型相加必然得到一个String类型,但是会产生垃圾。
    System.out.println(str.replaceAll("0","9"));
  }
}
```

正确的编写方法如下：

```
public class TestDemo{
  public static void main(String args[]){
    int num=100;
    String str=String.valueOf(num);
    System.out.println(str.replaceAll("0","9"));
  }
}
```

1、一定要清楚JDK1.5之后才提供有自动装箱和拆箱操作；
2、字符串与基本数据类型的互相转换：
字符串变为基本数据类型，依靠包装类的parseXxx()方法
基本数据类型变为字符串，依靠String.valueOf(基本数据类型)

章节27 课时 106 03087_包的定义及使用（包的定义）

1、对于整个面向对象核心的概念：类、接口、抽象类、对象，到此为止，所以的主体结构的代码都已经讲解完成。
2、包的定义及使用；
3、系统包的介绍；
4、Jar命令的操作；
包指的是一个程序的目录，在最早的时候如果要开发一个程序，只需要定义一个*.java文件，而后再这个文件里面编写所需要的类文件，而在编译之后程序将直接保存在根目录下，而利用包可以实现同一个程序的拆分，即：可以根据程序的要求将代码保存在不同的目录下。
//此条无关主题转目录：d: cd mydemo
如果说类有包的定义了，那么就必须让其保存在特定的目录下，只不过不要自己手工创建这些目录，应该使用命令自动生成。
范例：

```
package cn.mldn.demo;//包，其中的“.”表示子目录
public class Hello{
  public static void main(String args[]){
    System.out.println("Hello World!");
  }
}
```

打包编译javac-d . Hello.java
"-d" 生成目录，根据package的定义生成
"." 设置保存的路径，如果为"."表示在当前所在目录下。
在解释程序的时候不要进入到包里面解释程序，应该在包外面解释程序。
应该输入java cn.mldn.demo.Hello执行

章节27 课时 107 03088_包的定义及使用（包的导入）

```
package cn.mldn.util;//包1,文件名为Message
public class Message{
  public void print(){
    System.out.println("Hello World!");
  }
}

package cn.mldn.demo;//包2,文件名为TestMessage
import cn.mldn.util.Message;//调用包1
public class TestMessage{
  public static void main(String args[]){
    Message temp=new Message();
    temp.print();
  }
```

```
}
//编译: javac -d . *.java
//以上编译代码表示编译当前目录下所有的.java文件
//最后执行:java cn.mldn.demo.TestMessage

//但是发现在导入包的时候也出现了一个问题，如果使用一个包中类的时候要编写"import 包.类"，如果说现在要使用一个包中多个类的时候，那么肯定要重复去编写"import 包.类",很麻烦，所以此时可以使用"包.*"的方式来代替一个包中多个类的导入操作。
//使用"包.*"指的并不是全部导入，而是只导入程序里面所需要的类，所以不需要去考虑性能问题。
//例如：import cn.mldn.util.*
//
那么久必须有一个重要的问题需要注意，有可能同一个代码里面会同时导入不同的包，并且不同的包里面有可能会存在同名类。例如：现在有两个类：
  cn.mldn.util.Message;
  org.lxh.Message;
由于某种需要，要同时导入以上两个包。为了可以明确找到所需要的类，那么可以在使用类的时候加上包名称。
如果觉得导包操作会造成冲突，那么就在实例化对象时写上完整的类名称（包含目录）。
package cn.mldn.demo;
//这两个包里面都有同一个名称的类(Message)
import cn.mldn.util.*;
import org.lxh.*;
public class TestMessage{
  public static void main(String args[]){
    cn.mldn.util.Message temp=new cn.mldn.util.Message();
    temp.print();
  }
}
```

章节27 课时 108 03089_包的定义及使用（系统常用包）

Java本身提供了大量的程序开发包（除了java自己提供的，还有许多第三方提供了开发包）。在java开发里面有如下的一些常见的额系统包：
1、java.lang:包含了String、Object、Integer等类，从JDK1.1开始此包自动导入。
2、java.lang.reflect:反射开发包；
3、java.util:Java工具包，提供了大量的工具类，像链表；
4、java.util.regex:正则工具包；
5、java.text:国际化处理程序包；
6、java.io:进行输入、输入处理以及文件操作；
7、java.net:网络编程开发包
8、java.sql:数据库程序开发包
9、java.applet:Applet程序开发包（已经不用了）；指的是在网页上嵌套的程序，可以用它做一些动态效果。

章节27 课时 109 03090_包的定义及使用（jar命令）

```
用法: jar {ctxui}[vfmn0PMe] [jar-file] [manifest-file] [entry-point] [-C dir] files ...
选项:
    -c  创建新档案
    -t  列出档案目录
    -x  从档案中提取指定的 (或所有) 文件
    -u  更新现有档案
    -v  在标准输出中生成详细输出
    -f  指定档案文件名
    -m  包含指定清单文件中的清单信息
    -n  创建新档案后执行 Pack200 规范化
    -e  为捆绑到可执行 jar 文件的独立应用程序
        指定应用程序入口点
    -0  仅存储; 不使用任何 ZIP 压缩
    -P  保留文件名中的前导 '/' (绝对路径) 和 ".." (父目录) 组件
    -M  不创建条目的清单文件
    -i  为指定的 jar 文件生成索引信息
    -C  更改为指定的目录并包含以下文件
如果任何文件为目录, 则对其进行递归处理。
清单文件名, 档案文件名和入口点名称的指定顺序
与 'm', 'f' 和 'e' 标记的指定顺序相同。

示例 1: 将两个类文件归档到一个名为 classes.jar 的档案中:
       jar cvf classes.jar Foo.class Bar.class
```

```
//范例：定义一个Message.java文件
package cn.mldn.util;
public class Message{
  public void print(){
    System.out.println("Hello World!");
  }
}
//打包编译此文件：javac -d . Message.java
//此时会形成“包.类”形式。随后假设这里有很多*.class文件，并且交付用户使用，那么将这个包的代码压缩，输入jar-cvf my.jar cn
//生成的my.jar文件并不能够直接使用，必须配置CLASSPATH才可以加载。
//SET CLASSPATH=.;D:\mydemo\my.jar
//设置：SET CLASSPATH=.;D:\mydemo\my.jar
package cn.mldn.demo;
import cn.mldn.util.*;
public class TestMessage{
  public static void main(String args[]){
    cn.mldn.util.Message temp=new cn.mldn.util.Message();
    temp.print();
  }
}
//编译：javac -d . TestMessage.java
//执行：java cn.mldn.demo.TestMessage


//在以后的开发之中需要大量的使用第三方的jar文件，那么所有的jar文件必须配置CLASSPATH，否则不能够使用。
//以后开发的程序一定要有包的存在
//如果包冲突的时候需要协商完成的类名称；
//以后使用第三方jar文件一定要配置CLASSPATH
```

章节28 课时 110 03091_访问控制权限

在之前只是简单的见到了封装型，但是对于封装是与访问控制权限有直接联系的。在java里面一种支持四种访问控制权限。public、protected、default、private，这四种访问控制权限特点如下：

NO	范围	private	default	protected	public
1	在同一个类中	√	√	√	√
2	在同一包的不同类		√	√	√
3	在不同包的子类里			√	√
4	在不同包的非子类				√

除了public之外，对于封装可以使用private、protected、default，只不过一般我们不会去考虑使用default。对于其他的权限基本上已经见过了，重点放在protected，这种权限它直接与包的定义有关。

```
//范例：定义cn.mldn.demoa.A类
package cn.mldn.demoa;
public class A{
  protected String info="Hello";
}
//范例：定义cn.mldn.demob.B类，此类继承A类
package cn.mldn.demob;
import cn.mldn.demoa.A;
public class B extends A{//是A不同包的子类
  public void print(){
    System.out.println("A类的info="+super.info);
  }
}
//范例：定义test.Test类
package test;
import cn.mldn.demob.B;
public class Test{
  public static void main(String args[]){
    new B().print();
  }
}
//编译：javac -d . A.java
```

//编译：javac -d . B.java
//编译：javac -d . Test.java
//执行：java test.Test
//java的封装性是以private、protected、default三种权限的定义；
//对于权限的选择，给出以下的建议：
声明属性就使用private；
声明方法就是用public；
//关于命名要求：
类名称每一个单词的首字母大写，其余小写，例如：StudentInfo；
属性名称第一个单词字母小写，而后每个单词首字母大写，例如：studentName；
方法名称和属性名称规则一样。
常量名称用大写字母表示；
包名称用小写字母，例如cn.mldn.demo;

章节28 课时 111 03092_构造方法私有化（单例设计）

```
class Singleton{
  static Singleton instance =new Singleton();
  private Singleton(){}//构造方法私有化
  public void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    Singleton s=null;//声明对象
    s=Singleton.instance;//直接访问static属性
    s.print();
  }
}
//在一个类定义的时候首先想到的就是类中的属性需要进行封装
static Singleton instance =new Singleton();
//而一旦封装之想访问此属性只能够通过getter方法，那么就需要提供有一个getter方法可以同样不受到Singleton实例化对象的控制，继续使用static属性。
class Singleton{
  private static Singleton instance =new Singleton();
  private Singleton(){}//构造方法私有化
  public static Singleton getInstance(){
    return instance;
  }
  public void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    Singleton s=null;//声明对象
    s=Singleton.getInstance();//直接访问static属性
    s.print();
  }
}
//代码意义：//如果说现在要想控制一个类中实例化对象的产生个数，那么首先要锁定的就是类中的构造方法，因为在实例化任何新对象都要使用构造方法，如果构造方法被锁了，呢么自然就无法产生新的实例化对象了。
//可是既然需要是一个实例化对象，那么就可以在类的内部使用static方式来定义一个公共的对象，并且每一次通过static方法返回唯一的一个对象，这样外部不管产生多少次调用，那么最终一个类只能够产生唯一的一个对象，这样的设计就属于单例设计模式。（Singleton）。
class Singleton{
  private static Singleton instance =new Singleton();
  private Singleton(){}//构造方法私有化
  public static Singleton getInstance(){
    return instance;
  }
  public void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
```

```
    Singleton s1=Singleton.getInstance();
    Singleton s2=Singleton.getInstance();
    Singleton s3=Singleton.getInstance();
    Singleton s4=Singleton.getInstance();
    Singleton s5=Singleton.getInstance();
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s3);
    System.out.println(s4);
    System.out.println(s5);
  }
}
```

```
//终极代码（饿汉式）
class Singleton{
  private static final Singleton INSTANCE =new Singleton();
  private Singleton(){}//构造方法私有化
  public static Singleton getInstance(){
    return INSTANCE;
  }
  public void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    Singleton s=Singleton.getInstance();
    s.print();
  }
}
//程序特点：构造方法私有化，在类的内部定义static属性与方法，利用static方法取得本类的实例化对象，不管外部会产生多少个Singleton类的对象，但是本质上永远只有唯一的一个实例化对象。
//可是对于单例设计模式有两种形式：饿汉式、懒汉式。
//在之前所编写的单例实际上就属于饿汉式的应用，在Singleton类定义的时候就已经准备比好了一个Singleton实例化对象INSTANCE，而并没眼关心这个对象是否使用。
//懒汉式最大的特点就是在于它是在第一次使用的时候才进行实例化操作。
//范例：实现懒汉式
class Singleton{
  private static  Singleton instance ;//此处老师讲解的有final而且instance没有大写，没有测试运行。但我测试是不通过的。
  private Singleton(){}//构造方法私有化
  public static Singleton getInstance(){
    if(instance==null){//此时还没有实例化
      instance =new Singleton();
    }
    return instance;
  }
  public void print(){
    System.out.println("Hello World!");
  }
}
public class TestDemo{
  public static void main(String args[]){
    Singleton s=Singleton.getInstance();
    s.print();
  }
}

//单例是一个理解过程，其核心目的是，让一个类在整个系统里只允许存在有一个实例化对象。
```

章节28 课时 112 03093_构造方法私有化（多例设计）

```
class Sex{
  private String title;
  private static final Sex MALE=new Sex("男");
  private static final Sex FEMALE=new Sex("女");
  private Sex(String title){//构造私有化了
    this.title=title;
  }
```

```
  public String toString(){
    return this.title;
  }
  public static Sex getInstance(String ch){
    switch(ch){
      case "man":
        return MALE;
      case "woman":
        return FEMALE;
      default:
        return null;
    }
  }
}
public class TestDemo{
  public static void main(String args[]){
     Sex sex=Sex.getInstance("man");
     System.out.println(sex);
  }
}
```

```
class Sex{
  private String title;
  private static final Sex MALE=new Sex("男");
  private static final Sex FEMALE=new Sex("女");
  private Sex(String title){//构造私有化了
    this.title=title;
  }
  public String toString(){
    return this.title;
  }
  public static Sex getInstance(int ch){
    switch(ch){
      case 1:
        return MALE;
      case 2:
        return FEMALE;
      default:
        return null;
    }
  }
}
interface Choose{//纯做属性的接口
  public static final int MAN=1;//全局常量
  public int WOMAN=2;//全局常量
}
public class TestDemo{
  public static void main(String args[]){
     Sex sex=Sex.getInstance(Choose.MAN);
     System.out.println(sex);
  }
}
//以上代码自己看没有什么问题，但如果交给其他人就看不懂了
//单例设计模式就是一个类只能够产生唯一的一个实例化对象。
//多例设计模式可以产生多个实例化对象，要取得的时候需要加上标记。
//不管是单例还是多例设计，核心：构造方法私有化。
```

章节29 课时 113 03094_异常的捕获及处理（异常的产生）

1、异常的产生以及对于程序的影响；
2、异常处理的格式；
3、异常的处理流程；
4、throw、throws关键字的使用；
5、异常处理的使用标准；
6、自定义异常。
异常是Java的一个重大特色，合理的使用异常处理，可以让我们的程序更加的健壮。
异常是导致程序中断执行的一种指令流，异常一旦出现并且没有进行合理处理的话，那么程序就将终端执行。

范例：不产生异常的代码
```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    System.out.println("2、除法计算(10/2)="+(10/2));
    System.out.println("3、除法计算的结束");
  }
}
```
范例：产生异常的代码
```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    System.out.println("2、除法计算(10/0)="+(10/0));
    System.out.println("3、除法计算的结束");
  }
}
```
一旦产生异常之后，发现产生异常的语句及之后的语句将不再执行，默认情况是进行异常信息的输出，而后自动结束程序的执行。
我们要做的事情是：即使出现了异常，那么也应该让程序正确的执行完毕。

章节29 课时 114 03095_异常的捕获及处理（处理异常）

如果要想进行异常的处理，在java之中提供了三个关键字：try、catch、finally，而这三个关键字的使用语法如下：
```
try{
  //有可能出现异常的语句
}[catch(异常类型 对象){
  //异常处理
}catch(异常类型 对象){
  //异常处理
}catch(异常类型 对象){
  //异常处理
}...][finally{
  //不管是否出现异常，都执行的统一代码
}]
```
对于以上的操作组合：try...catch、try...catch...finally、try...finally
应用异常处理格式
```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    try{
    System.out.println("2、除法计算(10/0)="+(10/0));
    }catch(ArithmeticException e){
      System.out.println("*****出现异常了****");
    }
    System.out.println("3、除法计算的结束");
  }
}
```
由于使用了异常处理，这样即使程序中出现了异常，发现也可以正常的执行完毕。
出现异常是为了解决异常，所以为了能够进行异常的处理，可以使用异常类中提供过得printStackTrace()方法进行异常信息的完整输出。
```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    try{
    System.out.println("2、除法计算(10/0)="+(10/0));
    }catch(ArithmeticException e){
      e.printStackTrace();
    }
    System.out.println("3、除法计算的结束");
  }
}
//此时发现打印的异常信息是很完整的（java.lang.是包）
1、除法计算的开始
java.lang.ArithmeticException: / by zero
        at TestDemo.main(TestDemo.java:5)
3、除法计算的结束
```

//范例：还可以使用try...catch...finally

```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    try{
    System.out.println("2、除法计算(10/0)="+(10/0));
    }catch(ArithmeticException e){
      e.printStackTrace();
    }finally{
      System.out.println("##不管是否出现异常我都执行！");
    }
    System.out.println("3、除法计算的结束");
  }
}
```

```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    try{
      int x=Integer.parseInt(args[0]);
      int y=Integer.parseInt(args[1]);
      System.out.println("2、除法计算(10/0)="+(x/y));
    }catch(ArithmeticException e){
      e.printStackTrace();
    }finally{
      System.out.println("##不管是否出现异常我都执行！");
    }
    System.out.println("3、除法计算的结束");
  }
}
//在异常捕获的时候发现一个try语句后面可以跟多个catch语句。
//1、用户执行的时候不输入参数(java TestDemo):java.lang.ArrayIndexOutOfBoundsException
//2、用户输入的参数不是数字(java TestDemo a b):java.lang.NumberFormatException
//3、被除数为0(java TestDemo 10 0):java.lang.ArithmeticException已处理
//以上的代码里只存在一个catch，所以只能够处理一个异常，如果其他没有处理的异常，会导致程序的中断执行。发现：finally一直在执行，但是"3、除法计算的结束"，只有在第三次异常被处理后才执行了。
```

```
//范例：加入多个catch
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    try{
      int x=Integer.parseInt(args[0]);
      int y=Integer.parseInt(args[1]);
      System.out.println("2、除法计算(10/0)="+(x/y));
      System.out.println("##################");
    }catch(ArrayIndexOutOfBoundsException e){//数组越界
      e.printStackTrace();
    }catch(NumberFormatException e){//数字格式异常
      e.printStackTrace();
    }catch(ArithmeticException e){//算数异常
      e.printStackTrace();
    }finally{
      System.out.println("##不管是否出现异常我都执行！");
    }
    System.out.println("3、除法计算的结束");
  }
}
//编译完成后分别输入javac TestDemo、javac TestDemo a b、javac TestDemo 10 0来执行,发现均已处理。
//程序现在的确很健壮，所有可能出现的异常都处理完了。
//但是，以上的异常都已经知道了，你还让它出现，这绝对是技术问题。
```

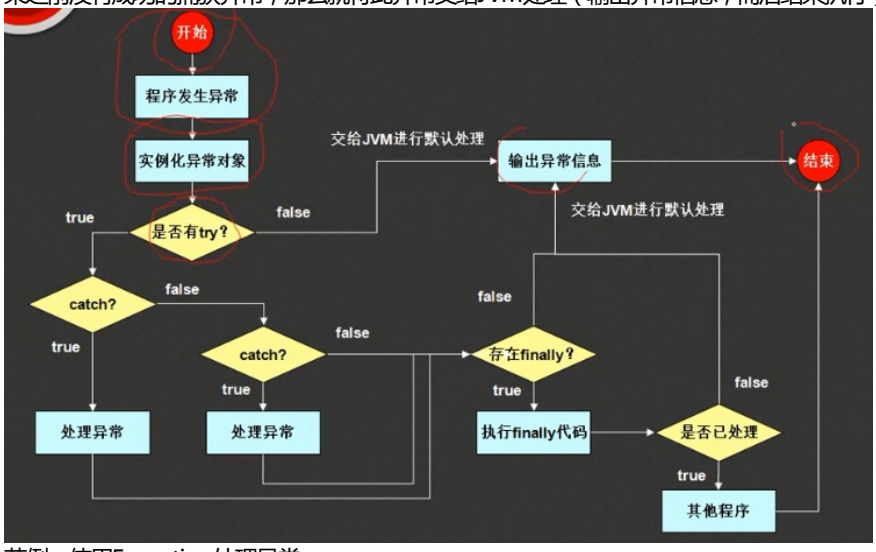
章节29 课时 115 03096_异常的捕获及处理（异常处理流程）（核心）

算数异常：

```
Class ArithmeticException
  java.lang.Object
    java.lang.Throwable
      java.lang.Exception
```

java.lang.RuntimeException
java.lang.ArithmeticException

数字格式异常：
Class NumberFormatException
java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
java.lang.IllegalArgumentException
java.lang.NumberFormatException
经过异常类的观察可以发现所有的异常类都是Throwable的子类。而在Throwable下有两个子类（面试题：请解释Error和Exception的区别）
Error：指的是JVM错误，即此时的程序还没有执行，如果没有执行用户无法处理。
Exception：指的是程序运行中产生的异常，用户可以处理。也就是所谓的异常处理指的就是所有Exception以及它的子类异常。
请解释Java中的异常处理流程：
1、当程序出现了异常后，那么会有JVM自动根据异常的类型实例化一个与之类型匹配的异常类对象。（此处用户不用去关系new，系统自动负责处理）
2、产生了异常对象之后会判断当前的语句上是否存在有异常处理，如果现在没有异常处理，那么就交给JVM进行默认的异常处理，输出异常信息，而后结束程序的调用；
3、如果此时存在有异常的捕获操作，那么会由try语句来捕获产生的异常类实例化对象，而后与try语句后的每个catch进行比较，如果现在有符合的捕获类型，则使用当前catch的语句来进行异常的处理，如果不匹配则向下继续匹配其他的catch；
4、不管最后异常处理是否能够匹配，那么都要向后执行，如果此时程序中存在有finally语句，那么就先执行finally语句中的代码，但是执行完毕后需要根据之前的catch匹配结果来决定如何执行，如果之前已经成功的捕获了异常，那么我们就执行finally类之后的代码，如果之前没有成功的捕获异常，那么就将此异常交给JVM处理（输出异常信息，而后结束执行）。



范例：使用Exception处理异常
```
public class TestDemo{
  public static void main(String args[]){
    System.out.println("1、除法计算的开始");
    try{
      int x=Integer.parseInt(args[0]);
      int y=Integer.parseInt(args[1]);
      System.out.println("2、除法计算="+(x/y));
      System.out.println("##################");
    }catch(Exception e){//数组越界
      e.printStackTrace();
    }finally{
      System.out.println("##不管是否出现异常我都执行！");
    }
    System.out.println("3、除法计算的结束");
  }
}
```
此时所有的异常都使用了Exception进行处理，所以在程序之中不用再去关心到底使用哪一个异常。
在使用多个catch捕获异常的时候，捕获范围大的异常一定要放在捕获范围小的异常之后，否则程序编译错误。
虽然直接捕获Exception比较方便，但这样也不好，因为所以的异常都会按照同样的一种方式处理；如果再一想要求严格的项目里面，异常一定要分开处理会更好。

章节29 课时 116 03097_异常的捕获及处理（throws关键字）

throws关键字主要用于方法声明上，指的是当方法之中出现异常后交由被调用处来进行处理。
范例：使用throws
```
class MyMath{
  //由于存在有throws，那么就表示此方法里面产生的异常交给被调用处处理。
  public static int div(int x,int y)throws Exception{
    return x/y;
  }
}
public class TestDemo{
  public static void main(String args[]){
    try{
      System.out.println(MyMath.div(10,2));
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```
也就是说调用了具有throws声明的方法之后，不管操作是否出现异常，都必须使用try...catch进行处理。可是在程序之中主方法也属于方法，那么主方法上能否继续使用throws抛出异常呢？

```
class MyMath{
  public static int div(int x,int y)throws Exception{
    return x/y;
  }
}
public class TestDemo{
  public static void main(String args[])throws Exception{
      System.out.println(MyMath.div(10,2));
  }
}
```
在主方法上如果出现了异常，那么这个异常就将交给JVM进行处理也就是采用默认的处理方式，输出异常信息，而后结束程序调用。
主方法上不要加throws，因为程序如果出错了，也希望可以正常的结束调用。

章节29 课时 117 03098_异常的捕获及处理（throw关键字）

在程序之中可以直接使用throw手工的抛出一个异常类的实例化对象。
```
public class TestDemo{
  public static void main(String args[]){
    try{
      throw new Exception("自己定义的异常！");
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```
异常肯定应该尽量回避，你没事非要自己再抛出个异常。
面试题：throw和throws的区别？
throw：指的是在方法之中人为抛出一个异常类对象（这个异常类对象可能是自己实例化或者是抛出已存在的）；
throws：指的是在方法的声明上使用，表示此方法在调用时必须处理异常。

章节29 课时 118 03099_异常的捕获及处理（异常处理标准格式）
现在要求定义一个div()方法，要求此方法在进行计算之前提示打印信息，在计算结束完毕也提示打印信息，如果在计算之中产生了异常，则交给被调用处进行处理。
范例：首先给出代码不出错的情况
```
class MyMath{
  public static int div(int x ,int y){
    int result=0;
    System.out.println("***1、除法计算开始！***");
    result=x/y;
    System.out.println("***2、除法计算结束！***");
    return result;
  }
}
public class TestDemo{
  public static void main(String args[]){
    System.out.println(MyMath.div(10,2));
  }
```

```
}


class MyMath{
  //此时表示div()方法如果出现了异常交给被调用处处理。
  public static int div(int x ,int y)throws Exception{
    int result=0;
    System.out.println("***1、除法计算开始！***");
    try{
      result=x/y;
    }catch(Exception e){
      throw e;//继续抛异常
    }finally{
      System.out.println("***2、除法计算结束！***");
    }
    return result;
  }
}
public class TestDemo{
  public static void main(String args[]){
    try{
      System.out.println(MyMath.div(10,0));
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
//可是如果以上的代码真的出错了呢？程序有些内容就不执行了，这样明显不对。
//以上代码可以简化，省略红字内容，结果没有任何不同。
```

章节29 课时 119 03100_异常的捕获及处理（RuntimeException类）

```
//下面首先来观察一段程序代码：
public class TestDemo{
  public static void main(String args[]){
    int temp=Integer.parseInt("100");
  }
}
```

现在来观察一下parseInt()方法的定义
public static int parseInt(String s)throws NumberFormatException
此时parseInt()方法上抛出了NumberFormatException,按道理来讲，应该进行强制性的异常捕获，可是现在并没有这种强制性的要求，来观察一下NumberFormatException的继承结构；
数字格式异常：
Class NumberFormatException
  java.lang.Object
    java.lang.Throwable
      java.lang.Exception
        java.lang.RuntimeException//表示运行时异常
          java.lang.IllegalArgumentException
            java.lang.NumberFormatException

在Java里面为了方便用户代码的编写，专门提供了一众RuntimeException类，这种异常类最大的特征在于：程序在编译的时候不会强制性要求用户处理异常，用户可以根据自己的需要选择性的进行处理，但是如果没有处理又发生异常了，将交给JVM默认处理，也就是说RuntimeException的子异常类，可以由用户根据需要选择进行处理。
面试题：请解释Exception与RuntimeException的区别？请列举出几个你常见的RuntimeException；
Exception是RuntimeException的父类
使用Exception定义的异常必须要被处理，而RuntimeException的异常可以选择性处理；
常见的RuntimeException：ArithmeticException,NullPointerException
,ClassCastException

章节29 课时 120 03101_异常的捕获及处理（断言）
assert关键字(了解)
在java中的断言指的是程序执行到某行代码处一定是预期的结果。
范例：观察断言
默认情况下断言是不应该影响程序的运行，也就是说在java解释程序的时候，断言是默认不起作用的。
启用断言：java -ea TestDemo

```
public class TestDemo{
```

```
  public static void main(String args[]){
    int num=10;
    //中间可能经过了20行代码来操作num的内容
    //期望中的内容应该是20
    assert num==20:"num的不是20";
    System.out.println("num="+num);
  }
}
```

章节29 课时 121 03102_异常的捕获及处理（自定义异常）

Java已经提供了大量的异常，但是这些异常在实际的工作之中往往并不够去使用，例如：当你要执行数据增加操作的时候，有可能会出现一些错误的数据，而这些错误的数据一旦出现就应该抛出异常，如：addException
但是这样的异常Java并没有，所以就需要由用户自己去开发一个自己的异常类。
如果想要开发自定义的异常类可以选择继承Exception或者是RuntimeException
范例：定义AddException

```
class AddException extends Exception{
  public AddException(String msg){
    super(msg);
  }
}
public class TestDemo{
  public static void main(String args[]){
    int num=20;
    try{
      if(num>10){//出现了错误，应该产生异常。
        throw new AddException("数值传递的过大！");
      }
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

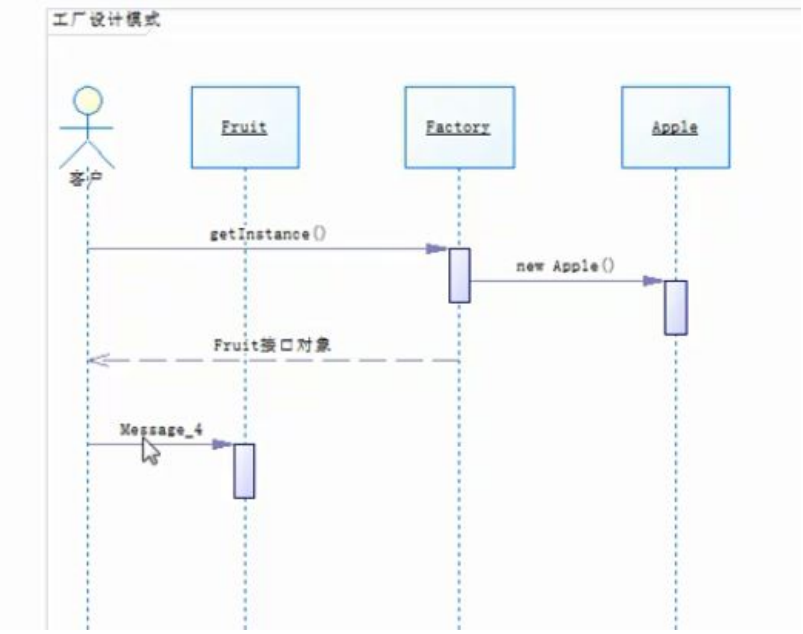
Exception的父类是Throwable，但是在编写代码的时候尽量不要使用Throwable，因为Throwable下还包含了一个Error子类，我们能够处理的只有Exception子类；
异常处理的标准格式：try catch finally throw throws
RuntimeException(强制)与Exception(有选择性的处理)的区别

章节30 课时 122 03103_类图描述（类图）
在软件设计上有一门课程——UML，它就是利用一系列的图形来描述项目结构、代码结构、执行顺序等等，现在的开发之中如果是一个开发的项目设计书上，一定要提供有UML图，但是很少再有人直接去画了。
如果要想描述类图一般有三个组成结构：
第一层：类名称，如果是抽象类使用斜体字；
第二次：描述类中的属性，对于属性肯定要封装如果是封装使用“-”表示
public(+)、protected(#)、private(-);
第三层：类中定义的方法。PowerDesigner

章节30 课时 123 03104_类图描述（时序图）

```
interface Fruit{
  public void eat();
}
class Apple implements Fruit{
  public void eat(){
    System.out.println("***吃苹果***");
  }
}
class Factory{
  public static Fruit getInstance(){
    return new Apple();
  }
}
public class TestDemo{
  public static void main(String args[]){
    Fruit f=Factory.getInstance();
    f.eat();
```

```
    }
}
```


工厂设计模式
Fruit
Factory
Apple
客户
getInstance()
new Apple()
Fruit接口对象
Message_4