1 数据类型

1.1 基本数据类型

byte short int long char float double boolean

基本类型	默认值	存储	封装类	数据范围
byte	0	1 字节	Byte	-128~127
short	0	2 字节	Shrot	-32768~32767
Int	0	4 字节	Integer	-21 亿~21 亿
long	0L(0l)	8 字节	Long	-2^63~2^63-1
char	‘\u0000’	2 字节	Character	0~65535
float	0.0F(0.0f)	4 字节	Float	32 位 EEE754 单精度
double	0.0(0.0d()D)	8 字节	Double	64 位 EEE754 双精度
boolean	false	1 位	Boolean	true/false

1.2 引用数据类型

接口 类 数组 String 等

1.3 数据类型转换

通常在进行整形数据处理时，会存在数据溢出，即数据过大或太小产生的错误数据的情况。这时候便需要进行数据类型转换。

描述小数默认类型为 double 类型，需要定义为 float 类型的时候需要在数字后加上 f 或者 F 如：1.2f。

1. 范围小的数据类型可以自动变为数据范围大的数据类型（在数学计算时）
2. 数据范围大的数据类型只有强制转换才能转为数据类型小的数据类型。

然后是字符类型，字符类型可以和 int 型的数据相互转换，但需要注意 char 类型中的‘0’～‘9’和数字 0~9 所代表的范围并不同。

最后是布尔类型，在 java 中的布尔类型与其他语言不同，在其他语言中用 0 代表 false!0 表示 true，但是在 java 中并没有这种表达方式，中能用 true 和 false 表示。

String 作为一种引用类型，其实是一个类，在后面的总结中会详细介绍。

2 运算符

2.1 基本运算符

+ - * / %

这里需要注意的是前置还是后置的问题，后置先运算，后进行自增自

减。前置相反。

2.2 三目运算符

（运算符，结果为 boolean 类型）？结果 1： 结果 2;

例如: 1 > 2 ? x- y: y - x;

等价于

If(1>2){

x-y;

}else{

y-x;

}

2.3 关系运算符

> < = <= >= == :返回类型均为 boolean 类型（注意在比较引用类型的时候不可以用==，而应该使用.equals 方法，这个在后面会详细介绍）

2.4 逻辑运算符

(&和&&为与) (|和||为或) (!为非)

其中|| 和 &&代表短路或和短路与，在进行逻辑判断的时候都运用短路与和或。

2.5 位运算符

& | ~ ^分别代表按位与，按位或，按位取反，以及按位异或

&:同为 1 才为 1，其余为 0

| : 有一个 1 就为 1，反之为 0

~ : 1 变为 0，0 变为 1

^: 相同为 0，不同为 1

3 Java 中的逻辑控制

3.1 分支结构

If(布尔表达式){

true：表达式 1;

}else{

false：表达式 2;

}

可以一个 if 对应多个 else 代表不同分支，也支持嵌套。

switch(数字|枚举|字符|字符串){

case 内容 1 : {

内容满足时执行语句;

[break;]

} case 内容 2 : {

内容满足时执行语句;

[break;]

} ... default:{

内容都不满足时执行语句;

[break;]

}

}

需要注意，break 通常不能省略。

3.2 循环结构

while 循环

white（循环条件）{

循环体；

更新循环变量；

}

do...while()循环

do{

循环体；

更新循环变量；

}while(循环条件)

for()循环

for（循环变量初始值;循环终止条件;循环标量更新）{

循环体;

}

3.3 循环控制

break 直接结束循环循环。

continue 跳出本次循环，进行下一次循环条件判断。

4 方法的定义和使用

4.1 方法的定义

方法是一段可以被重复调用代码，格式如下：

访问权限修饰符 (static) (final)返回值类型 方法名（参数列表）{

方法体代码;

return 返回值;

}

4.2 方法的重载

在同一个类中方法名相同，参数列表不同（顺序或个数）的方法，需要注意，返回值类型不做要求，也就是说，不能存在只有返回值类型不同的两个方法。

4.3 方法的递归

递归就是方法自己调用自己，需要有一个递归的结束条件，而且每次递归的参数需要做一定的调整，保证能越来越趋近于结束条件。所有的分治思想，都可以采用递归的方式去实现。

4.4 方法的调用

一般的方法都是需要通过实例化对象来调用的，但是加了 static 修饰的方法，不依赖于对象，可以直接通过类名调用。具体的会在面向对象的时候详细介绍。

5 面向对象

在《java 核心技术 卷一》这本书中对面向对象是这样定义的：面向对象的程序是由对象组成的， 每个对象包含对用户公开的特定功能部分和隐藏的实 现部分。程序中的很多对象来自标准库，还有一些是自定义的。究竟是自己构造对象， 还是 从外界购买对象完全取决于开发项目的预算和时间。但是， 从根本上说， 只要对象能够满足 要求，就不必关心其功能的具体实现过程。在 OOP 中， 不必关心对象的具体实现，只要能 够满足用户的需求即可。

所以我们了解到在面向对象中引入了类与对象的概念。

5.1 类

类是构造对象的模板，在类中定义了对象的属性以及行为。我们可以将类想象成打造铁器的模具，它规定了打造出来的铁器的形状与大小。而对象就是打造出来的铁器。打造铁器的这个过程，我们将其称为类的实例化。在平常开发中我们几乎所有的代码都在某个类的内部，在 java 的类库中提供了很多的类，为我们日常的开发提供了很多帮助，但是这些类想要达到我们的目的是远远不够的，所以我们还要自己创造更多的类，来达到我们想要的结果。

封装是面向对象有的一个重要概念。从形式上看， 封装不过是将数据和行为组合在一个包中， 并对对象的使用者隐藏了数据的实现方式。对象 中的数据称为实例代码块，操纵数据的过程称为方法，对于每个特定的 类实例（对象）都有一组特定的实例域值。这些值的集合就是这个对象的当前状态。 无论何时，只要向对象发送一个消息，它的状态就有可能发生改变。 实现封装的关键在于绝对不能让类中的方法直接地访问其他类的实例域。程序仅通过对 象的方法与对象数据进行交互。封装给对象赋予了“ 黑盒” 特征， 这是提高重用性和可靠性的关键。 这意味着一个类可以全面地改变存储数据的方式，只要仍旧使用同样的方法操作数据， 其他对象就不会知道或介意所发生的变化。

5.2 对象

同一个类的所有实例化对象具有类似的属性和行为，这些行为需

要对象调用方法来实现，我们可以用一个类来描述一个对象，但是对象却不能表示一个类，这就相当于现实中，可以把桃子说成是水果，但是却不能说水果就是桃子。这里的水果就相当于一个类，而桃子便是类的实例化对象。

我们下来就用一段代码来表示对象的实例化过程化，以及调用成员方法的过程。

```
class Student{
  private String name;//学生的姓名，private 表示私有
private int age;//学生的年龄
private String school;//学校名称
public Student(){
//构造函数
}
public Student(String name,int age, String school){
//构造函数的重载
  //this  表示当前对象的引用，可以调用本类属性和方法
this.name=name;//this  调用本类属性
this.age=age; this.school=school; this.read;//this 调用本类方法 } public void read(){
//成员方法
System.out.println("读书");
  }
```

}

在类的实例化中，我们只需要使用 new 关键字来进行实例化，这一过程包括两个步骤：1.开辟内存空间（new 关键字实现）

2. 调用合适的构造方法。代码如下：（省略主类代码）：

Student s1=new Student();//(调用不带参数的构造方法)

Student s2=new Student(“李雷”,25,”清华大学”);//(调用带有参数的方法)

s1.read;

s2.read;//调用类的成员方法，即对象的行为。

5.3 继承

OOP 的另一个原则会让用户自定义 Java 类变得轻而易举，这就是：可以通过扩展一个 类来建立另外一个新的类。事实上， 在 Java 中， 所有的类都源自于一个 “ 神通广大的超类” ， 它就 Object。而这个过程就是面向对象的另一个重要特征：继承。利用继承，我们可以基于已存在的类构造一个新类。继承已存在的类就是复用（继承）这些类的方法和域。在此基础上，还可以添加一些新的方法和域， 以满足新的需求。这是 Java 程序设计中的一项核心技术。

在 java 中一个类只能继承自一个类，也就是单继承，需要实现多继承，我们需要借助接口来实现。我们下面来看一段继承的代码;

class Person{

```
private String name;

private int age;

static {//静态代码块，在类加载时就就执行了，并且只执行一次

}

public Person(String name,int age){//调用合适的构造函数


this.name=name;//this  关键字表示当前对象的引用

this.age=age;

}

public void setName(){//提供一系列的 get set 方法

this.name=name;
```

```
        }

    }


        class Student extends Person{

        private String school;//对父类的属性进行扩充

        static{ //静态代码块

        }

        public Student(String name, int age,String school){

        super(name,age);//利用 super 调用父类构造函数。必须写在第一行。

        this.school=school;

        }
```

```
    public void func(){ //子类的成员方法


    }
}
```

在这段代码中我们可以发现，继承只继承了属性和成员方法，并没有继承父类的构造方法。并且在子类中可以用 super 关键字来调用父类的属性和方法。注意 用 final 修饰的类不可以被继承。

5.4 多态

多态是面向对象的第三个重要的特征，多态的体现主要体现在方法的多态和对象的多态。

方法的多态：方法的重载与覆写都是多态的体现。

对象的多态：在抽象类和接口中，一个类实现了抽象类的抽象方法，后者接口，在进行实例化的时候，发生向上转型。（前提条件是发生了方法的覆写），代码示例如下：

```
class Person{

    private String name;

    private int age;

    static {//静态代码块，在类加载时就就执行了，并且只执行一次
```

```
    }

    public Person(String name,int age){//调用合适的构造函数


    this.name=name;//this 关键字表示当前对象的引用

    this.age=age;

    }

    public void setName(){//提供一系列的 get set 方法

    this.name=name;

    }

}


    class Student extends Person{
```

```
    private String school;//对父类的属性进行扩充

    static{ //静态代码块

    }

    public Student(String name, int age,String school){

    super(name,age);//利用 super 调用父类构造函数。必须写在第一行。

    this.school=school;

    }

    public void func(){ //子类的成员方法

    }

}
```

多态是面向对象编程的一个重要特征，其本质是在方法区存放的方法表中，在向上转型中，子类的方法覆盖了父类的方法，从而最终的结果为调用的子类的方法。

5.5 代码执行顺序

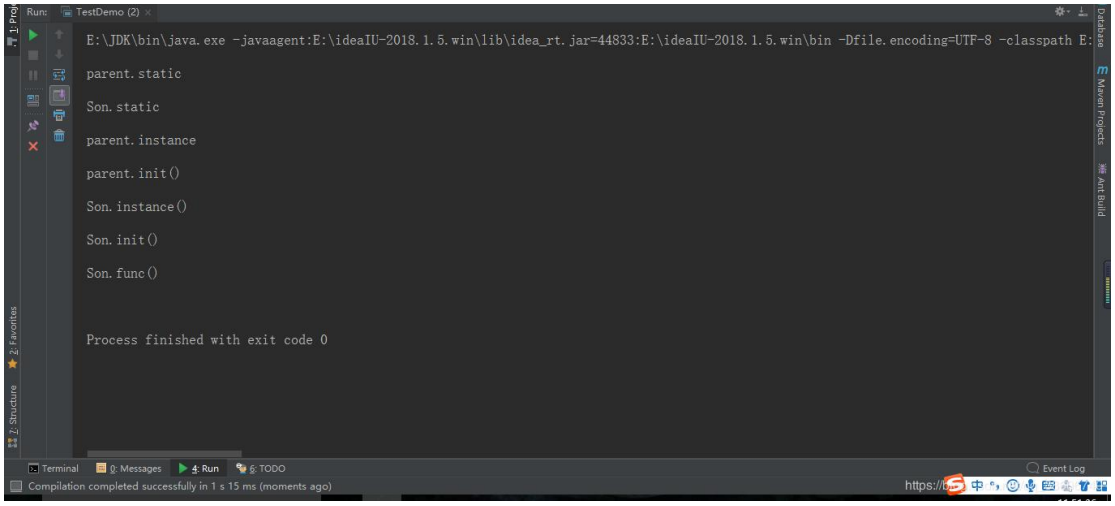
在面向对象的过程中，发生了继承关系，往往代码的执行顺序会变得比较复杂，下面我们来用一段代码来分析一下，在这个过程中代码的执行顺序究竟是什么样的。

```
public class Parent {
    private int age;
    private String name;
    public Parent(){
        System.out.println("parent.init()");
    }
    {
        System.out.println("parent.instance");
    }
    static {
        System.out.println("parent.static");
    }
    public void func(){
        System.out.println("parent.func()");
```

```
    }
}
public class Son extends Parent{
    private String school;
    public Son(){
        System.out.println("Son.init()");
    }
    {
        System.out.println("Son.instance()");
    }
    static{
        System.out.println("Son.static");
    }
    public void func(){
        System.out.println("Son.func()");
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Son son=new Son();
        Parent parent=new Son();
        parent.func();
```

```
        son.func();
    }
}
```

这段代码的运行结果如下图，



从这个运行结果我们可以看出来首先执行父类的静态代码块，然后子类静态代码块，接下来是父类属性已经构造方法，最后是子类属性和构造方法。成员方法看调用顺序。

5.6 内部类

内部类顾名思义就是一个类的内部嵌套另外一个类，利用内部类可以方便操作外部类的私有成员，但是却会破坏程序的结构，使程序的逻辑显得比较混乱。

下面我们便来看一下内部类的具体分类。

5.6.1 成员内部类

成员内部类就是在一个类的内部嵌套一个普通类，需要注意成员内部类定义静态成员变量的时候必须背 final 所修饰。具体代码如下：

```
//代码示例：
class OuterClass{
    private int age;
    private String name;
    public static int data=11;
    public OuterClass(){
        System.out.println("外部类构造函数");
    }
    class Inter{
        //实例内部类定义静态成员变量必须使用 final 修饰
        public static final int age=10;
        public Inter(){
            System.out.println("内部类构造函数");
        }
        public Inter(int age,String name){
//实例内部了消耗了额外的内存空间，因为实例内部类具有外部类的
```

this 引用

```
        OuterClass.this.age=age;
        OuterClass.this.name=name;
            }
        public void print(){
            System.out.println("内部类成员方法");
        }
        }
        public void print(){
            System.out.println("外部类成员方法");
        }
    }
    public class TestDemo1 {
        public static void main(String[] args) {
//实例内部类的实例化需要借助外部类对象
        OuterClass.Inter  inter=new  OuterClass().new  Inter();
        inter.print();
        }
    }
```

5.6.2 静态内部类

静态内部类，静态内部类就是外部类嵌套了的被 static 修饰的类。由于被 static 所修饰，所以静态内部类的实例化可以不需要借助外部类对象来创建。代码如下：

```
class OuterClass{
    private int age;
    private String name;
    public static int data=11;
    public OuterClass(){
        System.out.println("外部类构造函数");
    }
    static class Inter{
        public    static int age=10;//静态内部类定义静态成员变量不需要使用 final 修饰
        public Inter(){
            System.out.println("内部类构造函数");
        }
    public void print(){
        System.out.println("外部类成员方法");
    }
    }
```

```
        public void print(){
            System.out.println("内部部类成员方法");
        }
    }
    public class TestDemo1 {
        public static void main(String[] args) {
        //静态内部类的实例化
        OuterClass.Inter inter = new OuterClass.Inter();
        inter.print();
        }
    }
```

5.6.3 方法内部类

方法内部类定义在外部类的方法中,局部内部类和成员内部类基本一致，只是它们的作用域不同，方法 内部类只能在该方法中被使用，出了该方法就会失效。 对于这个类的使用主要是应用与解决比较复杂的问题，想创建一个类来辅助我们的解决方案，到那时又不希望这个类是公共可用的，所以就产生了局部内部类也就是方法内部类。需要注意方法内部类不允许使用任何访问修饰符，代码如下：

```
class OuterClass{
    private int age;
    private String name;
    public static int data=11;
    public OuterClass(){
        System.out.println("外部类构造函数");
    }
    public void print(){
        class Inter{//本地内部类，不允许被修饰符修饰
            public    static final int age=10;
            public Inter(){
                System.out.println("内部类构造函数");
            }
        }
        new Inter();
    }
}
public class TestDemo1 {
    public static void main(String[] args) {
        OuterClass outerClass=new OuterClass();
        outerClass.print();
    }}
```

5.6.4 匿名内部类

匿名内部类其实就是一个没有名字的方法内部类 ,所以它符合方法内部类的所有约束.除此之外，还有一
些地方需要注意:

1. 匿名内部类是没有访问修饰符的。
2. 匿名内部类必须继承一个抽象类或者实现一个接口
3. 匿名内部类中不能存在任何静态成员或方法
4. 匿名内部类是没有构造方法的，因为它没有类名。
5. 与局部内部相同匿名内部类也可以引用方法形参。此形参也必须声明为 final

代码如下：

```
interface    Myinterface{
      void print();
            }
class OuterClass{
private int age;
public String name;
public void Myinterface(){
      new Myinterface() {//不允许使用访问修饰符修饰
            @Override
            public void print() {
```

```
                System.out.println("匿名内部类实现接口");
            }
        }.print();
        }
}
public class TestDemo1 {
        public static void main(String[] args) {
            OuterClass outerClass=new OuterClass();
            outerClass.Myinterface();
        }
}
```

5.6.5 内部类的执行顺序

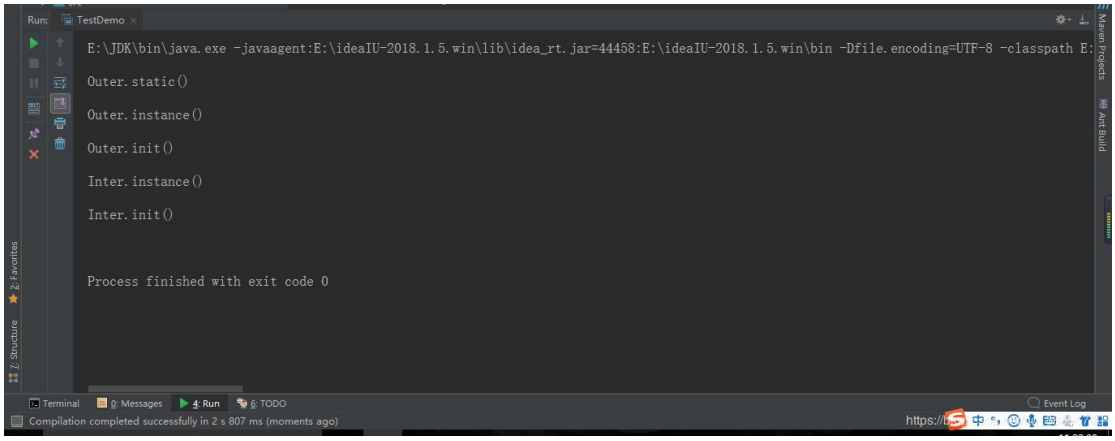
内部类的使用往往使得程序的结构变得混乱，所以我们需要看一下程序的执行顺序，我们借助代码来看一下。

```
public class Outer {
        private int age;
        private String name;
        public Outer(){
            System.out.println("Outer.init()");
        }
```

```
{
        System.out.println("Outer.instance()");
}
static {
        System.out.println("Outer.static()");
}
public void func(){
        System.out.println(name+age);
}
class Inter{
        public static final int age2=10;
        private String name2;
        public Inter(){
                System.out.println("Inter.init()");
        }
        {
                System.out.println("Inter.instance()");
        }
        public void func(){
                System.out.println(name+age);
        }
}
```

```
}
public class TestDemo {
    public static void main(String[] args) {
        Outer.Inter inter=new Outer().new Inter();
    }
}
```

运行结果：



从运行结果我们可以看出，首先执行的是外部类的静态代码块，外部类实例代码块，外部类构造函数，然后是内部类实例代码块，内部类构造函数。最后还有内部类的成员方法，代码中并没有加入。需要注意的是，实例内部类也可以定义静态的成员属性，不过需要用

final 修饰，代表这个属性在编译时期就已经确定下来了，不能再修改。静态内部类大体和实例内部类相似，不过内部类要调用外部类成员，需要提供有外部类引用的构造函数。

6 java 中的数组

6.1 数组的定义

Java 中的数组与 C 语言中的类似，都是一组相关类型的变量集合，并且这一系列变量可以按统一的方式操作。

首先是数组的定义，

语法格式是：类型[] 数组名称=new 类型[数组长度];

```
//代码示例：
  int[] arr=new int[10];//定义一个长度为 10 的数组 ,下标从 0-9.
  arr[0]=1;//动态初始化。
  arr[1]=2;
  int[] arr2={0,1,2};//静态初始化，在 Java 中未初始化的数组默认填充为 0
```

注意在使用时不能发生数组越界 ,即使用的空间超过了数组所开辟的空间。同时也必须为数组开辟空间，即实例化之后才可以使用。数组在内存中的存储是，数组的引用即首元素的地址存放在栈上，数组元素存放在堆上，同一块堆可以被不同的引用指向（这也是浅拷贝的基础）

代码示例：

```
int[]arr1=null;
int[] arr2=null;
arr1[0]=1;
arr1[1]=2;
arr1[2]=3;
arr2=arr1;
arr2[0]=30;
```

最后的结果是 arr1[0]的值为 30，因为将 arr1 赋给 arr2 即他两个指向同一个数组，arr2 元素改变 arr1 中的元素也相应的发生了变化。java 中的二维数组与 C 语言中的不同，java 中二维数组是不规则的。代码示例：

```
int[2][3]arr={{1,2}{1,2,3}}
```

这个数组在 java 中指的是有两个一维数组，每个一维数组最长为三个元素，但不一定是三个。数组也可以通过方法进行处理。代码示例：

```
class ArrDemo{

public static void showArray(int []arr){//方法接收数组
for(int i=0;i<arr.length;i++){//arr.length 返回数组的长度
System.out.println(arr[i]);
}
```

```
    }
    public static int[] init(){//方法返回数组
    int[]arr1={1,2,3,4};
    return arr;
    }
}
```

6.2 java 对数组的支持

首先是数组的四种拷贝方式

（1）利用 for 循环拷贝，这种方式和 C 语言中相同，不再赘述。

（2）Arrays.copy,利用 Arrays 类中的 copy 方法实现对数组的拷贝，代码示例：

```
int[]array={1,2,3,4,5,6,7,8,9}
int[]array2=Arrays.copyof(array,array.length);//参数，（原数组，拷贝长度）
```

（3）System.arrayCopy,System 类中的 arrayCopy 方法，用 native 修饰（这种拷贝速度最快）代码示例

```
System.arrayCopy(array,0,array2,0,array.length);
```

（4）clone 代码示例

```
array2=array.clone();
```

这四种拷贝方式早拷贝基本类型的时候，都是深拷贝，只有在引用类型的时候是浅拷贝。

然后是 Arrays 类中对数组的操作支持

Arrays.binarySearch(arr,0,arr.length,5); /*二分查找法查找数组中的元素,如果找到

返回元素下标，如果没找到就以负数返回该值将要插入的位置，参数分别为，需要查找的数组

查找的起始位置，查找的结束位置，查找的值*/

arr2=Arrays.copyOf(arr,4); //拷贝数组为一个新长度，截取或者用 0 填充

arr2=Arrays.copyOfRange(arr,0,3); //拷贝数组的指定范围

Arrays.equals(arr,arr2); //判断两个数组深层是否相等，返回布尔类型

Arrays.fill(arr2,2,5,4); /*将指定的值填充到数组的指定位置，参数分别为，被填充

数组，其实填充位置，终止填充位置（不包括），填充值*/

Arrays.sout(arr); //对 arr 数组进行升序排序

System.out.println(Arrays.toString(arr))

//将 arr 中的内容以字符串的形式返回。

7 抽象类与接口

7.1 抽象类的定义和使用。

抽象类只是在普通类的基础上扩充了一些抽象方法而已，所谓的抽象方法指的是只声明而未实现的方法 （即没有方法体）。所有抽象方法要求使用**abstract** 关键字来定义，并且抽象方法所在的类也一定要使 用**abstract。代码示例：**

```
abstract class Person{
private String name ; // 属性
public String getName(){ // 普通方法
return this.name;
}
public void setName(String name){
this.name = name ;
}
// {}为方法体，所有抽象方法上不包含方法体
public abstract void getPersonInfo() ; //抽象方法
}
```

在使用抽象类的时候，需要注意，抽象类中并不一定有抽象方法，但抽象方法必定是在抽象类或者接口里面的。再定义一个抽象类时候，这个抽象类必须有子类，在一个非抽象子类中，必须要实现当前抽象类的抽象方法，这也就规定了抽象类在定义时不允许使用 final

修饰。抽象类本身不能实例化，但可以借助子类利用多态性实现实例化。同时抽象类由于被继承，所以其子类同样遵循继承原则。

7.2 接口的定义与使用

抽象类与普通类相比最大的特点是约定了子类的实现要求，但是抽象类存在单继承局限。如果要约定子 类的实现要求并避免单继承局限就需要使用接口。 在开发过程之中：接口优先（在一个操作既可以使用抽象类又可以使用接口的时候，优先考虑使 用接口）

代码如下：

```
interface IMessage{
public static final String MSG = "Hello World" ; // 全局常量
public abstract void print() ; // 抽象方法
}
```

子类如果要想使用接口，那么就必须使用implements 关键字来实现接口，同时，一个子类可以实现多 个接口，【可以使用接口来实现多继承的概念】对于接口的子类（不是抽象类）必须覆写接口中的全部抽象方法。

代码如下：

```
interface IMessage{
public static final String MSG = "I am a biter" ; // 全局常量
public abstract void print() ; // 抽象方法
}
```

```
interface INews {
public abstract String getNews() ;
}
class MessageImpl implements IMessage,INews {
public void print() {
System.out.println(IMessage.MSG) ;
}
public String getNews(){
return IMessage.MSG ; // 访问常量都建议加上类名称
}
}
public class Test{
public static void main(String[] args) {
IMessage m = new MessageImpl() ; //子类向上转型,为父接口实例化对象
m.print() ; // 调用被子类覆写过的方法
INews n = (INews) m ;
System.out.println(n.getNews()) ;
}
```

在这里就是多态性的一个体现，实际产生的对象是 new 产生的实例化对象，而不是前面的那个。这就好比说，水果 桃子=new 桃子。最终产生的对象是桃子一样。在接口中，所有的属性和方法都是

public 的，所以 weight 保持代码的简洁性，我们在使用接口时，通常不写 public。

7.3 接口和抽象类的区别。

no	区别	抽象类	接口
1	结构组成	普通类+抽象方法	抽象方法+全局变量
2	权限	各种权限	public
3	子类使用	extends	implements
4	关系	可以实现多个接口	不能继承抽象类，可以继承多个接口
5	子类限制	单继承	利用接口实现多继承

7.4 设计模式

7.4.1 模块设计模式

模块设计模式是抽象类的一个实际运用场景，比如说平常煮茶时的几个步骤。

1. 将水煮沸
2. 用沸水浸泡茶叶
3. 把茶倒进杯子

4. 加佐料

u示例代码：

```
class Tea {
void prepareRecipe() {
boilWater();
steepTeaBag();
pourInCp();
addLemon();
}
```

将水煮沸

```
public void boilWater() {
System.out.println("Boiling Water");
}
```

冲泡茶

```
public void steepTeaBag() {
System.out.println("Steeping the tea");
}
```

把茶倒进杯子中

```
public void pourInCup() {
System.out.println("Pouring into cup");
}
```

加佐料

```
public void addSeasoning() {
System.out.println("Adding Seasoning");
}
}
```

我们可以发现，其实在冲泡奶茶咖啡等饮料时其实都是使用的同样的操作，所以我们便可以将这作为一个模板，来实现不同的饮料制作。

代码如下：

```
* 基类声明为抽象类的原因是
* 其子类必须实现其操作
*/
abstract class AbstractClass {
/**
* 模板方法，被声明为 final 以免子类改变这个算法的顺序
*/
final void templateMethod() {
}
/**
* 具体操作延迟到子类中实现
*/
abstract void primitiveOperation1();
abstract void primitiveOperation2();
```

```
/**
* 具体操作且共用的方法定义在超类中，可以被模板方法或子类直接使用
*/
final void concreteOperation() {
// 实现
}
/**
* 钩子方法是一类"默认不做事的方法"
* 子类可以视情况决定要不要覆盖它们。
*/
void hook() {
// 钩子方法
}
}
```

扩展上述类，引入"钩子"方法

超类实现:

```
abstract class CaffeineBeverage {
final void prepareRecipe() {
boilWater();
brew();
pourInCup();
```

```
// 如果顾客想要饮料我们才调用加料方法
if (customerWantsCondiments()){
addCondiments();
}
}
abstract void brew(); abstract void addCondiments();
void boilWater() {
System.out.println("Boiling water");
}
void pourInCup() {
System.out.println("Pouring into cup");
}
/**
* 钩子方法
* 超类中通常是默认实现
* 子类可以选择性的覆写此方法
* @return
*/
boolean customerWantsCondiments() {
return true;
}
}
```

子类实现:

```
class Tea extends CaffeineBeverage {
void brew() {
System.out.println("Steeping the tea");
}
void addCondiments() {
System.out.println("Adding Lemon");
}
}
class Coffee extends CaffeineBeverage {
void brew() {
System.out.println("Dripping Coffee through filter");
}
void addCondiments() {
System.out.println("Adding Sugar and Milk");
}
/**
*  子类覆写了钩子函数，实现自定义功能
* @return
*/
public boolean customerWantsCondiments() {
String answer = getUserInput();
```

```
if (answer.equals("y")) {
return true; }else {
return false;
}
}
private String getUserInput() {
String answer = null;
System.out.println(" 您 想 要 在 咖 啡 中 加 入 牛 奶 或 糖 吗 (y/n)?");
Scanner scanner = new Scanner(System.in);
answer = scanner.nextLine();
return answer;
}
}
```

测试类:

```
public class Test {
public static void main(String[] args) {
CaffeineBeverage tea = new Tea();
CaffeineBeverage coffee = new Coffee();
System.out.println("\nMaking tea...");
tea.prepareRecipe();
System.out.println("\nMaking Coffee");
```

```
coffee.prepareRecipe();
}
```

通过这个例子，我们可以发现，其实很多具有相同操作的过程，都可以使用模板设计模式来使这些代码变得更加简单一些。

7.4.2 简单工厂模式

简单工厂模式。有一个抽象的产品的接口，需要具体的产品类去实现这个接口，然后需要有一个工厂类去产生具体产品。最后用客户端去实现买电脑。

代码如下：

```
public interface Computer {//首先定义一个电脑接口
    void print();
}
class Macbook implements Computer{//具体产品类
    @Override
    public void print() {//MacBook 去实现 Computer 接口
        System.out.println("MacBook");
    }
}
class SurfaceBook implements Computer{//具体产品类
    @Override
    public void print() {//SurfaceBook 去实现 Computer 接口
        System.out.println("SurfaceBook");
    }
}
class ComputeFactory {//产品工厂类
    public static Computer getInstance(String type){//返回一种具体产品
        Computer computer=null;
        switch (type){
            case "Mac":
```

```
                    computer=new  Macbook();break;// 生产
Macbook
                case "Sur":
                    computer=new  SurfaceBook();break;//生
产 SurfaceBook
                default:
                    System.out.println("请输入型号");break;//
其他型号
                }
            return computer;//返回生产的电脑
        }
    }
    import java.util.Scanner;//导入 Scanner 类

    public class Client {//客户端类
        public static void Buy(Computer computer){//接收电脑
型号买电脑
            computer.print();
        }

        public static void main(String[] args) {
            Scanner scanner=new Scanner(System.in);
```

String type=scanner.nextLine();//用户输入需要购买的型号

Computer computer=ComputeFactory.getInstance(type);//根据用户输入型号生产电脑

Buy(computer);调用买电脑方法，传过去用户需要购买的型号买到电脑。

}

}

简单工厂模式的优点顾名思义就是简单，缺点是，在工厂中添加其他型号电脑时违反了 OCP 封装原则。

7.4.3 工厂方法模式

工厂方法模式。有一个产品接口 ,需要具体的产品实现产品接口，有一个工厂接口 ,需要具体的工厂实现工厂接口。同样用代码来详解。

```
public interface Computer {//产品接口
    void print();
}
public interface ComputerFactory {//工厂接口
    Computer CreatComputer();//生产一种具体产品
}
public class MacBook implements Computer{//具体产品类
```

```
MacBook
        @Override
        public void print() {//实现产品接口
            System.out.println("MacBook");
        }
    }
    public class SurfaceBook implements Computer {//具体产品类 SurfaceBook
        @Override
        public void print() {//实现产品接口
            System.out.println("SurfaceBook");
        }
    }
    public class MacFactory implements ComputerFactory {//具体工厂类 MacFactory
        @Override
        public Computer CreatComputer() {//实现工厂接口
            return new MacBook();//生产 MacBook
        }
    }
    public class SurfaceFactory implements ComputerFactory {//具体工厂类 SurfaceFactory
```

```
        @Override
        public Computer CreatComputer() {
            return new SurfaceBook();//生产 SurfaceBook
        }
    }
    public class Client {//客户端类，实现购买电脑
        public void BuyComputer(Computer computer){//根据型号购买电脑
            computer.print();
        }

        public static void main(String[] args) {
            Client client=new Client();
            ComputerFactory computerFactory=new SurfaceFactory();//生产 SurfaceBook

client.BuyComputer(computerFactory.CreatComputer());//购买生产出的电脑
        }
    }
```

工厂方法类的优点是降低了代码的耦合度，实现了开放封闭原则，添加新的产品类的时候不需要修改原有的代码，缺点同样也很明

显，代码量增多，并且在添加抽象产品的时候，比如 Mac 添加苹果手机，同样会违反 OCP 封装原则。

7.4.4 抽象工厂模式

```
public interface Computer {//抽象产品接口电脑
    void print();
}
public interface OpreatSystem {//抽象产品接口操作系统
    void print();
}
public interface Factory {//抽象工厂接口
    Computer createComputer();//生产电脑
    OpreatSystem createOpreatSystem();//安装电脑操作系统
}
class MacBook implements Computer {// 具体产品类 MacBook
    @Override
    public void print() {
        System.out.println("MacBook");
    }
}
```

```
    class SurfaceBook implements Computer {//具体产品类 SurfaceBook
        @Override
        public void print() {
            System.out.println("SurfaceBook");
        }
    }
    class MacFactory implements Factory {//具体工厂类 Mac 工厂
        @Override
        public Computer createComputer() {
            return new MacBook();//生产 MacBook
        }
        @Override
        public OpreatSystem createOpreatSystem() {
            return new MacOs();//安装 MacOs 系统
        }
    }
    class SurfaceFactory implements Factory {// 具体工厂 Surface 工厂
        @Override
        public Computer createComputer() {
```

```
            return new SurfaceBook();//生产 SurfaceBook
        }

        @Override
        public OpreatSystem createOpreatSystem() {
            return new Win10();//安装 Win10 系统
        }
    }
    public class Client {
        public void BuyComputer(Computer computer){
            computer.print();//购买电脑的型号
        }
        public void Os(OpreatSystem opreatSystem){
            opreatSystem.print();//购买电脑的操作系统
        }

        public static void main(String[] args) {
            Client client=new Client();
            Factory factory=new MacFactory();// 使 用
MacFactory
            client.BuyComputer(factory.createComputer());//
买 MacFactory 生产出的电脑
```

```
            client.Os(factory.createOpreatSystem());// 购 买 的
电脑操作系统是
        }
    }
```

抽象工厂模式总体上和工厂方法模式差不多，是对工厂方法模式的扩充，抽象工厂接口可以灵活扩充，缺点也同工厂方法模式，并且代码量更大，在扩展具体产品时同样会违反 OCP 封装原则。

7.4.5 代理设计模式

```
interface ISubject {
public void buyComputer() ; //  核心功能是买电脑
}
class RealSubject implements ISubject {
public void buyComputer() {
System.out.println("买一台外星人电脑") ;
}
}
class ProxySubject implements ISubject {
private ISubject subject ; //  真正的操作业务
public ProxySubject(ISubject subject) {
this.subject = subject ;
```

```
}
public void produceComputer() {
System.out.println("1.生产外星人电脑") ;
}
public void afterSale() {
System.out.println("3.外星人电脑售后团队") ;
}
public void buyComputer() {
this.produceComputer() ; // 真实操作前的准备
this.subject.buyComputer() ; // 调用真实业务
this.afterSale() ; // 操作后的收尾
}
}
class Factory {
public static ISubject getInstance(){
return new ProxySubject(new RealSubject()) ;
}
}
public class Test{ public static void main(String[] args) {
ISubject subject = Factory.getInstance() ;
subject.buyComputer() ;
}
```

代理模式的本质 :所有的真实业务操作都会有一个与之辅助的工具类（功能类）共同完成。

8 java 特殊类

8.1 Object 类

Object 类是 java 中默认提供的一个类，在 java 中，所有的类都默认继承自 Object 类，它是所有类的父类。Object 可以接收所有类的对象。

在 Object 类中为我们提供了很多的基本方法，在我们使用时可以直接调用这些方法，但是有的方法比如 toString 这个方法，返回值为一个哈希码，所以在使用时需要对这个方法进行覆写。

8.2 String 类

8.2.1 String 基本

String 在我们前面讲数据类型的时候讲过 ,String 是一个引用类型，其实 String 本质上是一个特殊的类。我们平常使用 String 是都是

String str =“Hello World”;这是 String 最常用的一种初始化方式，但是 String 作为一个类，必然是有自己构造方法的，所以也可以用 String str=new String(“Hello World “)；

那么这两种实例化方式有什么区别呢 ,在我们了解到这两种那个

实例化方式的底层的时候,发现使用第二种实例化方法会产生一块垃圾空间，需要我们使用.inter()；方法来进行手动入池，而第一种实例化方法只初始化一次,字符串放在对象池中,在下次使用字符串时候，只需要在对象池中找有没有该字符串，如果没有再开辟新的内存空间。另外字符串每进行一次更改就需要开辟一块新的内存空间，所以在使用的时候最好不要多次修改。

字符串比较，我们前面所接触到的比较都是使用比较运算符==来进行比较的，但我们使用==对字符串进行比较时：

```
String str1 = "Hello";
String str = new String("Hello");
System.out.println(str1==str); // false
```

== 本身是进行数值比较的，如果现在用于对象比较，那么所比较的就应该是两个对象所保存的内存地址。那么我们进行字符串进行比较时，就需要调用我们字符串所提供的.equals 方法了。

在任何的语言的底层，都不会提供有直接的字符串类型。现在所谓的字符串只是高级语言提供给用户方 便开发的支持而已。在 java 之中，本身也没有直接提供字符串常量的概念，所有使用 “” 定义的内容本质上来讲都是 String 的匿名对象。

8.2.2 Sting 类中的方法

```
public class TestDemo {
    public static void main(String[] args) {
```

```
        char[] value={'a','b','c','d'};
        String str =new String(value);
        String str2=new String(value,1,3);
        System.out.println(str2);
        String str3="abcde";
        char ch=str3.charAt(3);//取得指定下标的值（默认从0号位置开始找）
        System.out.println(ch);
        char[] chars=str3.toCharArray();//将字符串转换为字符数组
        System.out.println(Arrays.toString(chars));
        String str4="dahda";
        System.out.println(isNumber(str4));
        byte[] bytes={97,98,99,100};
        String s=new String(bytes,1,3);
        System.out.println(s);
        byte[] bytes1=s.getBytes();
        System.out.println(Arrays.toString(bytes1));
        String str5="HELLO";
        String str6="Hello";
        System.out.println(str5.equals(str6));
        System.out.println(str5.equalsIgnoreCase(str6));
```

```
        System.out.println(str5.compareTo(str6));//字符串比较，找到第一个不相同的字符，进行比较
        //如果长度不一样，先结束的字符串小。
        System.out.println(str6.contains("lo"));//查找字符串中是否存在另一个字符串返回 Boolea 类型
        System.out.println(str6.indexOf("lle",2));//从默认位置查找是否存在另一个字符串返回第一个匹配的下标
        //找不到返回-1
        System.out.println(str6.lastIndexOf("llo",3));//从后向前找
        System.out.println(str5.startsWith("H"));//判断字符串是否以指定参数开始
        System.out.println(str5.endsWith("LO"));//判断字符串是否以指定参数结束
        System.out.println(str5.replace(str6,str5));
    }
    public   static Boolean isNumber(String str4){
        char[] arr=str4.toCharArray();
        for(int i=0;i<arr.length;i++){
            if('0'<=arr[i]&&arr[i]<='9'){
                return false;
            }
```

```
            }
            return true;
        }
    }
```

8.2.3 StringBuffer 与 StringBuilder

通过前面我们知道了 String 不可修改，更改会产生很多的垃圾空间，为了方便在开发时候的修改，我们采用 StringBuffer 与 StringBuilder 这两个类。String 和 StringBuffffer 最大的区别在于：String 的内容无法修改，而StringBuffffer 的内容可以修改。频繁修改字符串的情况考虑使用StingBuffer。

StringBuffer 与 StringBuilder 以及 String 的区别：

String 的内容不可修改，StringBuffffer 与 StringBuilder 的内容可以修改. StringBuffer 采用同步处理，属于线程安全操作；而 StringBuilder 采用异步处理，属于线程不安全 操作。

8.3 包装类

在前面讲数据类型的时候，每一种基本数据类型都对应着一个包装类 Java 为了方便开发，专门提供了包装类的使用，而对于包装类的使用，提供了两种类型。

对象型(Object 的直接子类):Boolean、Character(char)；

数值型(Number 的直接子类): Byte、Double、Short、Long、Integer(int) Float。

在包装类与基本数据类型转换的过程中,通常对应着装箱和拆箱的操作。

装箱:将基本数据类型变为包装类对象,利用每一个包装类提供的构造方法实现装箱处理。

拆箱:将包装类中包装的基本数据类型取出。利用Number 类中提供的六种方法。

如:

```
Integer num = new Integer(10) ; // 装箱
int data = num.intValue() ; // 拆箱
System.out.println(data);
```

9 java 中的其他操作

9.1 包的定义及使用

包的本质实际上就属于一个文件夹。在项目开发中很难避免类名称重复的问题。如果所有的 java文件都

放在一个文件夹中,就有可能存在覆盖问题。

9.1.1 包的定义

在.java 文件首行使用package 包名称; 即可

范例：定义包

```
package www.linqxxy.wwh ;
public class Test {
public static void main(String[] args) {
System.out.println("Hello World") ;
}
}
```

```
javac -d . Test.java
java www.bit.java.Test
```

一旦程序出现包名称，那么*.class 必须存在相应目录下。在 JDK 编译的时候使用配置参数。

打包编译命令： javac -d . 类.java

1. -d：表示生成目录，根据 package 的定义生成。
2. "." :表示在当前所在目录生成目录。

按照此种方式编译完成之后发现，自动会在当前目录下生成相应的文件夹以及相应的*.class文件。一旦 程序类上出现了包名称，那么在执行的时候就需要带上包名称，即使用完整类名称"包.类"。 在以后进行项目开发之中，一定要定义包

9.1.2 包的导入

开发中使用包的定义之后,相当于把一个大的项目分别按照一定要求保存在了不同的包之中，但是这些程序类一定会发生互相调用的情况,这个时候就需要包的导入. 最好的方法是让 java自己去匹配编译的先后顺序，最常用的打包编译命令为:javac -d . ./*.java

注意：类使用class 和 public class 的区别：

1. public class: 文件名称必须与类名称保持一致，如果希望一个类被其他包访问，则必须定义 为 public class .

2. class: 文件名称可以与类名称不一致,在一个*.java 中可以定义多个 class，但是这个类不允 许被其他包所访问。

另外需要注意的是,以上导入的语句为"import 包.类"这样只会导入一个类，如果说现在导入一个包中

的多个类，可以直接采用通配符"*"来完成。

import www.linqxxy.java.util.* ;

这种"*"并不意味要将包中的所有类都进行导入，而是根据你的需求来导入。

9.1.3 系统常用包(了解)

系统常用包：

1. java.lang:系统常用基础类(String、Object),此包从 JDK1.1 后自动导入。

2. java.lang.reflflect:java 反射编程包;

3. java.net:进行网络编程开发包。

4. java.sql:进行数据库开发的支持包。

5. java.util:是 java 提供的工具程序包。(集合类等)(**巨重要**)

6. java.io:I/O 编程开发包。

7. java.awt(离不开 windows 平台)、java.swing:UI 开发包，主要进行界面开发包，目前已经不
用了。

9.1.4 访问控制权限

在之前所学习到的 private、public 就是访问控制权限。在 java 中提供有四种访问控制权 限： private<default<protected<public，这四种访问控制权限的定义如下：

No	范围	private	default	Protect	public
1	同一包中的同一类	允许	允许	允许	允许
2	同一包中的不同类	不允许	允许	允许	允许
3	不同包中的子类	不允许	不允许	允许	允许
3	不同包中的非子类	不允许	不允许	不允许	允许

9.1.5 jar 命令

jar 本质上也是一种压缩文件，里面保存的都是*.class文件。也就是说现在要实现某一个功能模块，可能
有几百个类，最终交付给用户使用时，为了方便管理，就会将这些文件形成压缩包提供给用户。
在 JDK 中提供实现 jar文件操作的命令，只需要输入一个 jar 即可。
对于此命令，有如下几个常用参数：

1. "c":创建新档案
2. "f":指定档案文件名
3. "v":在标准输出中生成详细输出

源文件编译而后变为 jar文件。

1. 打包进行程序编译: javac -d . Message.java
2. 将生成的程序类打包为 jar文件:jar -cvf Message.jar Message.class

打开后发现有一个 META-INF文件夹，里面包含版本号等信息。
此时的 Message.jar 就包含我们需要的程序类.

9.2 设计模式

单例模式就是一个类只允许产生一个实例化对象。在设计一个单例模式中通常需要三个要素

1. 构造方法私有

2. 有一个静态的私有的实例化对象

3. 有一个静态的获取实例化对象的方法。

懒汉式单例设计模式代码示例：

```
class Singleton{
private static Singleton instance ;
private Singleton() { // private 声明构造
}
public static Singleton getInstance() {
if (instance==null) { // 表示此时还没有实例化
instance = new Singleton() ;
}
return instance ;
}
public void print() {
System.out.println("Hello World");
}
}
public class SingletonTest {
public static void main(String[] args) {
Singleton singleton = null ; // 声明对象
  singleton = Singleton.getInstance() ;
singleton.print();
```

```
}
}
```

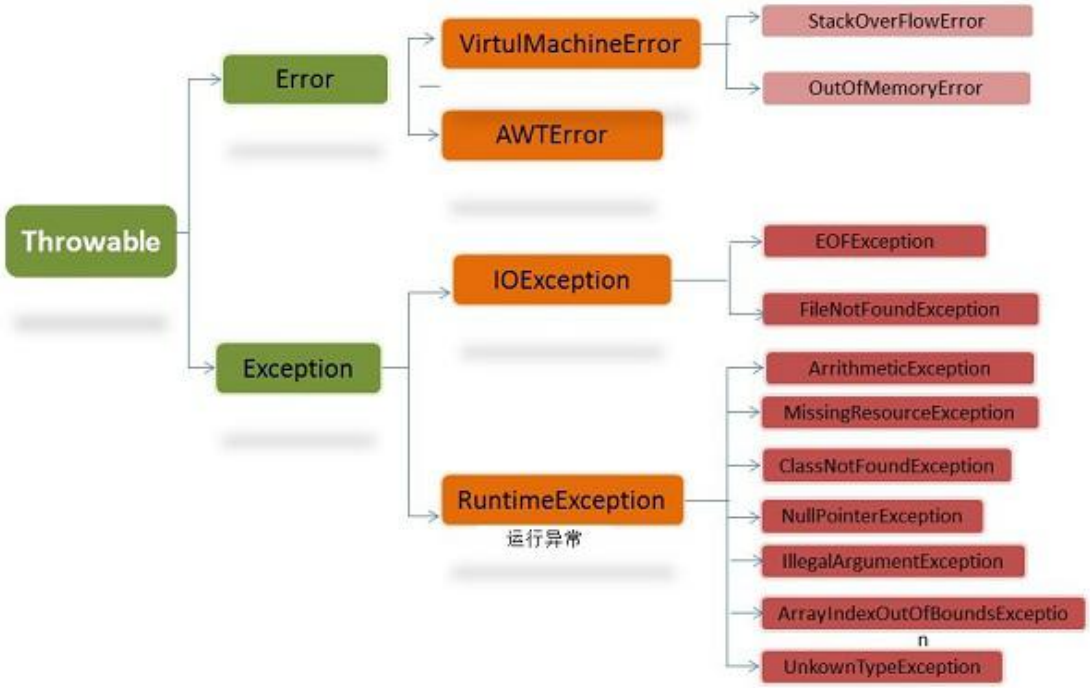
饿汉式单例设计模式代码如下：

```
class Singleton{
// 在类的内部可以访问私有结构，所以可以在类的内部产生实例化对象
private final static Singleton INSTANCE = new Singleton() ;
private Singleton() { // private 声明构造
}
public static Singleton getInstance() {
return INSTANCE ;
}
public void print() {
System.out.println("Hello World");
}
}
public class SingletonTest {
public static void main(String[] args) {
Singleton singleton = null ; // 声明对象
singleton = Singleton.getInstance() ;
singleton.print();
}
```

}

9.3 异常捕获

几乎所有的代码里面都会出现异常，为了保证程序在出现异常之后可以正常执行完毕，就需要进行异常处理。



在 Java 中，所有的异常都有一个共同的祖先 Throwable（可抛出）。Throwable 指定代码中可用异常传播机制通过 Java 应用程序传输的任何问题的共性。

Throwable 有两个重要的子类：Exception（异常）和 Error（错误），二者都是 Java 异常处理的重要子类，各自都包含大量子类。

Error（错误）

Error 是程序无法处理的错误 ,表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时 ,Java 虚拟机(JVM ）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误 ,本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 Error 的子类描述。

Exception（异常）

Exception 是程序本身可以处理的异常。

Exception 类有一个重要的子类 RuntimeException。RuntimeException 类及其子类表示 “JVM 常用操作” 引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

通常，Java 的异常(包括 Exception 和 Error)分为可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）。

可查异常（编译器要求必须处置的异常）:正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。

除了 RuntimeException 及其子类以外，其他的 Exception 类及其子类都属于可查异常。这种异常的特点是 Java 编译器会检查它，也就是说，当程序中可能出现这类异常，要么用 try-catch 语句捕获它，要么用 throws 子句声明抛出它，否则编译不会通过。

不可查异常(编译器不要求强制处置的异常)：包括运行时异常（RuntimeException 与其子类）和错误（Error）。

Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

运行时异常：都是 RuntimeException 类及其子类异常，如 NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是 Java 编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用 try-catch 语句捕获它，也没有用 throws 子句声明抛出它，也会编译通过。

非运行时异常 （编译异常）：是 RuntimeException 以外的异常，类型上都属于 Exception 类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如 IOException、SQLException 等以及用户自定义的 Exception 异常，一般情况下不自定义检查异常。

10 数据结构

10.1 时间复杂度与空间复杂度

10.1.1 算法效率

算法效率分析分为两种 :第一种是时间效率 ,第二种是空间效率。时间效率被称为时间复杂度，而空间效率 被称作空间复杂度。 时间复杂度主要衡量的是一个算法的运行速度 ,而空间复杂度主要衡量一个算法所需要 的额外空间，在计算机发展的早期，计算机的存储容量很小。所以对空间复杂度很是在乎。但是经过计算机 行业的迅速发展，计算机的存储容量已经达到了很高的程度。所以我们如今已经不需要再特别关注一个算法的空间复杂度。

10.1.2 时间复杂度

时间复杂度的定义：在计算机科学中，算法的时间复杂度是一个函数，它定量描述了该算法的运行时间。一 个算法执行所耗费的时间，从理论上说，是不能算出来的，只有你把你的程序放在机器上跑起来，才能知 道。但是我们需要每个算法都上机测试吗？是可以都上机测试，但是这很麻烦，所以才有了时间复杂度这个 分析方式。一个算法所花费的时间与其中语句的执行次数成正比例，算法中的基本操作的执行次数，为算法的时间复杂度。

我们一般都是采用大 O 渐进法来表示算法的时间复杂度的。

1、用常数 1 取代运行时间中的所有加法常数。

2、在修改后的运行次数函数中，只保留最高阶项。

3、如果最高阶项存在且不是 1，则去除与这个项目相乘的常数。得到的结果就是大 O 阶。

10.1.3 空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度 。空间复杂度不是程序占用了多少 bytes 的空间，因为这个也没太大意义，所以空间复杂度算的是变量的个数。空间复杂度计算规则基本跟时间复杂度类似，也使用大 O 渐进表示法。

10.2 线性表

10.2.1 顺序表

顺序表是用一段物理地址连续的存储单元依次存储数据元素的线性结构，一般情况下采用数组存储。在数组上完成数据的增删查改。

顺序表一般可以分为：

1. 静态顺序表：使用定长数组存储。
2. 动态顺序表：使用动态开辟的数组存储。

代码实现如下：

```
public interface IMySquence {//定义顺序表接口
    boolean add(int pos,Object obj);//在指定位置插入元素
    Object remove(Object key);//删除指定元素
    int search(Object key);//查找元素，找到了返回下表
    boolean contain(Object key);//查找元素是否存在
    Object getPos(int pos);//获取指定下表的元素
    int size();//返回顺序表的大小
    void display();//打印顺序表
    void clear();//清空顺序表
}
public class MySquence implements IMySquence {//实现顺序表接口
    private Object[]elem;
```

```
        private int usedSize;//
        private static final int DEFAULT_SIZE = 10;//默认顺序表
大小
        public MySquence(){
            this.elem=new Object[DEFAULT_SIZE];
            this.usedSize=0;
        }
        public boolean isFull(){//判断顺序表是否满了
            return (usedSize==elem.length-1);
        }
        public boolean isEmpty(){//判断顺序表是否为空
            return (usedSize==0);
        }
        @Override
        public boolean add(int pos, Object obj) {
            if(pos<0||pos>usedSize){//判断指定的位置是否合
法
                return false;
            }
            if(isFull()){//判断顺序表是否为满，满了扩容二倍

this.elem=Arrays.copyOf(elem,elem.length*2);
```

```
            }
            for(int i=usedSize;i>=pos;i--){
                elem[i+1]=elem[i];//将指定位置以及后面的元
素后移
            }
            elem[pos]=obj;
            usedSize++;
            return true;
        }

        @Override
        public Object remove(Object key) {
            if(isEmpty()){//判断顺序表是否为空，为空抛出异常
                throw new
UnsupportedOperationException("顺序表为空");
            }
            Object object=key;//将要删除的数据保留下来
            for(int i=0;i<usedSize;i++){
                if(key.equals(elem[i])){
                    for(int j=i;j<=usedSize;j++){
                        elem[j]=elem[j+1];//后面的元素前移，
覆盖需要删除的元素
```

```
                }
                elem[usedSize]=null;//将最后面的元素置
空
            }
        }
        usedSize--;//更新顺序表大小
        return object;
    }

    @Override
    public int search(Object key) {
        if(isEmpty()){//顺序表为空，则说明找不到
            return -1;
        }
        for(int i=0;i<usedSize;i++){
            if(key.equals(elem[i])){//找到了返回下标
                return i;
            }
        }
        return -1;
    }
```

```
    @Override
    public boolean contain(Object key) {
        if(isEmpty()){//顺序表为空，则说明不包含元素。
            return false;
        }
        for(int i=0;i<usedSize;i++){
            if(key.equals(elem[i])){//如果包含，返回 true
                return true;
            }
        }
        return false;
    }

    @Override
    public Object getPos(int pos) {
        if(isEmpty()||pos<0||pos>usedSize){//指定位置的
合法性
            return null;
        }
        return this.elem[pos];
    }
```

```
    @Override
    public int size() {
        return usedSize;
    }

    @Override
    public void display() {
        for(int i=0;i<usedSize;i++){
            System.out.print(this.elem[i]);
        }
        System.out.println();
    }

    @Override
    public void clear() {
        for (int i=0;i<usedSize;i++){
            this.elem[i]=null;
        }
    usedSize=0;
    }
}
```

我们发现，顺序表存在着如下的问题：

1. 中间/头部的插入删除，时间复杂度为 O(N)

2. 增容需要申请新空间，拷贝数据，释放旧空间。会有不小的消耗。

3. 增容一般是呈 2 倍的增长，势必会有一定的空间浪费。例如当前容量为 100，满了以后增容到 200， 我们再继续插入了 5 个数据，后面没有数据插入了，那么就浪费了 95 个数据空间。

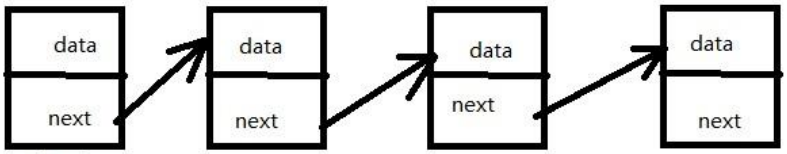
所以我们通常采用一种更好的数据结构。

10.2.2 链表

链表是另外一种线性表是一种物理存储结构上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的引用链接次序实现的。

链表（Linked List）是一种常见的数据结构，也是一种线性表，它相比于其他数据结构具有插入快，删除快的优点，同时链表可以灵活的分配存储空间。但链表并不能说是完美的数据结构，它的结构由于每个节点中会存放下一个节点中的位置，所以会有额外的内存开销，同时链表的结构也表明其查找的效率会比较低。

这里我们采用无头单链表来作为示例 ：



head

Node{

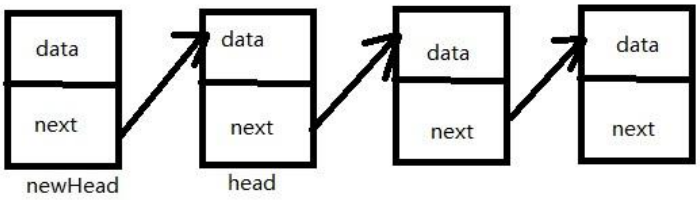
Size=4 ;

Node.data ;

Node.next ;

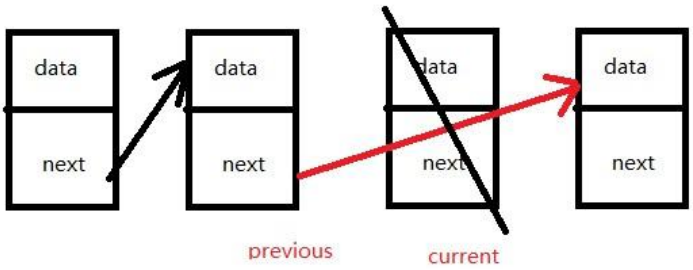
}

这就是一个简单的单链表，Node 代表节点，data 表示节点中存储的数据，而 next 代表下一个节点的位置。



新增加一个节点　　newHead.next=head;
size++;

这个就是节点的头插法。



删除current节点。
previous.next=current.next;
size--

然后是节点的删除，只需要将被删除的前一个节点的 next 指向被删除节点的 next 就可以了。下面我们来看代码示例

public interface IMySingleList {

```
public interface ILinked {
    //头插法
    void addFirst(int data);
    //尾插法
    void addLast(int data);
    //任意位置插入,第一个数据节点为 0 号下标
    boolean addIndex(int index,int data);
    //查找是否包含关键字 key 是否在单链表当中
    boolean contains(int key);
    //删除第一次出现关键字为 key 的节点
    int remove(int key);
    //删除所有值为 key 的节点
    void removeAllKey(int key);
    //得到单链表的长度
    int getLength();
    void display();
    void clear();
}
}
```

```
public class MySingleList implements
IMySingleList.ILinked{
    private Node head;
    private int size;
    public MySingleList(){
        this.head=null;
        this.size=0;
    }
    private class Node{
        int data;
        Node next;
        public Node(int data){
            this.data=data;
            this.next=null;
        }
    }
    public boolean isEmpty(){
        return this.head==null;
    }
    @Override
    public void addFirst(int data) {
        Node newHead=new Node(data);
```

```
        if(isEmpty()){
            head=newHead;
        }else{
            newHead.next=head;
            head=newHead;
        }
        size++;
    }

    @Override
    public void addLast(int data) {
        Node node=new Node(data);
        Node cur=head;
        while (cur.next!=null){
            cur=cur.next;
        }
        cur.next=node;
        size++;
    }

    @Override
    public boolean addIndex(int index, int data) {
```

```
        if(index<0||index>size) {
            return false;
        }else {
            Node node=new Node(data);
            Node dumyHead=new Node(-1);
            dumyHead.next=head;
            Node cur=head;
            Node prev=dumyHead;
            while (index!=0){
                cur=cur.next;
                prev=prev.next;
                index--;
            }
            prev.next=node;
            node.next=cur;
        }
        this.size++;
        return true;
    }

    @Override
    public boolean contains(int key) {
```

```
        if (isEmpty()){
            return false;
        }
        Node cur=head;
        while (cur!=null){
            if(cur.data==key){
                return true;
            }
            cur=cur.next;
        }
        return false;
    }

    @Override
    public int remove(int key) {
        if (isEmpty()){
            throw new
UnsupportedOperationException("链表为空");
        }
        Node cur=head;
        Node prev=new Node(-1);
        prev.next=head;
```

```
        while (cur.data!=key){
            if(cur.next==null){
                return 0;
            }else {
                prev=prev.next;
                cur=cur.next;
            }
        }
        if(cur==head){
            head=head.next;
            this.size--;
        }else {
            prev.next=cur.next;
            this.size--;
        }
        return 0;
    }

    @Override
    public void removeAllKey(int key) {
        if (isEmpty()){
```

```
                throw new
UnsupportedOperationException("链表为空");
            }
            Node cur=head;
            Node prev=new Node(-1);
            prev.next=head;
            int count=0;
            while (cur!=null){
                if (cur.data==key){
                    remove(key);
                    cur=cur.next;
                }else {
                    cur=cur.next;
                    prev=prev.next;
                }
            }
        }

        @Override
        public int getLength() {
            return this.size;
        }
```

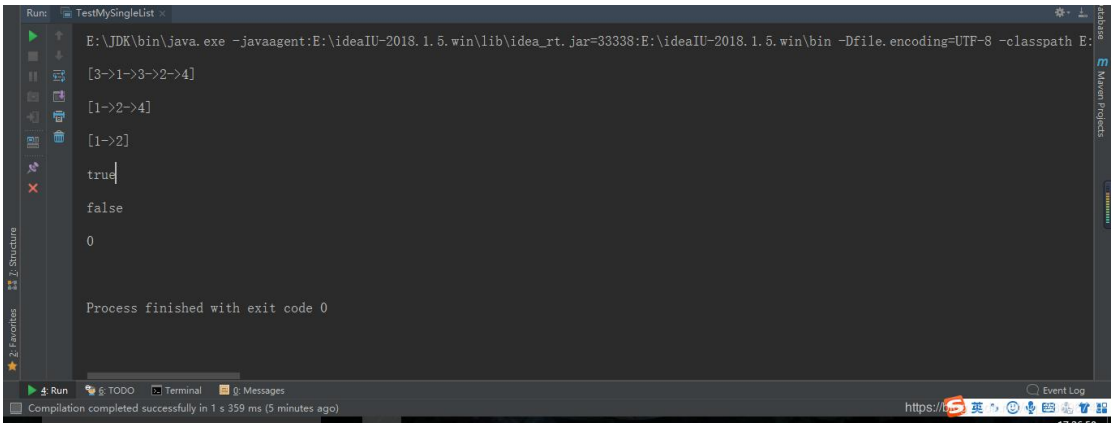
```
    @Override
    public void display() {
        Node cur = head;
        if (isEmpty()) {
            System.out.println("[]");
        } else if (cur.next == null) {
            System.out.println("[" + cur.data + "]");
        } else {
            for (cur = head; cur != null; cur = cur.next) {
                if (cur == head) {
                    System.out.print("[" + cur.data +
"->");
                }else if(cur.next==null){
                    System.out.print(cur.data+"]");
                }else {
                    System.out.print(cur.data+"->");
                }
            }
            System.out.println();
        }
    }
```

```
    @Override
    public void clear() {
        while (head!=null){
            head=head.next;
            this.size--;
        }
    }
}

public class TestMySingleList {
    public static void main(String[] args) {
        MySingleList mySingleList=new MySingleList();
        mySingleList.addFirst(1);
        mySingleList.addFirst(3);
        mySingleList.addLast(2);
        mySingleList.addLast(4);
        mySingleList.addIndex(2,3);
        mySingleList.display();
        mySingleList.removeAllKey(3);
        mySingleList.display();
        mySingleList.remove(4);
        mySingleList.display();
```

```
            System.out.println(mySingleList.contains(1));
            System.out.println(mySingleList.contains(6));
            mySingleList.clear();
            System.out.println(mySingleList.getLength());
        }
    }
```

运行结果：



10.3 栈和队列

10.3.1 栈

栈：一种特殊的线性表，其只允许在固定的一端进行插入和删除元素操作。进行数据插入和删除操作的一端

称为栈顶，另一端称为栈底。栈中的数据元素遵守后进先出 LIFO（Last In First Out）的原则。

压栈：栈的插入操作叫做进栈/压栈/入栈，入数据在栈顶。

出栈：栈的删除操作叫做出栈。出数据也在栈顶。

代码如下：

```
public class MyStack {
    private MyQueueImpl myQueue1;
    private MyQueueImpl myQueue2;
    private int usedSize;
    public MyStack(){
        this.myQueue1=new MyQueueImpl();
        this.myQueue2=new MyQueueImpl();
        this.usedSize=0;
    }
    /** Push element x onto stack. */
    public void push(int x) {
        if (myQueue1.empty()&&myQueue1.empty()){
            myQueue1.add(x);
        }else if(!myQueue1.empty()){
            myQueue1.add(x);
        }else {
            myQueue2.add(x);
        }
        this.usedSize++;
    }
```

```
    /** Removes the element on top of the stack and
returns that element. */
    public int pop() {
        if(empty()){
            throw new
UnsupportedOperationException("栈空");
        }
        int oldData=0;
        if(!myQueue1.empty()){
            for(int i=0;i<myQueue1.size()-1;i++){
                myQueue2.add(myQueue1.poll());
            }
            oldData =myQueue1.poll();
        }else {
            for(int i=0;i<myQueue2.size()-1;i++){
                myQueue1.add(myQueue2.poll());
            }

            oldData=myQueue2.poll();
        }
        usedSize--;
        return oldData;
```

```
    }

    /** Get the top element. */
    public int top() {
        if(empty()){
            throw new
UnsupportedOperationException("栈空");
        }
        int data=0;
        if(!myQueue1.empty()){
            for(int i=0;i<usedSize-1;i++){
                data=myQueue1.peek();
                myQueue2.add(myQueue1.poll());
            }
            } else {
            for(int i=0;i<usedSize-1;i++){
                data = myQueue2.peek();
                myQueue1.add(myQueue2.poll());
            }
        }
        return data;
    }
```

```
    /** Returns whether the stack is empty. */
    public boolean empty() {
    return usedSize==0;
    }
}
```

10.3.2

队列：只允许在一端进行插入数据操作，在另一端进行删除数据操作的特殊线性表，队列具有先进先出

FIFO(First In First Out) 入队列：进行插入操作的一端称为队尾 出队列：进行删除操作的一端称为队头

代码如下：

```
package TestDemo2.bit.src;

interface IMyQueue {
    // 判断这个队列是否为空
    boolean empty();
    // 返回队首元素，但不出队列
    int peek();
    // 返回队首元素，并且出队列
    int poll();
    // 将 item 放入队列中
    void add(int item);
    // 返回元素个数
    int size();
}

public class MyQueueImpl implements  IMyQueue {
    private  Node front;
    private Node rear;
    private int usedSize;
    private class Node {
        private int data;
        private Node next;
```

```
        public    Node(int data){
            this.data=data;
            this.next=null;
        }
    }
    public MyQueueImpl(){
        this.front=null;
        this.rear=null;
        this.usedSize=0;
    }
    @Override
    public boolean empty() {
        return this.usedSize==0;
    }

    @Override
    public int peek() {
        if(empty()){
            throw new
UnsupportedOperationException("队列为空");
        }
        return this.front.data;
```

```
    }

    @Override
    public int poll() {
        if(empty()){
            throw new
UnsupportedOperationException("队列为空");
        }

        int data=front.data;
        if(front==rear){
            front=null;
            rear=null;
        }else {
            front = front.next;
        }
        usedSize--;
        return data;
    }

    @Override
    public void add(int item) {
```

```
        Node node=new Node(item);
        if(empty()){
            this.front=node;
            this.rear=node;
        }else{
            this.rear.next=node;
            this.rear=node;
        }
        this.usedSize++;
    }

    @Override
    public int size() {
        return this.usedSize;
    }
}
```

11 排序算法

11.1 简单基础排序

11.1.1 冒泡排序

排序思路：每一趟将最大的一个数据放在最后面，重复多次实现排序。

算法优点：思路简单容易实现，是一种稳定的排序算法 java 中内置排序算法就是用的冒泡排序。

缺点：时间复杂度为 O(N^2)，效率低。

代码实现：

```
import java.util.Arrays;

public class BubbleSort {
    public static int[] bubbleSort(int[] array){
        for(int i=0;i<array.length-1;i++){
            boolean flag=true;
            for(int j=0;j<array.length-1-i;j++){
                if(array[j]>array[j+1]){
                    int temp=array[j];
                    array[j]=array[j+1];
                    array[j+1]=temp;
```

```
                    flag=false;
                }
            }
            if (flag){
                break;
            }
            System.out.print("第"+i+"趟排序为:");
            disPlay(array);
        }
        return array;
    }
    public static void disPlay(int[] array){
        for(int i=0;i<array.length;i++){
            System.out.print(array[i]);
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] array={3,2,5,6,9,8,1,7,0,4};
        BubbleSort.bubbleSort(array);
        System.out.println(Arrays.toString(array));
```

```
            }
        }
```

11.1.2 选择排序

算法思路：每次将数组未排序区间第一个值选为最小值，往后面找，如果有比最小值小的，就将两个值交换。

算法优点：不需要每次交换，只需要找到最小值了再发生交换。

算法缺点：时间复杂度为 O(N^2),效率较低。

代码如下：

```
public class ChoiceSort {
    public static int[] choiceSort(int[]array){
        for(int i=0;i<array.length-1;i++){
            int min=i;
            for(int j=i+1;j<array.length;j++){
                if(array[j]<array[min]){
                    min=j;
                }
            }
            if (min!=i){
                int temp=array[i];
                array[i]=array[min];
                array[min]=temp;
```

```
                }
                System.out.println("第"+i+"次排序的结果是
"+Arrays.toString(array));
            }
            return array;
        }

        public static void main(String[] args) {
            int[]array={1,2,6,8,0,9,4,3,5,7};
            choiceSort(array);
            System.out.println("排序的结果为
"+Arrays.toString(array));
        }
    }
```

11.1.3 插入排序

算法思路：分为已排序区间和未排序区间，每次从为排序区间选择一个值插入到已排序区间的指定位置。

算法优点：比较快而且稳定

算法缺点：比较次数不一定，比较次数越少，插入点后的数据移动越多。

代码如下：

```
public class InsertSort {
    public static int[] insertSort(int[] array){
        int j;
        for (int i=1;i<array.length;i++){
          int temp=array[i];
          j=i;
          while (j>0&&temp<array[j-1]){
              array[j]=array[j-1];
              j--;
          }
          array[j]=temp;
        }
        return array;
    }

    public static void main(String[] args) {
        int[]array={3,5,7,9,0,1,2,4,6,8};
        insertSort(array);
        System.out.println("排序的结果为"+
Arrays.toString(array));
    }
}
```

11.2 进阶排序算法

11.2.1 希尔排序

算法思路：对插入排序的一种优化。每次将数据分组进行插入排序，直到排序结束。

算法优点：快，数据移动少。N(LogN)

算法缺点：不稳定

代码如下：

```
public class ShellSort {
    public static void sort(int[]arr){
        int n=arr.length;
        if(n<=1){
            return;
        }else {
            int step = n / 2;
            while (step >= 1) {
                for (int i = step; i < n; i++) {
                    int tmp = arr[i];
                    int j = i - step;
                    for (; j >= 0; j -= step) {
                        if (arr[j] > tmp) {
                            arr[j + step] = arr[j];
```

```
                    } else {
                        break;
                    }
                }
                arr[j + step] = tmp;
            }
            step = step / 2;
        }
    }
}
```

11.2.2 堆排序

算法思路：将所有的数据写为二叉树的形式，然后将二叉树整体调为大根堆或者小根堆的形式，在将二叉树分为已排序和未排序区间，每次将未排序区间的值和根的值进行交换，再次进行根堆调整。直到所有的数据全部有序。

算法优点：快，在最坏的情况下比快速排序还要快。N(LogN)

算法缺点：不稳定。而且不适合太小的待排序序列。

代码如下：

```
public class HeapSort {
```

```
private static void adjust(int[]arr,int start,int end){
        int tmp=arr[start];
        for(int i=2*start+1;i<=end;i=i*2+1){
            if((i<end&&arr[i]<arr[i+1])){
                i++;
            }
            if(arr[i]>tmp){
                arr[start]=arr[i];
                start=i;
            }else {
                break;
            }
        }
        arr[start]=tmp;
}
```

```
public static void sort(int[]arr){
    long start=System.currentTimeMillis();
    int n=arr.length;
    if(n<=1){
        return;
    }
    for(int i=(arr.length-1-1)/2;i>=0;i--){
        adjust(arr,i,n-1);
    }
    for(int i=0;i<n;i++){
        int tmp=arr[arr.length-1-i];
        arr[arr.length-1-i]=arr[0];
        arr[0]=tmp;
        adjust(arr,0,n-1-i-1);
    }
```

```
        long end=System.currentTimeMillis();
        System.out.println("堆排序耗时"+(end-start)+"毫秒");
    }
}
```

11.2.3 归并排序

算法思路：采用分治思想进行排序，先将所有数据从中间二等分，然后分别对左右区间继续进行归并排序。最后将左右合并成有序区间。、

算法优点：最坏的情况时间复杂度也是 n(logn)，很快。也很稳定

算法缺点：需要借助辅助空间。

代码如下：

```
public class MergeSort {
    public static void sort(int[]arr){
        int n=arr.length;
        if(n<=1){
            return;
        }
        mergeSort(arr,0,n-1);
    }
    private static void mergeSort(int[]arr,int start,int end){
        if(start==end){
```

```
            return;
        }
        int mid=(start+end)/2;
        mergeSort(arr,start,mid);
        mergeSort(arr,mid+1,end);
        merge(arr,start,mid,end);
    }
    private static void merge(int[]arr,int start,int mid,int end){
        int[]tmpArray=new int[arr.length];
        int tmpIndex=start;
        int i=start;
        int start2=mid+1;
        while (start<=mid&&start2<=end){
            if(arr[start]<=arr[start2]){
                tmpArray[tmpIndex++]=arr[start++];
            }else {
                tmpArray[tmpIndex++]=arr[start2++];
            }
        }
        while (start<=mid){
            tmpArray[tmpIndex++]=arr[start++];
        }
```

```
            while (start2<=end){
                tmpArray[tmpIndex++]=arr[start2++];
            }
            while (i<=end){
                arr[i]=tmpArray[i];
                i++;
            }
        }
}
```

11.2.4 快速排序

算法思路：首先选取一个基准点（通常选择最左边的值），从最右边标记开始找，直到找到比基准点小的值，如果遇见比基准点小的值，就将其放在最左边标记点。然后最左边标记点往右边找比基准点大的值 ,找到后将其放在最右边标记点。如果最左边和最右边标记点相遇，则将该点作为标记点分为左右两个待排序区间 ,继续进行快速排序操作。

算法优点：是最快的排序算法 n(logn)

算法缺点：不稳定

代码如下：

```
public class QuickSort {
    public static void sort(int[]arr){
```

```
        int n=arr.length;
        if(n<=1){
            return;
        }
        quickSort(arr,0,n-1);
    }
    private static void quickSort(int[]arr,int start,int end){
//          if(start>=end){
//              return;
//          }
        int par=partition(arr,start,end);
        if(par>start+1){
            quickSort(arr,start,par-1);
        }
        if(par<end-1){
            quickSort(arr,par+1,end);
        }
    }
    private static int partition(int[]arr,int low,int high){
        int tmp=arr[low];
        while (low<high){
            while ((low<high)&&arr[high]>=tmp){
```

```
                high--;
            }
            if(low>=high){
                break;
            }else {
                arr[low]=arr[high];
            }
            while ((low<high)&&arr[low]<=tmp){
                low++;
            }
            if(low>=high){
                break;
            }else {
                arr[high]=arr[low];
            }
        }
        arr[low]=tmp;
        return low;
    }
}
```