# 岗位资格认证考试

## 一、基础算法

### 1、排序算法

**空间复杂度只考虑额外的空间需求**

#### （1）冒泡排序

空间复杂度$O(1)$，时间复杂度$O(n^2)$

#### （2）插入排序

空间复杂度，时间复杂度

#### （3）快速排序

空间复杂度，时间复杂度

#### （4）归并排序

空间复杂度，时间复杂度

#### （5）堆排序

空间复杂度，时间复杂度

### 2、查找算法

#### 1、顺序查找

#### 2、二分查找

#### 3、插值查找

#### 4、斐波那契查找

#### 5、数表查找

#### 6、哈希查找

### 3、常见数据结构

#### （1）列表

#### （2）链表

#### （3）树

#### （4）队列

#### （5）堆栈

## 二、Java基础

### 1、Java数据类型

#### （1）变量

在Java中，变量分为两种：基本类型的变量和引用类型的变量。在Java中，变量必须先定义后使用，在定义变量的时候，可以给它一个初始值。变量的一个重要特点是可以重新赋值。变量不但可以重新赋值，还可以赋值给其他变量。

#### （2）基本数据类型

基本数据类型是CPU可以直接进行运算的类型。Java定义了以下几种基本数据类型：

- 整数类型：byte，short，int，long
- 浮点数类型：float，double
- 字符类型：char
- 布尔类型：boolean
- void类型：void

其中：

byte类型，大小8 bits（1个字节），取值范围-128~+127；

short类型，大小16 bits（2个字节），取值范围$-2^{15}$~$+2^{15}-1$；

int类型，大小32 bits（4个字节），取值范围$-2^{31}$~$+2^{31}-1$；

long类型，大小64 bits（8个字节），取值范围$-2^{63}$~$+2^{63}-1$；

float类型，大小32 bits（4个字节），取值范围IEEE754~IEEE754；

double类型，大小64 bits（8个字节），取值范围IEEE754~IEEE754；

char类型，大小16 bits（2个字节），取值范围Unicode $0$~$+2^{16}-1$；

### （3）引用数据类型

其余数据类型均为引用类型，如String类型。引用类型的变量类似于C语言的指针，它内部存储一个“地址”，指向某个对象在内存的位置。定义变量的时候，如果加上`final`修饰符，这个变量就变成了常量。有些时候，类型的名字太长，写起来比较麻烦，如果想省略变量类型，可以使用`var`关键字，编译器会根据赋值语句自动推断出变量的数据类型。

### （4）变量的作用范围

在Java中，多行语句用{ }括起来。很多控制语句，例如条件判断和循环，都以{ }作为它们自身的范围。只要正确地嵌套这些{ }，编译器就能识别出语句块的开始和结束。而在语句块中定义的变量，它有一个作用域，就是从定义处开始，到语句块结束。超出了作用域引用这些变量，编译器会报错。定义变量时，要遵循作用域最小化原则，尽量将变量定义在尽可能小的作用域，并且，不要重复使用变量名。

### （5）整数运算

Java的整数运算遵循四则运算规则，可以使用任意嵌套的小括号。整数的数值表示不但是精确的，而且整数运算永远是精确的，即使是除法也是精确的，因为两个整数相除只能得到结果的整数部分，求余运算使用`%`。特别注意：整数的除法对于除数为0时运行时将报错，但编译不会报错。

### （6）溢出运算

整数由于存在范围限制，如果计算结果超出了范围，就会产生溢出，而溢出不会出错，由于最高位计算结果为`1`，因此，加法结果变成了一个负数。整数运算在除数为`0`时会报错，而浮点数运算在除数为`0`时，不会报错，但会返回几个特殊值：

```
double d1 = 0.0 / 0; // NaN
double d2 = 1.0 / 0; // Infinity
double d3 = -1.0 / 0; // -Infinity
```

### （7）自增/自减

注意`++`写在前面和后面计算结果是不同的，`++n`表示先加1再引用n，`n++`表示先引用n再加1。不建议把`++`运算混入到常规运算中，容易自己把自己搞懵了。

### （8）移位运算

可以对整数进行移位运算：

```
int n = 7;       // 00000000 00000000 00000000 00000111 = 7
int a = n << 1;  // 00000000 00000000 00000000 00001110 = 14
int b = n << 2;  // 00000000 00000000 00000000 00011100 = 28
int c = n << 28; // 01110000 00000000 00000000 00000000 = 1879048192
int d = n << 29; // 11100000 00000000 00000000 00000000 = -536870912
```

### （9）位运算

位运算是按位进行与、或、非和异或的运算。

与运算：

```
n = 0 & 0; // 0
n = 0 & 1; // 0
n = 1 & 0; // 0
n = 1 & 1; // 1
```

或运算：

```
n = 0 | 0; // 0
n = 0 | 1; // 1
n = 1 | 0; // 1
n = 1 | 1; // 1
```

非运算：

```
n = ~0; // 1
n = ~1; // 0
```

异或运算：

```
n = 0 ^ 0; // 0
n = 0 ^ 1; // 1
n = 1 ^ 0; // 1
n = 1 ^ 1; // 0
```

## （10）类型自动提升与强制转型

在运算过程中，如果参与运算的两个数类型不一致，那么计算结果为较大类型的整型。例如，`short`和`int`计算，结果总是`int`，原因是`short`首先自动被转型为`int`。如果参与运算的两个数其中一个是整型一个是浮点型，那么整型可以自动提升到浮点型。

## （11）浮点数运算

浮点数运算和整数运算相比，只能进行加减乘除这些数值计算，不能做位运算和移位运算。

在计算机中，浮点数虽然表示的范围大，但是，浮点数有个非常重要的特点，就是浮点数常常无法精确表示。因为浮点数常常无法精确表示，因此，浮点数运算会产生误差。由于浮点数存在运算误差，所以比较两个浮点数是否相等常常会出现错误的结果。正确的比较方法是判断两个浮点数之差的绝对值是否小于一个很小的数：

```
// 比较x和y是否相等，先计算其差的绝对值：
double r = Math.abs(x - y);
// 再判断绝对值是否足够小：
if (r < 0.00001) {
    // 可以认为相等
} else {
    // 不相等
}
```

浮点数在内存的表示方法和整数比更加复杂。Java的浮点数完全遵循IEEE-754标准，这也是绝大多数计算机平台都支持的浮点数标准表示方法。可以将浮点数强制转型为整数。在转型时，浮点数的小数部分会被丢掉。如果转型后超过了整型能表示的最大范围，将返回整型的最大值。如果要进行四舍五入，可以对浮点数加上0.5再强制转型。

## （12）短路运算

布尔运算的一个重要特点是短路运算。如果一个布尔运算的表达式能提前确定结果，则后续的计算不再执行，直接返回结果。

```
public class Main {
    public static void main(String[] args) {
        boolean b = 5 < 3;
        boolean result = b && (5 / 0 > 0);
        System.out.println(result);
    }
}
```

如果没有短路运算，`&&`后面的表达式会由于除数为`0`而报错，但实际上该语句并未报错，原因在于与运算是短路运算符，提前计算出了结果`false`。如果变量`b`的值为`true`，则表达式变为`true && (5 / 0 > 0)`。因为无法进行短路运算，该表达式必定会由于除数为`0`而报错，可以自行测试。类似的，对于`||`运算，只要能确定第一个值为`true`，后续计算也不再进行，而是直接返回`true`。

## （13）字符和字符串

字符类型`char`是基本数据类型，它是`character`的缩写。一个`char`保存一个Unicode字符。因为Java在内存中总是使用Unicode表示字符，所以，一个英文字符和一个中文字符都用一个`char`类型表示，它们都占用两个字节。要显示一个字符的Unicode编码，只需将`char`类型直接赋值给`int`类型即可。

```
int n1 = 'A'; // 字母“A”的Unicodde编码是65
int n2 = '中'; // 汉字“中”的Unicode编码是20013
```

还可以直接用转义字符`\u`+Unicode编码来表示一个字符:

```
// 注意是十六进制:
char c3 = '\u0041'; // 'A', 因为十六进制0041 = 十进制65
char c4 = '\u4e2d'; // '中', 因为十六进制4e2d = 十进制20013
```

和`char`类型不同，字符串类型`String`是引用类型，我们用双引号`"..."`表示字符串。一个字符串可以存储0个到任意个字符。Java的编译器对字符串做了特殊照顾，可以使用`+`连接任意字符串和其他数据类型，这样极大地方便了字符串的处理。

```
public class Main {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "world";
        String s = s1 + " " + s2 + "!";
        System.out.println(s);
    }
}

```

从Java 13开始，字符串可以用`"""..."""`表示多行字符串（Text Blocks）了。

```
public class Main {
    public static void main(String[] args) {
        String s = """
                   SELECT * FROM
                     users
                   WHERE id > 100
                   ORDER BY name DESC
                   """;
        System.out.println(s);
    }
}

```

Java的字符串除了是一个引用类型外，还有个重要特点，就是字符串不可变。

```
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        System.out.println(s); // 显示 hello
        s = "world";
        System.out.println(s); // 显示 world
    }
}
```

变的不是字符串，而是变量`s`的“指向”。原来的字符串`"hello"`还在，只是我们无法通过变量`s`访问它而已。因此，字符串的不可变是指字符串内容不可变。

## （14）数组类型

数组的初始化：

```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[5];
        ns[0] = 68;
        ns[1] = 79;
        ns[2] = 91;
        ns[3] = 85;
        ns[4] = 62;
    }
}

```

定义一个数组类型的变量，使用数组类型“类型[]”，例如，`int[]`。和单个基本类型变量不同，数组变量初始化必须使用`new int[5]`表示创建一个可容纳5个`int`元素的数组。

Java的数组有几个特点：

- 数组所有元素初始化为默认值，整型都是`0`，浮点型是`0.0`，布尔型是`false`；
- 数组一旦创建后，大小就不可改变。

要访问数组中的某一个元素，需要使用索引。数组索引从`0`开始，例如，5个元素的数组，索引范围是`0`~`4`。

可以修改数组中的某一个元素，使用赋值语句，例如，`ns[1] = 79;`。

可以用`数组变量.length`获取数组大小。

```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[] { 68, 79, 91, 85, 62 };
        System.out.println(ns.length); // 编译器自动推算数组大小为5
    }
}
```

如果数组元素不是基本类型，而是一个引用类型。数组中只存放指向被引用对象的指针。

# 2、流程控制

## （1）if判断

在Java中，判断值类型的变量是否相等，可以使用`==`运算符。但是，判断引用类型的变量是否相等，`==`表示“引用是否相等”，或者说，是否指向同一个对象。例如，下面的两个String类型，它们的内容是相同的，但是，分别指向不同的对象，用`==`判断，结果为`false`。要判断引用类型的变量内容是否相等，必须使用`equals()`方法。

## （2）break和continue

`break`语句通常都是配合`if`语句使用。要特别注意，`break`语句总是跳出自己所在的那一层循环。

# 3、数组操作

## （1）遍历数组

方法1:

```
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int i=0; i<ns.length; i++) {
            int n = ns[i];
            System.out.println(n);
        }
    }
}
```

方法2:

```
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int n : ns) {
            System.out.println(n);
        }
    }
}
```

### （2）二维数组

```
public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        int[] arr0 = ns[0];
        System.out.println(arr0.length); // 4
    }
}
```

### （3）命令行参数

Java程序的入口是`main`方法，而`main`方法可以接受一个命令行参数，它是一个`String[]`数组。这个命令行参数由JVM接收用户输入并传给`main`方法。

```
public class Main {
    public static void main(String[] args) {
        for (String arg : args) {
            if ("-version".equals(arg)) {
                System.out.println("v 1.0");
                break;
            }
        }
    }
}
```

上面这个程序必须在命令行执行，先进行编译：

```
$ javac Main.java
```

然后，执行的时候，给它传递一个`-version`参数：

```
$ java Main -version
v 1.0
```

命令行参数类型是`String[]`数组；

命令行参数由JVM接收用户输入并传给`main`方法；

如何解析命令行参数需要由程序自己实现。

## 4、面向对象

Java是一种面向对象的编程语言。面向对象编程，英文是Object-Oriented Programming，简称OOP。和面向对象编程不同的，是面向过程编程。面向过程编程，是把模型分解成一步一步的过程。class是一种对象模版，它定义了如何创建实例，因此，class本身就是一种数据类型。而instance是对象实例，instance是根据class创建的实例，可以创建多个instance，每个instance类型相同，但各自属性可能不相同。

面向对象的基本概念，包括：

- 类
- 实例
- 方法

面向对象的实现方式，包括：

- 继承
- 多态

###

## （1）定义一个对象

```
class Person {
    public String name;
    public int age;
}
```

直接把`field`用`public`暴露给外部可能会破坏封装性。为了避免外部代码直接去访问`field`，我们可以用`private`修饰`field`，拒绝外部访问：

```
class Person {
    private String name;
    private int age;
}
```

把`field`从`public`改成`private`，外部代码不能访问这些`field`，需要使用方法（`method`）来让外部代码可以间接修改`field`：

```
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.setName("Xiao Ming"); // 设置name
        ming.setAge(12); // 设置age
        System.out.println(ming.getName() + ", " + ming.getAge());
    }
}

class Person {
    private String name;
    private int age;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        if (age < 0 || age > 100) {
```

```
            throw new IllegalArgumentException("invalid age value");
        }
        this.age = age;
    }
}
```

外部代码不能直接修改`private`字段，但是，外部代码可以调用方法`setName()`和`setAge()`来间接修改`private`字段。在方法内部，我们就有机会检查参数对不对。比如，`setAge()`就会检查传入的参数，参数超出了范围，直接报错。这样，外部代码就没有任何机会把`age`设置成不合理的值。对`setName()`方法同样可以做检查，例如，不允许传入`null`和空字符串：

```
public void setName(String name) {
    if (name == null || name.isBlank()) {
        throw new IllegalArgumentException("invalid name");
    }
    this.name = name.strip(); // 去掉首尾空格
}
```

有`public`方法，自然就有`private`方法。和`private`字段一样，`private`方法不允许外部调用，定义`private`方法的理由是内部方法是可以调用`private`方法的。

### （2）this变量

在方法内部，可以使用一个隐含的变量`this`，它始终指向当前实例。因此，通过`this.field`就可以访问当前实例的字段。如果没有命名冲突，可以省略`this`。

```
class Person {
    private String name;

    public String getName() {
        return name; // 相当于this.name
    }
}
```

如果有局部变量和字段重名，那么局部变量优先级更高，就必须加上`this`：

```
class Person {
    private String name;

    public void setName(String name) {
        this.name = name; // 前面的this不可少，少了就变成局部变量name了
    }
}
```

方法可以包含0个或任意个参数。方法参数用于接收传递给方法的变量值。调用方法时，必须严格按照参数的定义一一传递。

```
class Person {
    ...
    public void setNameAndAge(String name, int age) {
        ...
    }
}
```

可变参数用`类型...`定义，可变参数相当于数组类型：

```
class Group {
    private String[] names;

    public void setNames(String... names) {
        this.names = names;
    }
}
```

上面的`setNames()`就定义了一个可变参数。调用时，可以这么写：

```
Group g = new Group();
g.setNames("Xiao Ming", "Xiao Hong", "Xiao Jun"); // 传入3个String
g.setNames("Xiao Ming", "Xiao Hong"); // 传入2个String
g.setNames("Xiao Ming"); // 传入1个String
g.setNames(); // 传入0个String
```

完全可以把可变参数改写为`String[]`类型，但是，调用方需要自己先构造`String[]`，比较麻烦。另一个问题是，调用方可以传入`null`，而可变参数可以保证无法传入`null`，因为传入0个参数时，接收到的实际值是一个空数组而不是`null`。

### （3）参数绑定

基本类型参数的传递，是调用方值的复制。双方各自的后续修改，互不影响。

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        int n = 15; // n的值为15
        p.setAge(n); // 传入n的值
        System.out.println(p.getAge()); // 15
        n = 20; // n的值改为20
        System.out.println(p.getAge()); // 15
    }
}

class Person {
    private int age;

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

引用类型参数的传递，调用方的变量，和接收方的参数变量，指向的是同一个对象。双方任意一方对这个对象的修改，都会影响对方。

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

```
        Person p = new Person();
        String[] fullname = new String[] { "Homer", "Simpson" };
        p.setName(fullname); // 传入fullname数组
        System.out.println(p.getName()); // "Homer Simpson"
        fullname[0] = "Bart"; // fullname数组的第一个元素修改为"Bart"
        System.out.println(p.getName()); // "Bart Simpson"
    }
}

class Person {
    private String[] name;

    public String getName() {
        return this.name[0] + " " + this.name[1];
    }

    public void setName(String[] name) {
        this.name = name;
    }
}
```

但String类为不可变类，会用一个新的地址引用。当运行 bob = "Alice"时，会在内存中新开辟一个空间，bob的指向该变了。但是name属性还是指向原来的内存空间 还是Bob。

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String bob = "Bob";
        p.setName(bob); // 传入bob变量
        System.out.println(p.getName()); // "Bob"
        bob = "Alice"; // bob改名为Alice
        System.out.println(p.getName()); // "Bob"
    }
}

class Person {
    private String name;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

### （4）构造方法

创建实例的时候，实际上是通过构造方法来初始化实例的。我们先来定义一个构造方法，能在创建`Person`实例的时候，一次性传入`name`和`age`，完成初始化。由于构造方法是如此特殊，所以构造方法的名称就是类名。构造方法的参数没有限制，在方法内部，也可以编写任意语句。但是，和普通方法相比，构造方法没有返回值（也没有`void`），调用构造方法，必须用`new`操作符。

```
public class Main {
```

```
    public static void main(String[] args) {
        Person p = new Person("Xiao Ming", 15);
        System.out.println(p.getName());
        System.out.println(p.getAge());
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}

```

如果一个类没有定义构造方法，编译器会自动为我们生成一个默认构造方法，它没有参数，也没有执行语句，类似这样：

```
class Person {
    public Person() {
    }
}
```

如果我们自定义了一个构造方法，那么，编译器就*不再*自动创建默认构造方法。如果既要能使用带参数的构造方法，又想保留不带参数的构造方法，那么只能把两个构造方法都定义出来：

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Xiao Ming", 15); // 既可以调用带参数的构造方法
        Person p2 = new Person(); // 也可以调用无参数构造方法
    }
}

class Person {
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

```

```
    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}

```

没有在构造方法中初始化字段时，引用类型的字段默认是`null`，数值类型的字段用默认值，`int`类型默认值是`0`，布尔类型默认值是`false`。在Java中，创建对象实例的时候，按照如下顺序进行初始化：

1. 先初始化字段，例如，`int age = 10;`表示字段初始化为`10`，`double salary;`表示字段默认初始化为`0`，`String name;`表示引用类型字段默认初始化为`null`；
2. 执行构造方法的代码进行初始化。

可以定义多个构造方法，在通过`new`操作符调用的时候，编译器通过构造方法的参数数量、位置和类型自动区分：

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this.name = name;
        this.age = 12;
    }

    public Person() {
    }
}
```

一个构造方法可以调用其他构造方法，这样做的目的是便于代码复用。调用其他构造方法的语法是`this(…)`：

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this(name, 18); // 调用另一个构造方法Person(String, int)
    }

    public Person() {
        this("Unnamed"); // 调用另一个构造方法Person(String)
```

```
    }
}
```

在一个类中，我们可以定义多个方法。如果有一系列方法，它们的功能都是类似的，只有参数有所不同，那么，可以把这一组方法名做成同名方法。

```
class Hello {
    public void hello() {
        System.out.println("Hello, world!");
    }

    public void hello(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public void hello(String name, int age) {
        if (age < 18) {
            System.out.println("Hi, " + name + "!");
        } else {
            System.out.println("Hello, " + name + "!");
        }
    }
}
```

这种方法名相同，但各自的参数不同，称为方法重载（`Overload`）。

注意：方法重载的返回值类型通常都是相同的。

方法重载的目的是，功能类似的方法使用同一名字，更容易记住，因此，调用起来更简单。

### （5）继承

继承是面向对象编程中非常强大的一种机制，它首先可以复用代码。当我们让`Student`从`Person`继承时，`Student`就获得了`Person`的所有功能，我们只需要为`Student`编写新增的功能。Java使用`extends`关键字来实现继承：

```
class Person {
    private String name;
    private int age;

    public String getName() {...}
    public void setName(String name) {...}
    public int getAge() {...}
    public void setAge(int age) {...}
}

class Student extends Person {
    // 不要重复name和age字段/方法,
    // 只需要定义新增score字段/方法:
    private int score;

    public int getScore() { … }
    public void setScore(int score) { … }
}
```

在OOP的术语中，我们把`Person`称为超类（super class），父类（parent class），基类（base class），把`Student`称为子类（subclass），扩展类（extended class）。注意到我们在定义`Person`的时候，没有写`extends`。在Java中，没有明确写`extends`的类，编译器会自动加上`extends Object`。所以，任何类，除了`Object`，都会继承自某个类。Java只允许一个class继承自一个类，因此，一个类有且仅有一个父类。只有`Object`特殊，它没有父类。

## （6）protected

继承有个特点，就是子类无法访问父类的`private`字段或者`private`方法。例如，`Student`类就无法访问`Person`类的`name`和`age`字段：

```
class Person {
    private String name;
    private int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // 编译错误：无法访问name字段
    }
}
```

这使得继承的作用被削弱了。为了让子类可以访问父类的字段，我们需要把`private`改为`protected`。用`protected`修饰的字段可以被子类访问：

```
class Person {
    protected String name;
    protected int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // OK!
    }
}
```

因此，`protected`关键字可以把字段和方法的访问权限控制在继承树内部，一个`protected`字段和方法可以被其子类，以及子类的子类所访问。

## （7）super

`super`关键字表示父类（超类）。子类引用父类的字段时，可以用`super.fieldName`。

```
class Student extends Person {
    public String hello() {
        return "Hello, " + super.name;
    }
}
```

实际上，这里使用`super.name`，或者`this.name`，或者`name`，效果都是一样的。编译器会自动定位到父类的`name`字段。但是，在某些时候，就必须使用`super`。

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

```
        Student s = new Student("Xiao Ming", 12, 89);
    }
}

class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        this.score = score;
    }
}

```

运行上面的代码，会得到一个编译错误，大意是在`Student`的构造方法中，无法调用`Person`的构造方法。

这是因为在Java中，任何`class`的构造方法，第一行语句必须是调用父类的构造方法。如果没有明确地调用父类的构造方法，编译器会帮我们自动加一句`super();`，所以，`Student`类的构造方法实际上是这样：

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(); // 自动调用父类的构造方法
        this.score = score;
    }
}
```

但是，`Person`类并没有无参数的构造方法，因此，编译失败。解决方法是调用`Person`类存在的某个构造方法。

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(name, age); // 调用父类的构造方法Person(String, int)
        this.score = score;
    }
}
```

如果父类没有默认的构造方法，子类就必须显式调用`super()`并给出参数以便让编译器定位到父类的一个合适的构造方法。这里还顺带引出了另一个问题：即子类*不会继承*任何父类的构造方法。子类默认的构造方法是编译器自动生成的，不是继承的。

## （8）阻止继承

正常情况下，只要某个class没有`final`修饰符，那么任何类都可以从该class继承。从Java 15开始，允许使用`sealed`修饰class，并通过`permits`明确写出能够从该class继承的子类名称。

```
public sealed class Shape permits Rect, Circle, Triangle {
    ...
}
```

如果写：

```
public final class Rect extends Shape {...}
```

是没有问题的。因为`Rect`出现在`Shape`的`permits`列表中。但是，如果定义一个`Ellipse`就会报错：

```
public final class Ellipse extends Shape {...}
// Compile error: class is not allowed to extend sealed class: Shape
```

原因是`Ellipse`并未出现在`Shape`的`permits`列表中。这种`sealed`类主要用于一些框架，防止继承被滥用。

## （9）向上转型

把一个子类类型安全地变为父类类型的赋值，被称为向上转型（upcasting）。

```
Student s = new Student();
Person p = s; // upcasting, ok
Object o1 = p; // upcasting, ok
```

## （10）向下转型

如果把一个父类类型强制转型为子类类型，就是向下转型（downcasting）。

```
Person p1 = new Student(); // upcasting, ok
Person p2 = new Person();
Student s1 = (Student) p1; // ok
Student s2 = (Student) p2; // runtime error! ClassCastException!
```

`Person`类型`p1`实际指向`Student`实例，`Person`类型变量`p2`实际指向`Person`实例。在向下转型的时候，把`p1`转型为`Student`会成功，因为`p1`确实指向`Student`实例，把`p2`转型为`Student`会失败，因为`p2`的实际类型是`Person`，不能把父类变为子类，因为子类功能比父类多，多的功能无法凭空变出来。因此，向下转型很可能会失败。失败的时候，Java虚拟机会报`ClassCastException`。为了避免向下转型出错，Java提供了`instanceof`操作符，可以先判断一个实例究竟是不是某种类型：

```
Person p = new Person();
System.out.println(p instanceof Person); // true
System.out.println(p instanceof Student); // false

Student s = new Student();
System.out.println(s instanceof Person); // true
System.out.println(s instanceof Student); // true

Student n = null;
System.out.println(n instanceof Student); // false
```

`instanceof`实际上判断一个变量所指向的实例是否是指定类型，或者这个类型的子类。如果一个引用变量为`null`，那么对任何`instanceof`的判断都为`false`。

```
Person p = new Student();
if (p instanceof Student) {
    // 只有判断成功才会向下转型:
    Student s = (Student) p; // 一定会成功
}
```

从Java 14开始，判断`instanceof`后，可以直接转型为指定变量，避免再次强制转型。

代码:

```
Object obj = "hello";
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.toUpperCase());
}
```

等价于:

```
public class Main {
    public static void main(String[] args) {
        Object obj = "hello";
        if (obj instanceof String s) {
            // 可以直接使用变量s:
            System.out.println(s.toUpperCase());
        }
    }
}
```

## （11）区分继承与组合

在使用继承时，我们要注意逻辑一致性。

```
class Book {
    protected String name;
    public String getName() {...}
    public void setName(String name) {...}
}
```

这个`Book`类也有`name`字段，那么，我们能不能让`Student`继承自`Book`呢?

```
class Student extends Book {
    protected int score;
}
```

显然，从逻辑上讲，这是不合理的，`Student`不应该从`Book`继承，而应该从`Person`继承。

究其原因，是因为`Student`是`Person`的一种，它们是is关系，而`Student`并不是`Book`。实际上`Student`和`Book`的关系是has关系。

具有has关系不应该使用继承，而是使用组合，即`Student`可以持有一个`Book`实例：

```
class Student extends Person {
    protected Book book;
    protected int score;
}
```

因此，继承是is关系，组合是has关系。

## （12）多态

在继承关系中，子类如果定义了一个与父类方法签名完全相同的方法，被称为覆写（Override）。

例如，在`Person`类中，我们定义了`run()`方法：

```
class Person {
    public void run() {
        System.out.println("Person.run");
    }
}
```

在子类`Student`中，覆写这个`run()`方法：

```
class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

Override和Overload不同的是，如果方法签名如果不同，就是Overload，Overload方法是一个新方法；如果方法签名相同，并且返回值也相同，就是`Override`。

```
class Person {
    public void run() { … }
}

class Student extends Person {
    // 不是Override，因为参数不同：
    public void run(String s) { … }
    // 不是Override，因为返回值不同：
    public int run() { … }
}
```

加上`@Override`可以让编译器帮助检查是否进行了正确的覆写。希望进行覆写，但是不小心写错了方法签名，编译器会报错。

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

class Person {
    public void run() {}
}

public class Student extends Person {
    @Override // Compile error!
    public void run(String s) {}
}
```

但是`@Override`不是必需的。引用变量的声明类型可能与其实际类型不符，例如：

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

如果子类覆写了父类的方法：

```
public class Main {
    public static void main(String[] args) {
        Person p = new Student();
        p.run(); // Student.run
    }
}

class Person {
    public void run() {
        System.out.println("Person.run");
    }
}

class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}

```

Java的实例方法调用是基于运行时的实际类型的动态调用，而非变量的声明类型。

这个非常重要的特性在面向对象编程中称之为多态。多态是指，针对某个类型的方法调用，其真正执行的方法取决于运行时期实际类型的方法。

```
public void runTwice(Person p) {
    p.run();
    p.run();
}
```

它传入的参数类型是`Person`，我们是无法知道传入的参数实际类型究竟是`Person`，还是`Student`，还是`Person`的其他子类，因此，也无法确定调用的是不是`Person`类定义的`run()`方法。

假设我们定义一种收入，需要给它报税，那么先定义一个`Income`类：

```
class Income {
    protected double income;
    public double getTax() {
        return income * 0.1; // 税率10%
    }
}
```

对于工资收入，可以减去一个基数，那么我们可以从`Income`派生出`SalaryIncome`，并覆写`getTax()`：

```
class Salary extends Income {
    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}
```

如果你享受国务院特殊津贴，那么按照规定，可以全部免税：

```
class StateCouncilSpecialAllowance extends Income {
    @Override
    public double getTax() {
        return 0;
    }
}
```

现在，我们要编写一个报税的财务软件，对于一个人的所有收入进行报税，可以这么写：

```
public double totalTax(Income... incomes) {
    double total = 0;
    for (Income income: incomes) {
        total = total + income.getTax();
    }
    return total;
}
```

完整的实例代码如下：

```
public class Main {
    public static void main(String[] args) {
        // 给一个有普通收入、工资收入和享受国务院特殊津贴的小伙伴算税:
        Income[] incomes = new Income[] {
            new Income(3000),
            new Salary(7500),
            new StateCouncilSpecialAllowance(15000)
        };
```

```
        System.out.println(totalTax(incomes));
    }

    public static double totalTax(Income... incomes) {
        double total = 0;
        for (Income income: incomes) {
            total = total + income.getTax();
        }
        return total;
    }
}

class Income {
    protected double income;

    public Income(double income) {
        this.income = income;
    }

    public double getTax() {
        return income * 0.1; // 税率10%
    }
}

class Salary extends Income {
    public Salary(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}

class StateCouncilSpecialAllowance extends Income {
    public StateCouncilSpecialAllowance(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        return 0;
    }
}

```

## （13）覆盖Object方法

因为所有的`class`最终都继承自`Object`，而`Object`定义了几个重要的方法：

- `toString()`：把instance输出为`String`；
- `equals()`：判断两个instance是否逻辑相等；
- `hashCode()`：计算一个instance的哈希值。

在必要的时候，我们可以覆写`Object`的这几个方法。

```
class Person {
    ...
    // 显示更有意义的字符串:
    @Override
    public String toString() {
        return "Person:name=" + name;
    }

    // 比较是否相等:
    @Override
    public boolean equals(Object o) {
        // 当且仅当o为Person类型:
        if (o instanceof Person) {
            Person p = (Person) o;
            // 并且name字段相同时，返回true:
            return this.name.equals(p.name);
        }
        return false;
    }

    // 计算hash:
    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
```

## （14）调用super

在子类的覆写方法中，如果要调用父类的被覆写的方法，可以通过`super`来调用。

```
class Person {
    protected String name;
    public String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    @Override
    public String hello() {
        // 调用父类的hello()方法:
        return super.hello() + "!";
    }
}
```

## （15）final

继承可以允许子类覆写父类的方法。如果一个父类不允许子类对它的某个方法进行覆写，可以把该方法标记为`final`。用`final`修饰的方法不能被`Override`。

```
class Person {
    protected String name;
    public final String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    // compile error: 不允许覆写
    @Override
    public String hello() {
    }
}
```

如果一个类不希望任何其他类继承自它，那么可以把这个类本身标记为`final`。用`final`修饰的类不能被继承。

```
final class Person {
    protected String name;
}

// compile error: 不允许继承自Person
Student extends Person {
}
```

对于一个类的实例字段，同样可以用`final`修饰。用`final`修饰的字段在初始化后不能被修改。

```
class Person {
    public final String name = "Unamed";
}
```

对`final`字段重新赋值会报错。

```
Person p = new Person();
p.name = "New Name"; // compile error!
```

可以在构造方法中初始化final字段。

```
class Person {
    public final String name;
    public Person(String name) {
        this.name = name;
    }
}
```

这种方法更为常用，因为可以保证实例一旦创建，其`final`字段就不可修改。

## （16）抽象类

如果父类的方法本身不需要实现任何功能，仅仅是为了定义方法签名，目的是让子类去覆写它，那么，可以把父类的方法声明为抽象方法。

```
class Person {
    public abstract void run();
}
```

把一个方法声明为`abstract`，表示它是一个抽象方法，本身没有实现任何方法语句。因为这个抽象方法本身是无法执行的，所以，`Person`类也无法被实例化。编译器会告诉我们，无法编译`Person`类，因为它包含抽象方法。

必须把`Person`类本身也声明为`abstract`，才能正确编译它。

```
abstract class Person {
    public abstract void run();
}
```

如果一个`class`定义了方法，但没有具体执行代码，这个方法就是抽象方法，抽象方法用`abstract`修饰。

因为无法执行抽象方法，因此这个类也必须申明为抽象类（abstract class）。

使用`abstract`修饰的类就是抽象类。我们无法实例化一个抽象类。因为抽象类本身被设计成只能用于被继承，因此，抽象类可以强迫子类实现其定义的抽象方法，否则编译会报错。因此，抽象方法实际上相当于定义了“规范”。

```
public class Main {
    public static void main(String[] args) {
        Person p = new Student();
        p.run();
    }
}

abstract class Person {
    public abstract void run();
}

class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

当我们定义了抽象类`Person`，以及具体的`Student`、`Teacher`子类的时候，我们可以通过抽象类`Person`类型去引用具体的子类的实例。

```
Person s = new Student();
Person t = new Teacher();
```

这种引用抽象类的好处在于，我们对其进行方法调用，并不关心`Person`类型变量的具体子类型。

```
// 不关心Person变量的具体子类型:
s.run();
t.run();
```

这种尽量引用高层类型，避免引用实际子类型的方式，称之为面向抽象编程。

面向抽象编程的本质就是：

- 上层代码只定义规范（例如：`abstract class Person`）；
- 不需要子类就可以实现业务逻辑（正常编译）；
- 具体的业务逻辑由不同的子类实现，调用者并不关心。

## （17）接口

在抽象类中，抽象方法本质上是定义接口规范：即规定高层类的接口，从而保证所有子类都有相同的接口实现，这样，多态就能发挥出威力。如果一个抽象类没有字段，所有方法全部都是抽象方法：

```
abstract class Person {
    public abstract void run();
    public abstract String getName();
}
```

就可以把该抽象类改写为接口：`interface`。在Java中，使用`interface`可以声明一个接口：

```
interface Person {
    void run();
    String getName();
}
```

所谓`interface`，就是比抽象类还要抽象的纯抽象接口，因为它连字段都不能有。因为接口定义的所有方法默认都是`public abstract`的，所以这两个修饰符不需要写出来（写不写效果都一样）。

当一个具体的`class`去实现一个`interface`时，需要使用`implements`关键字。

```
class Student implements Person {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(this.name + " run");
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

在Java中，一个类只能继承自另一个类，不能从多个类继承。但是，一个类可以实现多个`interface`。

```
class Student implements Person, Hello { // 实现了两个interface
    ...
}
```

Java的接口特指`interface`的定义，表示一个接口类型和一组方法签名，而编程接口泛指接口规范，如方法签名，数据格式，网络协议等。一个`interface`可以继承自另一个`interface`。`interface`继承自`interface`使用`extends`，它相当于扩展了接口的方法。

```
interface Hello {
    void hello();
}

interface Person extends Hello {
    void run();
    String getName();
}
```

此时，`Person`接口继承自`Hello`接口，因此，`Person`接口现在实际上有3个抽象方法签名，其中一个来自继承的`Hello`接口。接口中，可以定义`default`方法。例如，把`Person`接口的`run()`方法改为`default`方法：

```
public class Main {
    public static void main(String[] args) {
        Person p = new Student("Xiao Ming");
        p.run();
    }
}

interface Person {
    String getName();
    default void run() {
        System.out.println(getName() + " run");
    }
}

class Student implements Person {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}

```

实现类可以不必覆写`default`方法。`default`方法的目的是，当我们需要给接口新增一个方法时，会涉及到修改全部子类。如果新增的是`default`方法，那么子类就不必全部修改，只需要在需要覆写的地方去覆写新增方法。

## （18）静态字段和静态方法

在一个`class`中定义的字段，我们称之为实例字段。实例字段的特点是，每个实例都有独立的字段，各个实例的同名字段互不影响。还有一种字段，是用`static`修饰的字段，称为静态字段：`static field`。实例字段在每个实例中都有自己的一个独立“空间”，但是静态字段只有一个共享“空间”，所有实例都会共享该字段。

```
public class Main {
    public static void main(String[] args) {
        Person ming = new Person("Xiao Ming", 12);
        Person hong = new Person("Xiao Hong", 15);
```

```
        ming.number = 88;
        System.out.println(hong.number);//88
        hong.number = 99;
        System.out.println(ming.number);//99
        System.out.println(hong.number);//99
    }
}

class Person {
    public String name;
    public int age;

    public static int number;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

```

对于静态字段，无论修改哪个实例的静态字段，效果都是一样的：所有实例的静态字段都被修改了，原因是静态字段并不属于实例，虽然实例可以访问静态字段，但是它们指向的其实都是`Person class`的静态字段。所以，所有实例共享一个静态字段。

因此，不推荐用`实例变量.静态字段`去访问静态字段，因为在Java程序中，实例对象并没有静态字段。在代码中，实例对象能访问静态字段只是因为编译器可以根据实例类型自动转换为`类名.静态字段`来访问静态对象。

推荐用类名来访问静态字段。可以把静态字段理解为描述`class`本身的字段（非实例字段）。对于上面的代码，更好的写法是：

```
Person.number = 99;
System.out.println(Person.number);
```

有静态字段，就有静态方法。用`static`修饰的方法称为静态方法。

调用实例方法必须通过一个实例变量，而调用静态方法则不需要实例变量，通过类名就可以调用。静态方法类似其它编程语言的函数。

```
public class Main {
    public static void main(String[] args) {
        Person.setNumber(99);
        System.out.println(Person.number);
    }
}

class Person {
    public static int number;

    public static void setNumber(int value) {
        number = value;
    }
}
```

因为静态方法属于`class`而不属于实例，因此，静态方法内部，无法访问`this`变量，也无法访问实例字段，它只能访问静态字段。

通过实例变量也可以调用静态方法，但这只是编译器自动帮我们把实例改写成类名而已。

通常情况下，通过实例变量访问静态字段和静态方法，会得到一个编译警告。

静态方法经常用于工具类。例如：

- Arrays.sort()
- Math.random()

静态方法也经常用于辅助方法。注意到Java程序的入口`main()`也是静态方法。

因为`interface`是一个纯抽象类，所以它不能定义实例字段。但是，`interface`是可以有静态字段的，并且静态字段必须为`final`类型：

```
public interface Person {
    public static final int MALE = 1;
    public static final int FEMALE = 2;
}
```

实际上，因为`interface`的字段只能是`public static final`类型，所以我们可以把这些修饰符都去掉，上述代码可以简写为：

```
public interface Person {
    // 编译器会自动加上public statc final:
    int MALE = 1;
    int FEMALE = 2;
}
```

编译器会自动把该字段变为`public static final`类型。

## （19）包

在Java中，我们使用`package`来解决名字冲突。

Java定义了一种名字空间，称之为包：`package`。一个类总是属于某个包，类名（比如`Person`）只是一个简写，真正的完整类名是`包名.类名`。

```
package ming; // 申明包名ming

public class Person {
}
```

## （20）内部类

`Outer`是一个普通类，而`Inner`是一个Inner Class，它与普通类有个最大的不同，就是Inner Class的实例不能单独存在，必须依附于一个Outer Class的实例。

```
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested"); // 实例化一个Outer
        Outer.Inner inner = outer.new Inner(); // 实例化一个Inner
        inner.hello();
    }
```

```
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    class Inner {
        void hello() {
            System.out.println("Hello, " + Outer.this.name);
        }
    }
}

```

nner Class和普通Class相比，除了能引用Outer实例外，还有一个额外的“特权”，就是可以修改Outer Class的`private`字段，因为Inner Class的作用域在Outer Class内部，所以能访问Outer Class的`private`字段和方法。

# 5、Java核心类

## （1）字符串和编码

在Java中，`String`是一个引用类型，它本身也是一个`class`。但是，Java编译器对`String`有特殊处理，即可以直接用`"..."`来表示一个字符串。实际上字符串在`String`内部是通过一个`char[]`数组表示的，因此，可以使用如下方法表示字符串：

```
String s2 = new String(new char[] {'H', 'e', 'l', 'l', 'o', '!'});
String s1 = "Hello!";
```

Java字符串的一个重要特点就是字符串*不可变*。这种不可变性是通过内部的`private final char[]`字段，以及没有任何修改`char[]`的方法实现的。当我们想要比较两个字符串是否相同时，要特别注意，我们实际上是想比较字符串的内容是否相同。必须使用`equals()`方法而不能用`==`。Java编译器在编译期，会自动把所有相同的字符串当作一个对象放入常量池。`String`类还提供了多种方法来搜索子串、提取子串。常用的方法有：

```
// 是否包含子串:
"Hello".contains("ll"); // true
```

注意到`contains()`方法的参数是`CharSequence`而不是`String`，因为`CharSequence`是`String`的父类。

```
"Hello".indexOf("l"); // 2
"Hello".lastIndexOf("l"); // 3
"Hello".startsWith("He"); // true
"Hello".endsWith("lo"); // true
"Hello".substring(2); // "llo"
"Hello".substring(2, 4); "ll"
```

使用`trim()`方法可以移除字符串首尾空白字符。空白字符包括空格，`\t`，`\r`，`\n`：

```
"  \tHello\r\n ".trim(); // "Hello"
```

注意：`trim()`并没有改变字符串的内容，而是返回了一个新字符串。

另一个`strip()`方法也可以移除字符串首尾空白字符。它和`trim()`不同的是，类似中文的空格字符`\u3000`也会被移除：

```
"\u3000Hello\u3000".strip(); // "Hello"
" Hello ".stripLeading(); // "Hello "
" Hello ".stripTrailing(); // " Hello"
```

`String`还提供了`isEmpty()`和`isBlank()`来判断字符串是否为空和空白字符串：

```
"".isEmpty(); // true，因为字符串长度为0
"  ".isEmpty(); // false，因为字符串长度不为0
"  \n".isBlank(); // true，因为只包含空白字符
" Hello ".isBlank(); // false，因为包含非空白字符
```

要在字符串中替换子串，有两种方法。一种是根据字符或字符串替换：

```
String s = "hello";
s.replace('l', 'w'); // "hewwo"，所有字符'l'被替换为'w'
s.replace("ll", "~~"); // "he~~o"，所有子串"ll"被替换为"~~"
```

另一种是通过正则表达式替换：

```
String s = "A,,B;C ,D";
s.replaceAll("[\\,\\;\\s]+", ","); // "A,B,C,D"
```

要分割字符串，使用`split()`方法，并且传入的也是正则表达式：

```
String s = "A,B,C,D";
String[] ss = s.split("\\,"); // {"A", "B", "C", "D"}
```

拼接字符串使用静态方法`join()`，它用指定的字符串连接字符串数组：

```
String[] arr = {"A", "B", "C"};
String s = String.join("***", arr); // "A***B***C"
```

字符串提供了`formatted()`方法和`format()`静态方法，可以传入其他参数，替换占位符，然后生成新的字符串：

```
public class Main {
    public static void main(String[] args) {
        String s = "Hi %s, your score is %d!";
        System.out.println(s.formatted("Alice", 80));
        System.out.println(String.format("Hi %s, your score is %.2f!", "Bob",
59.5));
    }
}

```

有几个占位符，后面就传入几个参数。参数类型要和占位符一致。我们经常用这个方法来格式化信息。常用的占位符有：

- `%s`：显示字符串；
- `%d`：显示整数；
- `%x`：显示十六进制整数；
- `%f`：显示浮点数。

占位符还可以带格式，例如`%.2f`表示显示两位小数。如果你不确定用啥占位符，那就始终用`%s`，因为`%s`可以显示任何数据类型。

要把任意基本类型或引用类型转换为字符串，可以使用静态方法`valueOf()`。这是一个重载方法，编译器会根据参数自动选择合适的方法：

```
String.valueOf(123); // "123"
String.valueOf(45.67); // "45.67"
String.valueOf(true); // "true"
String.valueOf(new Object()); // 类似java.lang.Object@636be97c
```

要把字符串转换为其他类型，就需要根据情况。例如，把字符串转换为`int`类型：

```
int n1 = Integer.parseInt("123"); // 123
int n2 = Integer.parseInt("ff", 16); // 按十六进制转换，255
```

把字符串转换为`boolean`类型：

```
boolean b1 = Boolean.parseBoolean("true"); // true
boolean b2 = Boolean.parseBoolean("FALSE"); // false
```

要特别注意，`Integer`有个`getInteger(String)`方法，它不是将字符串转换为`int`，而是把该字符串对应的系统变量转换为`Integer`：

```
Integer.getInteger("java.version"); // 版本号，11
```

`String`和`char[]`类型可以互相转换，方法是：

```
char[] cs = "Hello".toCharArray(); // String -> char[]
String s = new String(cs); // char[] -> String
```

如果修改了`char[]`数组，`String`并不会改变。这是因为通过`new String(char[])`创建新的`String`实例时，它并不会直接引用传入的`char[]`数组，而是会复制一份，所以，修改外部的`char[]`数组不会影响`String`实例内部的`char[]`数组，因为这是两个不同的数组。从`String`的不变性设计可以看出，如果传入的对象有可能改变，我们需要复制而不是直接引用。

```
public class Main {
    public static void main(String[] args) {
        int[] scores = new int[] { 88, 77, 51, 66 };
        Score s = new Score(scores);
        s.printScores();//[88, 77, 51, 66]
        scores[2] = 99;
        s.printScores();//[88, 77, 99, 66]
    }
}
```

```
10
11 class Score {
12     private int[] scores;
13     public Score(int[] scores) {
14         this.scores = scores;
15     }
16
17     public void printScores() {
18         System.out.println(Arrays.toString(scores));
19     }
20 }
21
```

由于`Score`内部直接引用了外部传入的`int[]`数组，这会造成外部代码对`int[]`数组的修改，影响到`Score`类的字段。如果外部代码不可信，这就会造成安全隐患。

```
1  class Score {
2
3      private int[] scores;
4
5      public Score(int[] scores) {
6
7          this.scores = scores.clone();//外部代码不再影响scores
8
9      }
10
11     public void printScores() {
12
13         System.out.println(Arrays.toString(scores));
14
15     }
16
17 }
```

在早期的计算机系统中，为了给字符编码，美国国家标准学会（American National Standard Institute：ANSI）制定了一套英文字母、数字和常用符号的编码，它占用一个字节，编码范围从`0`到`127`，最高位始终为`0`，称为`ASCII`编码。例如，字符`'A'`的编码是`0x41`，字符`'1'`的编码是`0x31`。

如果要把汉字也纳入计算机编码，很显然一个字节是不够的。`GB2312`标准使用两个字节表示一个汉字，其中第一个字节的最高位始终为`1`，以便和`ASCII`编码区分开。例如，汉字`'中'`的`GB2312`编码是`0xd6d0`。

类似的，日文有`Shift_JIS`编码，韩文有`EUC-KR`编码，这些编码因为标准不统一，同时使用，就会产生冲突。

为了统一全球所有语言的编码，全球统一码联盟发布了`Unicode`编码，它把世界上主要语言都纳入同一个编码，这样，中文、日文、韩文和其他语言就不会冲突。因为英文字符的`Unicode`编码高字节总是`00`，包含大量英文的文本会浪费空间，所以，出现了`UTF-8`编码，它是一种变长编码，用来把固定长度的`Unicode`编码变成1～4字节的变长编码。通过`UTF-8`编码，英文字符`'A'`的`UTF-8`编码变为`0x41`，正好和`ASCII`码一致，而中文`'中'`的`UTF-8`编码为3字节`0xe4b8ad`。

`UTF-8`编码的另一个好处是容错能力强。如果传输过程中某些字符出错，不会影响后续字符，因为`UTF-8`编码依靠高字节位来确定一个字符究竟是几个字节，它经常用来作为传输编码。

在Java中，`char`类型实际上就是两个字节的`Unicode`编码。如果我们要手动把字符串转换成其他编码，可以这样做：

```
byte[] b1 = "Hello".getBytes(); // 按系统默认编码转换，不推荐
byte[] b2 = "Hello".getBytes("UTF-8"); // 按UTF-8编码转换
byte[] b2 = "Hello".getBytes("GBK"); // 按GBK编码转换
byte[] b3 = "Hello".getBytes(StandardCharsets.UTF_8); // 按UTF-8编码转换
```

注意：转换编码后，就不再是`char`类型，而是`byte`类型表示的数组。

如果要把已知编码的`byte[]`转换为`String`，可以这样做：

```
byte[] b = ...
String s1 = new String(b, "GBK"); // 按GBK转换
String s2 = new String(b, StandardCharsets.UTF_8); // 按UTF-8转换
```

Java的`String`和`char`在内存中总是以Unicode编码表示。对于不同版本的JDK，`String`类在内存中有不同的优化方式。具体来说，早期JDK版本的`String`总是以`char[]`存储，它的定义如下：

```
public final class String {
    private final char[] value;
    private final int offset;
    private final int count;
}
```

而较新的JDK版本的`String`则以`byte[]`存储：如果`String`仅包含ASCII字符，则每个`byte`存储一个字符，否则，每两个`byte`存储一个字符，这样做的目的是为了节省内存，因为大量的长度较短的`String`通常仅包含ASCII字符：

```
public final class String {
    private final byte[] value;
    private final byte coder; // 0 = LATIN1, 1 = UTF16
```

对于使用者来说，`String`内部的优化不影响任何已有代码，因为它的`public`方法签名是不变的。

### （2）StringBuilder

Java编译器对`String`做了特殊处理，使得我们可以直接用`+`拼接字符串。

```
String s = "";
for (int i = 0; i < 1000; i++) {
    s = s + "," + i;
}
```

虽然可以直接拼接字符串，但是，在循环中，每次循环都会创建新的字符串对象，然后扔掉旧的字符串。这样，绝大部分字符串都是临时对象，不但浪费内存，还会影响GC效率。

为了能高效拼接字符串，Java标准库提供了`StringBuilder`，它是一个可变对象，可以预分配缓冲区，这样，往`StringBuilder`中新增字符时，不会创建新的临时对象：

```
StringBuilder sb = new StringBuilder(1024);
for (int i = 0; i < 1000; i++) {
    sb.append(',');
    sb.append(i);
}
String s = sb.toString();
```

`StringBuilder`还可以进行链式操作：

```
public class Main {
    public static void main(String[] args) {
        var sb = new StringBuilder(1024);
        sb.append("Mr ")
          .append("Bob")
          .append("!")
          .insert(0, "Hello, ");
        System.out.println(sb.toString());
    }
}
```

### （3）StringJoiner

类似用分隔符拼接数组的需求很常见，所以Java标准库还提供了一个`StringJoiner`来干这个事：

```
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sj = new StringJoiner(", ");
        for (String name : names) {
            sj.add(name);
        }
        System.out.println(sj.toString());
    }
}

```

用`StringJoiner`的结果少了前面的`"Hello "`和结尾的`"!"`！遇到这种情况，需要给`StringJoiner`指定“开头”和“结尾”：

```
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sj = new StringJoiner(", ", "Hello ", "!");
        for (String name : names) {
            sj.add(name);
        }
        System.out.println(sj.toString());
    }
}
```

`String`还提供了一个静态方法`join()`，这个方法在内部使用了`StringJoiner`来拼接字符串，在不需要指定“开头”和“结尾”的时候，用`String.join()`更方便：

```
String[] names = {"Bob", "Alice", "Grace"};
var s = String.join(", ", names);
```

### （4）包装类型

Java的数据类型分两种：

- 基本类型：`byte`，`short`，`int`，`long`，`boolean`，`float`，`double`，`char`
- 引用类型：所有`class`和`interface`类型

引用类型可以赋值为`null`，表示空，但基本类型不能赋值为`null`。

如何把一个基本类型视为对象（引用类型）？

比如，想要把`int`基本类型变成一个引用类型，我们可以定义一个`Integer`类，它只包含一个实例字段`int`，这样，`Integer`类就可以视为`int`的包装类（Wrapper Class）：

```
public class Integer {
    private int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return this.value;
    }
}
```

定义好了`Integer`类，我们就可以把`int`和`Integer`互相转换：

```
Integer n = null;
Integer n2 = new Integer(99);
int n3 = n2.intValue();
```

| 基本类型 | 对应的引用类型 |
| --- | --- |
| boolean | java.lang.Boolean |
| byte | java.lang.Byte |
| short | java.lang.Short |
| int | java.lang.Integer |
| long | java.lang.Long |
| float | java.lang.Float |
| double | java.lang.Double |
| char | java.lang.Character |

使用方法：

```
public class Main {
    public static void main(String[] args) {
        int i = 100;
        // 通过new操作符创建Integer实例(不推荐使用,会有编译警告):
        Integer n1 = new Integer(i);
        // 通过静态方法valueOf(int)创建Integer实例:
        Integer n2 = Integer.valueOf(i);
        // 通过静态方法valueOf(String)创建Integer实例:
        Integer n3 = Integer.valueOf("100");
        System.out.println(n3.intValue());
    }
}
```

因为`int`和`Integer`可以互相转换：

```
int i = 100;
Integer n = Integer.valueOf(i);
int x = n.intValue();
```

Java编译器可以帮助我们自动在`int`和`Integer`之间转型：

```
Integer n = 100; // 编译器自动使用Integer.valueOf(int)
int x = n; // 编译器自动使用Integer.intValue()
```

这种直接把`int`变为`Integer`的赋值写法，称为自动装箱（Auto Boxing），反过来，把`Integer`变为`int`的赋值写法，称为自动拆箱（Auto Unboxing）。

注意：自动装箱和自动拆箱只发生在编译阶段，目的是为了少写代码。

装箱和拆箱会影响代码的执行效率，因为编译后的`class`代码是严格区分基本类型和引用类型的。并且，自动拆箱执行时可能会报`NullPointerException`：

```
public class Main {
    public static void main(String[] args) {
        Integer n = null;
        int i = n;
    }
}

```

所有的包装类型都是不变类。我们查看`Integer`的源码可知，它的核心代码如下：

```
public final class Integer {
    private final int value;
}
```

因此，一旦创建了`Integer`对象，该对象就是不变的。

对两个`Integer`实例进行比较要特别注意：绝对不能用`==`比较，因为`Integer`是引用类型，必须使用`equals()`比较。

```
public class Main {
    public static void main(String[] args) {
        Integer x = 127;
        Integer y = 127;
        Integer m = 99999;
        Integer n = 99999;
        System.out.println("x == y: " + (x==y)); // true
        System.out.println("m == n: " + (m==n)); // false
        System.out.println("x.equals(y): " + x.equals(y)); // true
        System.out.println("m.equals(n): " + m.equals(n)); // true
    }
}
```

因为`Integer.valueOf()`可能始终返回同一个`Integer`实例，因此，在我们自己创建`Integer`的时候，以下两种方法：

- 方法1：`Integer n = new Integer(100);`
- 方法2：`Integer n = Integer.valueOf(100);`

方法2更好，因为方法1总是创建新的`Integer`实例，方法2把内部优化留给`Integer`的实现者去做，即使在当前版本没有优化，也有可能在下一个版本进行优化。

我们把能创建“新”对象的静态方法称为静态工厂方法。`Integer.valueOf()`就是静态工厂方法，它尽可能地返回缓存的实例以节省内存。

### （5）枚举

```
public class Main {
    public static void main(String[] args) {
        for (Weekday day : Weekday.values())
{
        if (day == Weekday.SAT || day == Weekday.SUN) {
            System.out.println("Work at home!");
        } else {
            System.out.println("Work at office!");
        }
}
    }
}

enum Weekday {
    SUN, MON, TUE, WED, THU, FRI, SAT;
}

```

但是，如果不小心修改了枚举的顺序，编译器是无法检查出这种逻辑错误的。要编写健壮的代码，就不要依靠`ordinal()`的返回值。因为`enum`本身是`class`，所以我们可以定义`private`的构造方法，并且，给每个枚举常量添加字段：

```
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.WED;
        if (day.dayValue == 6 || day.dayValue == 0) {
            System.out.println("Work at home!");
        } else {
            System.out.println("Work at office!");
```

```
        }
    }
}

enum Weekday {
    MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6), SUN(0);

    public final int dayValue;

    private Weekday(int dayValue) {
        this.dayValue = dayValue;
    }
}

```

## （6）BigInteger和BigDecimal

在Java中，由CPU原生提供的整型最大范围是64位`long`型整数。使用`long`型整数可以直接通过CPU指令进行计算，速度非常快。

如果我们使用的整数范围超过了`long`型怎么办？这个时候，就只能用软件来模拟一个大整数。`java.math.BigInteger`就是用来表示任意大小的整数。`BigInteger`内部用一个`int[]`数组来模拟一个非常大的整数：

```
BigInteger bi = new BigInteger("1234567890");
System.out.println(bi.pow(5)); //
2867971860299718107233761438093672048294900000
```

对`BigInteger`做运算的时候，只能使用实例方法，例如，加法运算：

```
BigInteger i1 = new BigInteger("1234567890");
BigInteger i2 = new BigInteger("12345678901234567890");
BigInteger sum = i1.add(i2); // 12345678902469135780

BigInteger i = new BigInteger("123456789000");
System.out.println(i.longValue()); // 123456789000
System.out.println(i.multiply(i).longValueExact()); //
java.lang.ArithmeticException: BigInteger out of long range
```

使用`longValueExact()`方法时，如果超出了`long`型的范围，会抛出`ArithmeticException`。

`BigInteger`和`Integer`、`Long`一样，也是不可变类，并且也继承自`Number`类。因为`Number`定义了转换为基本类型的几个方法：

- 转换为`byte`：`byteValue()`
- 转换为`short`：`shortValue()`
- 转换为`int`：`intValue()`
- 转换为`long`：`longValue()`
- 转换为`float`：`floatValue()`
- 转换为`double`：`doubleValue()`

因此，通过上述方法，可以把`BigInteger`转换成基本类型。如果`BigInteger`表示的范围超过了基本类型的范围，转换时将丢失高位信息，即结果不一定是准确的。如果需要准确地转换成基本类型，可以使用`intValueExact()`、`longValueExact()`等方法，在转换时如果超出范围，将直接抛出`ArithmeticException`异常。

和`BigInteger`类似，`BigDecimal`可以表示一个任意大小且精度完全准确的浮点数。

```
BigDecimal bd = new BigDecimal("123.4567");
System.out.println(bd.multiply(bd)); // 15241.55677489
```

`BigDecimal`用`scale()`表示小数位数，例如：

```
BigDecimal d1 = new BigDecimal("123.45");
BigDecimal d2 = new BigDecimal("123.4500");
BigDecimal d3 = new BigDecimal("1234500");
System.out.println(d1.scale()); // 2,两位小数
System.out.println(d2.scale()); // 4
System.out.println(d3.scale()); // 0
```

通过`BigDecimal`的`stripTrailingZeros()`方法，可以将一个`BigDecimal`格式化为一个相等的，但去掉了末尾0的`BigDecimal`：

```
BigDecimal d1 = new BigDecimal("123.4500");
BigDecimal d2 = d1.stripTrailingZeros();
System.out.println(d1.scale()); // 4
System.out.println(d2.scale()); // 2,因为去掉了00

BigDecimal d3 = new BigDecimal("1234500");
BigDecimal d4 = d3.stripTrailingZeros();
System.out.println(d3.scale()); // 0
System.out.println(d4.scale()); // -2
```

对`BigDecimal`做加、减、乘时，精度不会丢失，但是做除法时，存在无法除尽的情况，这时，就必须指定精度以及如何进行截断。

## 6、异常处理

Java内置了一套异常处理机制，总是使用异常来表示错误。

Java标准库定义的常用异常包括：

```
Exception
│
├─ RuntimeException
│  │
│  ├─ NullPointerException
│  │
│  ├─ IndexOutOfBoundsException
│  │
│  ├─ SecurityException
│  │
│  └─ IllegalArgumentException
│     │
│     └─ NumberFormatException
│
├─ IOException
│  │
│  ├─ UnsupportedCharsetException
│  │
│  ├─ FileNotFoundException
```

```
|  |
|  └─ SocketException
|
├─ ParseException
|
├─ GeneralSecurityException
|
├─ SQLException
|
└─ TimeoutException
```

异常是一种`class`，因此它本身带有类型信息。异常可以在任何地方抛出，但只需要在上层捕获，这样就和方法调用分离了：

```
try {
    String s = processFile(“C:\\test.txt”);
    // ok:
} catch (FileNotFoundException e) {
    // file not found:
} catch (SecurityException e) {
    // no read permission:
} catch (IOException e) {
    // io error:
} catch (Exception e) {
    // other error:
}
```

从继承关系可知：`Throwable`是异常体系的根，它继承自`Object`。`Throwable`有两个体系：`Error`和`Exception`，`Error`表示严重的错误，程序对此一般无能为力，例如：

- `OutOfMemoryError`：内存耗尽
- `NoClassDefFoundError`：无法加载某个Class
- `StackOverflowError`：栈溢出

而`Exception`则是运行时的错误，它可以被捕获并处理。

某些异常是应用程序逻辑处理的一部分，应该捕获并处理。例如：

- `NumberFormatException`：数值类型的格式错误
- `FileNotFoundException`：未找到文件
- `SocketException`：读取网络失败

还有一些异常是程序逻辑编写不对造成的，应该修复程序本身。例如：

- `NullPointerException`：对某个`null`的对象调用方法或字段
- `IndexOutOfBoundsException`：数组索引越界

`Exception`又分为两大类：

1. `RuntimeException`以及它的子类；
2. 非`RuntimeException`（包括`IOException`、`ReflectiveOperationException`等等）

Java规定：

- 必须捕获的异常，包括`Exception`及其子类，但不包括`RuntimeException`及其子类，这种类型的异常称为Checked Exception。
- 不需要捕获的异常，包括`Error`及其子类，`RuntimeException`及其子类。

在方法定义的时候，使用`throws Xxx`表示该方法可能抛出的异常类型。调用方在调用的时候，必须强制捕获这些异常，否则编译器会报错。

Java使用异常来表示错误，并通过`try ... catch`捕获异常；

Java的异常是`class`，并且从`Throwable`继承；

`Error`是无需捕获的严重错误，`Exception`是应该捕获的可处理的错误；

`RuntimeException`无需强制捕获，非`RuntimeException`（Checked Exception）需强制捕获，或者用`throws`声明；

不推荐捕获了异常但不进行任何处理。

可以使用多个`catch`语句，每个`catch`分别捕获对应的`Exception`及其子类。JVM在捕获到异常后，会从上到下匹配`catch`语句，匹配到某个`catch`后，执行`catch`代码块，然后*不再*继续匹配。

简单地说就是：多个`catch`语句只有一个能被执行。

存在多个`catch`的时候，`catch`的顺序非常重要：子类必须写在前面。

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println("IO error");
    } catch (UnsupportedEncodingException e) { // 永远捕获不到
        System.out.println("Bad encoding");
    }
}
```

对于上面的代码，`UnsupportedEncodingException`异常是永远捕获不到的，因为它是`IOException`的子类。当抛出`UnsupportedEncodingException`异常时，会被`catch (IOException e) { ... }`捕获并执行。

因此，正确的写法是把子类放到前面。无论是否有异常发生，如果我们都希望执行一些语句，例如清理工作，怎么写？

可以把执行语句写若干遍：正常执行的放到`try`中，每个`catch`再写一遍。Java的`try ... catch`机制还提供了`finally`语句，`finally`语句块保证有无错误都会执行。

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    } finally {
        System.out.println("END");
    }
}
```

注意`finally`有几个特点：

1. `finally`语句不是必须的，可写可不写；
2. `finally`总是最后执行。

如果没有发生异常，就正常执行`try { ... }`语句块，然后执行`finally`。如果发生了异常，就中断执行`try { ... }`语句块，然后跳转执行匹配的`catch`语句块，最后执行`finally`。

可见，`finally`是用来保证一些代码必须执行的。

某些情况下，可以没有`catch`，只使用`try ... finally`结构。

```
void process(String file) throws IOException {
    try {
        ...
    } finally {
        System.out.println("END");
    }
}
```

因为方法声明了可能抛出的异常，所以可以不写`catch`。

断言（Assertion）是一种调试程序的方式。在Java中，使用`assert`关键字来实现断言。

```
public static void main(String[] args) {
    double x = Math.abs(-123.45);
    assert x >= 0;
    System.out.println(x);
}
```

语句`assert x >= 0;`即为断言，断言条件`x >= 0`预期为`true`。如果计算结果为`false`，则断言失败，抛出`AssertionError`。

使用`assert`语句时，还可以添加一个可选的断言消息：

```
assert x >= 0 : "x must >= 0";
```

## 7、集合

Java标准库自带的`java.util`包提供了集合类：`Collection`，它是除`Map`外所有其他集合类的根接口。Java的`java.util`包主要提供了以下三种类型的集合：

- `List`：一种有序列表的集合，例如，按索引排列的`Student`的`List`；
- `Set`：一种保证没有重复元素的集合，例如，所有无重复名称的`Student`的`Set`；
- `Map`：一种通过键值（key-value）查找的映射表集合，例如，根据`Student`的`name`查找对应`Student`的`Map`。

### （1）List

Java集合的设计有几个特点：一是实现了接口和实现类相分离，例如，有序表的接口是`List`，具体的实现类有`ArrayList`，`LinkedList`等，二是支持泛型，我们可以限制在一个集合中只能放入同一种数据类型的元素

```
List<String> list = new ArrayList<>(); // 只能放入String类型
```

最后，Java访问集合总是通过统一的方式——迭代器（Iterator）来实现，它最明显的好处在于无需知道集合内部元素是按什么方式存储的。

由于Java的集合设计非常久远，中间经历过大规模改进，我们要注意到有一小部分集合类是遗留类，不应该继续使用：

- `Hashtable`：一种线程安全的`Map`实现；
- `Vector`：一种线程安全的`List`实现；
- `Stack`：基于`Vector`实现的`LIFO`的栈。

还有一小部分接口是遗留接口，也不应该继续使用：

- `Enumeration<E>`：已被`Iterator<E>`取代。

集合类中，`List`是最基础的一种集合：它是一种有序列表。

`List`的行为和数组几乎完全相同：`List`内部按照放入元素的先后顺序存放，每个元素都可以通过索引确定自己的位置，`List`的索引和数组一样，从`0`开始。

数组和`List`类似，也是有序结构，如果我们使用数组，在添加和删除元素的时候，会非常不方便。这个“删除”操作实际上是把`'C'`后面的元素依次往前挪一个位置，而“添加”操作实际上是把指定位置以后的元素都依次向后挪一个位置，腾出来的位置给新加的元素。这两种操作，用数组实现非常麻烦。

因此，在实际应用中，需要增删元素的有序列表，我们使用最多的是`ArrayList`。实际上，`ArrayList`在内部使用了数组来存储所有元素。例如，一个`ArrayList`拥有5个元素，实际数组大小为`6`（即有一个空位）。

`ArrayList`把添加和删除的操作封装起来，让我们操作`List`类似于操作数组，却不用关心内部元素如何移动。

我们考察`List<E>`接口，可以看到几个主要的接口方法：

- 在末尾添加一个元素：`boolean add(E e)`
- 在指定索引添加一个元素：`boolean add(int index, E e)`
- 删除指定索引的元素：`E remove(int index)`
- 删除某个元素：`boolean remove(Object e)`
- 获取指定索引的元素：`E get(int index)`
- 获取链表大小（包含元素的个数）：`int size()`

但是，实现`List`接口并非只能通过数组（即`ArrayList`的实现方式）来实现，另一种`LinkedList`通过“链表”也实现了List接口。在`LinkedList`中，它的内部每个元素都指向下一个元素

| ArrayList | LinkedList | |
|---|---|---|
| 获取指定元素 | 速度很快 | 需要从头开始查找元素 |
| 添加元素到末尾 | 速度很快 | 速度很快 |
| 在指定位置添加/删除 | 需要移动元素 | 不需要移动元素 |
| 内存占用 | 少 | 较大 |

## （2）Map

通过一个键去查询对应的值，使用`List`来实现存在效率非常低的问题，因为平均需要扫描一半的元素才能确定，而`Map`这种键值（key-value）映射表的数据结构，作用就是能高效通过`key`快速查找`value`（元素）。

用`Map`来实现根据`name`查询某个`Student`的代码如下：

```
1 public class Main {
```

```
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 99);
        Map<String, Student> map = new HashMap<>();
        map.put("Xiao Ming", s); // 将"Xiao Ming"和Student实例映射并关联
        Student target = map.get("Xiao Ming"); // 通过key查找并返回映射的Student
实例
        System.out.println(target == s); // true, 同一个实例
        System.out.println(target.score); // 99
        Student another = map.get("Bob"); // 通过另一个key查找
        System.out.println(another); // 未找到返回null
    }
}

class Student {
    public String name;
    public int score;
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}

```

通过上述代码可知：`Map<K, V>`是一种键-值映射表，当我们调用`put(K key, V value)`方法时，就把`key`和`value`做了映射并放入`Map`。当我们调用`V get(K key)`时，就可以通过`key`获取到对应的`value`。如果`key`不存在，则返回`null`。和`List`类似，`Map`也是一个接口，最常用的实现类是`HashMap`。

如果只是想查询某个`key`是否存在，可以调用`boolean containsKey(K key)`方法。

对`Map`来说，要遍历`key`可以使用`for each`循环遍历`Map`实例的`keySet()`方法返回的`Set`集合，它包含不重复的`key`的集合：

```
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (String key : map.keySet()) {
            Integer value = map.get(key);
            System.out.println(key + " = " + value);
        }
    }
}
```

同时遍历`key`和`value`可以使用`for each`循环遍历`Map`对象的`entrySet()`集合，它包含每一个`key-value`映射：

```
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
```

```
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " = " + value);
        }
    }
}
```

`HashMap`之所以能根据`key`直接拿到`value`，原因是它内部通过空间换时间的方法，用一个大数组存储所有`value`，并根据key直接计算出`value`应该存储在哪个索引。因为`HashMap`是一种通过对key计算`hashCode()`，通过空间换时间的方式，直接定位到value所在的内部数组的索引，因此，查找效率非常高。

如果作为key的对象是`enum`类型，那么，还可以使用Java集合库提供的一种`EnumMap`，它在内部以一个非常紧凑的数组存储value，并且根据`enum`类型的key直接定位到内部数组的索引，并不需要计算`hashCode()`，不但效率最高，而且没有额外的空间浪费。

我们以`DayOfWeek`这个枚举类型为例，为它做一个“翻译”功能：

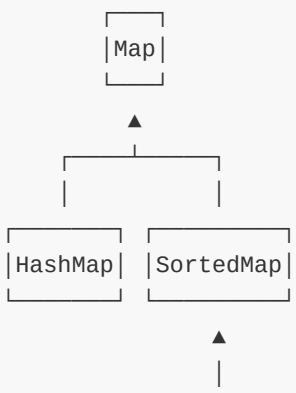
```
public class Main {
    public static void main(String[] args) {
        Map<DayOfWeek, String> map = new EnumMap<>(DayOfWeek.class);
        map.put(DayOfWeek.MONDAY, "星期一");
        map.put(DayOfWeek.TUESDAY, "星期二");
        map.put(DayOfWeek.WEDNESDAY, "星期三");
        map.put(DayOfWeek.THURSDAY, "星期四");
        map.put(DayOfWeek.FRIDAY, "星期五");
        map.put(DayOfWeek.SATURDAY, "星期六");
        map.put(DayOfWeek.SUNDAY, "星期日");
        System.out.println(map);
        System.out.println(map.get(DayOfWeek.MONDAY));
    }
}
```

如果`Map`的key是`enum`类型，推荐使用`EnumMap`，既保证速度，也不浪费空间。

使用`EnumMap`的时候，根据面向抽象编程的原则，应持有`Map`接口。

`HashMap`是一种以空间换时间的映射表，它的实现原理决定了内部的Key是无序的，即遍历`HashMap`的Key时，其顺序是不可预测的（但每个Key都会遍历一次且仅遍历一次）。

还有一种`Map`，它在内部会对Key进行排序，这种`Map`就是`SortedMap`。注意到`SortedMap`是接口，它的实现类是`TreeMap`。



```
            ┌─────────┐
            │ TreeMap │
            └─────────┘
```

`SortedMap`保证遍历时以Key的顺序来进行排序。例如，放入的Key是`"apple"`、`"pear"`、`"orange"`，遍历的顺序一定是`"apple"`、`"orange"`、`"pear"`，因为`String`默认按字母排序。

```
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new TreeMap<>();
        map.put("orange", 1);
        map.put("apple", 2);
        map.put("pear", 3);
        for (String key : map.keySet()) {
            System.out.println(key);
        }
        // apple, orange, pear
    }
}

```

使用`TreeMap`时，放入的Key必须实现`Comparable`接口。`String`、`Integer`这些类已经实现了`Comparable`接口，因此可以直接作为Key使用。作为Value的对象则没有任何要求。

如果作为Key的class没有实现`Comparable`接口，那么，必须在创建`TreeMap`时同时指定一个自定义排序算法：

```
public class Main {
    public static void main(String[] args) {
        Map<Person, Integer> map = new TreeMap<>(new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.name.compareTo(p2.name);
            }
        });
        map.put(new Person("Tom"), 1);
        map.put(new Person("Bob"), 2);
        map.put(new Person("Lily"), 3);
        for (Person key : map.keySet()) {
            System.out.println(key);
        }
        // {Person: Bob}, {Person: Lily}, {Person: Tom}
        System.out.println(map.get(new Person("Bob"))); // 2
    }
}

class Person {
    public String name;
    Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "{Person: " + name + "}";
    }
}

```

注意到`Comparator`接口要求实现一个比较方法，它负责比较传入的两个元素`a`和`b`，如果`a<b`，则返回负数，通常是`-1`，如果`a==b`，则返回`0`，如果`a>b`，则返回正数，通常是`1`。`TreeMap`内部根据比较结果对Key进行排序。

从上述代码执行结果可知，打印的Key确实是按照`Comparator`定义的顺序排序的。如果要根据Key查找Value，我们可以传入一个`new Person("Bob")`作为Key，它会返回对应的`Integer`值`2`。

另外，注意到`Person`类并未覆写`equals()`和`hashCode()`，因为`TreeMap`不使用`equals()`和`hashCode()`。

### （3）Properties

在编写应用程序的时候，经常需要读写配置文件。例如，用户的设置：

```
# 上次最后打开的文件:
last_open_file=/data/hello.txt
# 自动保存文件的时间间隔:
auto_save_interval=60

```

配置文件的特点是，它的Key-Value一般都是`String`-`String`类型的，因此我们完全可以用`Map<String, String>`来表示它。

因为配置文件非常常用，所以Java集合库提供了一个`Properties`来表示一组“配置”。由于历史遗留原因，`Properties`内部本质上是一个`Hashtable`，但我们只需要用到`Properties`自身关于读写配置的接口。

用`Properties`读取配置文件非常简单。Java默认配置文件以`.properties`为扩展名，每行以`key=value`表示，以`#`开头的是注释。

可以从文件系统读取这个`.properties`文件：

```
String f = "setting.properties";
Properties props = new Properties();
props.load(new java.io.FileInputStream(f));

String filepath = props.getProperty("last_open_file");
String interval = props.getProperty("auto_save_interval", "120");
```

可见，用`Properties`读取配置文件，一共有三步：

1. 创建`Properties`实例；
2. 调用`load()`读取文件；
3. 调用`getProperty()`获取配置。

调用`getProperty()`获取配置时，如果key不存在，将返回`null`。我们还可以提供一个默认值，这样，当key不存在的时候，就返回默认值。

也可以从classpath读取`.properties`文件，因为`load(InputStream)`方法接收一个`InputStream`实例，表示一个字节流，它不一定是文件流，也可以是从jar包中读取的资源流：

```
Properties props = new Properties();
props.load(getClass().getResourceAsStream("/common/setting.properties"));
```

果通过`setProperty()`修改了`Properties`实例，可以把配置写入文件，以便下次启动时获得最新配置。写入配置文件使用`store()`方法：

```
Properties props = new Properties();
props.setProperty("url", "http://www.liaoxuefeng.com");
props.setProperty("language", "Java");
props.store(new FileOutputStream("C:\\conf\\setting.properties"), "这是写入的properties注释");
```

## （4）Set

我们知道，`Map`用于存储key-value的映射，对于充当key的对象，是不能重复的，并且，不但需要正确覆写`equals()`方法，还要正确覆写`hashCode()`方法。

如果我们只需要存储不重复的key，并不需要存储映射的value，那么就可以使用`Set`。

`Set`用于存储不重复的元素集合，它主要提供以下几个方法：

- 将元素添加进`Set<E>`：`boolean add(E e)`
- 将元素从`Set<E>`删除：`boolean remove(Object e)`
- 判断是否包含元素：`boolean contains(Object e)`

```
public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        System.out.println(set.add("abc")); // true
        System.out.println(set.add("xyz")); // true
        System.out.println(set.add("xyz")); // false，添加失败，因为元素已存在
        System.out.println(set.contains("xyz")); // true，元素存在
        System.out.println(set.contains("XYZ")); // false，元素不存在
        System.out.println(set.remove("hello")); // false，删除失败，因为元素不存在
        System.out.println(set.size()); // 2，一共两个元素
    }
}

```

`Set`实际上相当于只存储key、不存储value的`Map`。我们经常用`Set`用于去除重复元素。

因为放入`Set`的元素和`Map`的key类似，都要正确实现`equals()`和`hashCode()`方法，否则该元素无法正确地放入`Set`。

最常用的`Set`实现类是`HashSet`，实际上，`HashSet`仅仅是对`HashMap`的一个简单封装，它的核心代码如下：

```
public class HashSet<E> implements Set<E> {
    // 持有一个HashMap:
    private HashMap<E, Object> map = new HashMap<>();

    // 放入HashMap的value:
    private static final Object PRESENT = new Object();

    public boolean add(E e) {
        return map.put(e, PRESENT) == null;
    }

```

```
    public boolean contains(Object o) {
        return map.containsKey(o);
    }

    public boolean remove(Object o) {
        return map.remove(o) == PRESENT;
    }
}
```

`Set`接口并不保证有序，而`SortedSet`接口则保证元素是有序的：

- `HashSet`是无序的，因为它实现了`Set`接口，并没有实现`SortedSet`接口；
- `TreeSet`是有序的，因为它实现了`SortedSet`接口。

用一张图表示：


把`HashSet`换成`TreeSet`，在遍历`TreeSet`时，输出就是有序的，这个顺序是元素的排序顺序：

```
public class Main {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

`Set`用于存储不重复的元素集合：

- 放入`HashSet`的元素与作为`HashMap`的key要求相同；
- 放入`TreeSet`的元素与作为`TreeMap`的Key要求相同；

利用`Set`可以去除重复元素；

遍历`SortedSet`按照元素的排序顺序遍历，也可以自定义排序算法。

## （5）Queue

队列（`Queue`）是一种经常使用的集合。`Queue`实际上是实现了一个先进先出（FIFO：First In First Out）的有序表。它和`List`的区别在于，`List`可以在任意位置添加和删除元素，而`Queue`只有两个操作：

- 把元素添加到队列末尾；
- 从队列头部取出元素。

在Java的标准库中，队列接口`Queue`定义了以下几个方法：

- `int size()`：获取队列长度；
- `boolean add(E)`/`boolean offer(E)`：添加元素到队尾；
- `E remove()`/`E poll()`：获取队首元素并从队列中删除；
- `E element()`/`E peek()`：获取队首元素但并不从队列中删除。

对于具体的实现类，有的Queue有最大队列长度限制，有的Queue没有。注意到添加、删除和获取队列元素总是有两个方法，这是因为在添加或获取元素失败时，这两个方法的行为是不同的。我们用一个表格总结如下：

| | throw Exception | 返回false或null |
|---|---|---|
| 添加元素到队尾 | add(E e) | boolean offer(E e) |
| 取队首元素并删除 | E remove() | E poll() |
| 取队首元素但不删除 | E element() | E peek() |

```
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 从队列取出元素:
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // null,因为队列是空的
    }
}
```

`LinkedList`即实现了`List`接口，又实现了`Queue`接口，但是，在使用的时候，如果我们把它当作List，就获取List的引用，如果我们把它当作Queue，就获取Queue的引用：

```
// 这是一个List:
List<String> list = new LinkedList<>();
// 这是一个Queue:
Queue<String> queue = new LinkedList<>();
```

## （6）PriorityQueue

`PriorityQueue`和`Queue`的区别在于，它的出队顺序与元素的优先级有关，对`PriorityQueue`调用`remove()`或`poll()`方法，返回的总是优先级最高的元素。

要使用`PriorityQueue`，我们就必须给每个元素定义“优先级”。我们以实际代码为例，先看看`PriorityQueue`的行为：

```
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new PriorityQueue<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // null,因为队列为空
    }
}

```

放入`PriorityQueue`的元素，必须实现`Comparable`接口，`PriorityQueue`会根据元素的排序顺序决定出队的优先级。

如果我们要放入的元素并没有实现`Comparable`接口怎么办？`PriorityQueue`允许我们提供一个`Comparator`对象来判断两个元素的顺序。我们以银行排队业务为例，实现一个`PriorityQueue`：

```
public class Main {
    public static void main(String[] args) {
        Queue<User> q = new PriorityQueue<>(new UserComparator());
        // 添加3个元素到队列:
        q.offer(new User("Bob", "A1"));
        q.offer(new User("Alice", "A2"));
        q.offer(new User("Boss", "V1"));
        System.out.println(q.poll()); // Boss/V1
        System.out.println(q.poll()); // Bob/A1
        System.out.println(q.poll()); // Alice/A2
        System.out.println(q.poll()); // null,因为队列为空
    }
}

class UserComparator implements Comparator<User> {
    public int compare(User u1, User u2) {
        if (u1.number.charAt(0) == u2.number.charAt(0)) {
            // 如果两人的号都是A开头或者都是V开头,比较号的大小:
            return u1.number.compareTo(u2.number);
        }
        if (u1.number.charAt(0) == 'V') {
            // u1的号码是V开头,优先级高:
            return -1;
        } else {
            return 1;
        }
    }

```

```
}

class User {
    public final String name;
    public final String number;

    public User(String name, String number) {
        this.name = name;
        this.number = number;
    }

    public String toString() {
        return name + "/" + number;
    }
}

```

`PriorityQueue`实现了一个优先队列：从队首获取元素时，总是获取优先级最高的元素。

`PriorityQueue`默认按元素比较的顺序排序（必须实现`Comparable`接口），也可以通过`Comparator`自定义排序算法（元素就不必实现`Comparable`接口）。

### （7）双端队列

我们知道，`Queue`是队列，只能一头进，另一头出。

如果把条件放松一下，允许两头都进，两头都出，这种队列叫双端队列（Double Ended Queue），学名`Deque`。

Java集合提供了接口`Deque`来实现一个双端队列，它的功能是：

- 既可以添加到队尾，也可以添加到队首；
- 既可以从队首获取，又可以从队尾获取。

我们来比较一下`Queue`和`Deque`出队和入队的方法：

| | Queue | Deque |
|---|---|---|
| 添加元素到队尾 | add(E e) / offer(E e) | addLast(E e) / offerLast(E e) |
| 取队首元素并删除 | E remove() / E poll() | E removeFirst() / E pollFirst() |
| 取队首元素但不删除 | E element() / E peek() | E getFirst() / E peekFirst() |
| 添加元素到队首 | 无 | addFirst(E e) / offerFirst(E e) |
| 取队尾元素并删除 | 无 | E removeLast() / E pollLast() |
| 取队尾元素但不删除 | 无 | E getLast() / E peekLast() |

```
public class Main {
    public static void main(String[] args) {
        Deque<String> deque = new LinkedList<>();
        deque.offerLast("A"); // A
        deque.offerLast("B"); // A <- B
        deque.offerFirst("C"); // C <- A <- B
        System.out.println(deque.pollFirst()); // C, 剩下A <- B
        System.out.println(deque.pollLast()); // B, 剩下A
        System.out.println(deque.pollFirst()); // A
        System.out.println(deque.pollFirst()); // null
    }
}
```

`Deque`实现了一个双端队列（Double Ended Queue），它可以：

- 将元素添加到队尾或队首：`addLast()`/`offerLast()`/`addFirst()`/`offerFirst()`；
- 从队首／队尾获取元素并删除：`removeFirst()`/`pollFirst()`/`removeLast()`/`pollLast()`；
- 从队首／队尾获取元素但不删除：`getFirst()`/`peekFirst()`/`getLast()`/`peekLast()`；
- 总是调用`xxxFirst()`/`xxxLast()`以便与`Queue`的方法区分开；
- 避免把`null`添加到队列。

## （8）栈

栈（Stack）是一种后进先出（LIFO：Last In First Out）的数据结构。`Stack`只有入栈和出栈的操作：

- 把元素压栈：`push(E)`；
- 把栈顶的元素“弹出”：`pop()`；
- 取栈顶元素但不弹出：`peek()`。

在Java中，我们用`Deque`可以实现`Stack`的功能：

- 把元素压栈：`push(E)`/`addFirst(E)`；
- 把栈顶的元素“弹出”：`pop()`/`removeFirst()`；
- 取栈顶元素但不弹出：`peek()`/`peekFirst()`。

为什么Java的集合类没有单独的`Stack`接口呢？因为有个遗留类名字就叫`Stack`，出于兼容性考虑，所以没办法创建`Stack`接口，只能用`Deque`接口来“模拟”一个`Stack`了。

当我们把`Deque`作为`Stack`使用时，注意只调用`push()`/`pop()`/`peek()`方法，不要调用`addFirst()`/`removeFirst()`/`peekFirst()`方法，这样代码更加清晰。

## （9）Iterator

Java的集合类都可以使用`for each`循环，`List`、`Set`和`Queue`会迭代每个元素，`Map`会迭代每个key。以`List`为例：

```
List<String> list = List.of("Apple", "Orange", "Pear");
for (String s : list) {
    System.out.println(s);
}
```

实际上，Java编译器并不知道如何遍历`List`。上述代码能够编译通过，只是因为编译器把`for each`循环通过`Iterator`改写为了普通的`for`循环：

```java
for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
     String s = it.next();
     System.out.println(s);
}
```

我们把这种通过`Iterator`对象遍历集合的模式称为迭代器。

使用迭代器的好处在于，调用方总是以统一的方式遍历各种集合类型，而不必关系它们内部的存储结构。

例如，我们虽然知道`ArrayList`在内部是以数组形式存储元素，并且，它还提供了`get(int)`方法。虽然我们可以用`for`循环遍历：

```java
for (int i=0; i<list.size(); i++) {
    Object value = list.get(i);
}
```

但是这样一来，调用方就必须知道集合的内部存储结构。并且，如果把`ArrayList`换成`LinkedList`，`get(int)`方法耗时会随着index的增加而增加。如果把`ArrayList`换成`Set`，上述代码就无法编译，因为`Set`内部没有索引。

用`Iterator`遍历就没有上述问题，因为`Iterator`对象是集合对象自己在内部创建的，它自己知道如何高效遍历内部的数据集合，调用方则获得了统一的代码，编译器才能把标准的`for each`循环自动转换为`Iterator`遍历。

如果我们自己编写了一个集合类，想要使用`for each`循环，只需满足以下条件：

- 集合类实现`Iterable`接口，该接口要求返回一个`Iterator`对象；
- 用`Iterator`对象迭代集合内部数据。

这里的关键在于，集合类通过调用`iterator()`方法，返回一个`Iterator`对象，这个对象必须自己知道如何遍历该集合。

```java

import java.util.*;

public class Main {
    public static void main(String[] args) {
        ReverseList<String> rlist = new ReverseList<>();
        rlist.add("Apple");
        rlist.add("Orange");
        rlist.add("Pear");
        for (String s : rlist) {
            System.out.println(s);
        }
    }
}

class ReverseList<T> implements Iterable<T> {

    private List<T> list = new ArrayList<>();

    public void add(T t) {
        list.add(t);
    }

```

```
    @Override
    public Iterator<T> iterator() {
        return new ReverseIterator(list.size());
    }

    class ReverseIterator implements Iterator<T> {
        int index;

        ReverseIterator(int index) {
            this.index = index;
        }

        @Override
        public boolean hasNext() {
            return index > 0;
        }

        @Override
        public T next() {
            index--;
            return ReverseList.this.list.get(index);
        }
    }
}

```

`Iterator`是一种抽象的数据访问模型。使用`Iterator`模式进行迭代的好处有：

- 对任何集合都采用同一种访问模型；
- 调用者对集合内部结构一无所知；
- 集合类返回的`Iterator`对象知道如何迭代。

Java提供了标准的迭代器模型，即集合类实现`java.util.Iterable`接口，返回`java.util.Iterator`实例。

## （10）Collections

`Collections`是JDK提供的工具类，同样位于`java.util`包中。它提供了一系列静态方法，能更方便地操作各种集合。

`Collections`提供了一系列方法来创建空集合：

- 创建空List：`List<T> emptyList()`
- 创建空Map：`Map<K, V> emptyMap()`
- 创建空Set：`Set<T> emptySet()`

要注意到返回的空集合是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T...)`方法创建空集合。例如，以下创建空`List`的两个方法是等价的：

```
List<String> list1 = List.of();
List<String> list2 = Collections.emptyList();
```

`Collections`类提供了一组工具方法来方便使用集合类：

- 创建空集合；
- 创建单元素集合；

- 创建不可变集合；
- 排序／洗牌等操作。

# 三、C与C++基础

# 四、Javascript、CSS和html

学习web前端开发基础技术需要掌握：HTML、CSS、JavaScript语言。下面我们就来了解下这三门技术都是用来实现什么的：

1. HTML是网页内容的载体。内容就是网页制作者放在页面上想要让用户浏览的信息，可以包含文字、图片、视频等。
2. CSS样式是表现。就像网页的外衣。比如，标题字体、颜色变化，或为标题加入背景图片、边框等。所有这些用来改变内容外观的东西称之为表现。
3. JavaScript是用来实现网页上的特效效果。如：鼠标滑过弹出下拉菜单。或鼠标滑过表格的背景颜色改变。还有焦点新闻（新闻图片）的轮换。可以这么理解，有动画的，有交互的一般都是用JavaScript来实现的。

## 1、Javascript

略

## 2、CSS

### （1）CSS盒模型

所有HTML元素可以看作盒子，在CSS中，"box model"这一术语是用来设计和布局时使用。

CSS盒模型本质上是一个盒子，封装周围的HTML元素，它包括：边距，边框，填充，和实际内容。

盒模型允许我们在其它元素和周围元素边框之间的空间放置元素。

下面的图片说明了盒子模型(Box Model)：


不同部分的说明：

- **Margin(外边距)** - 清除边框外的区域，外边距是透明的。
- **Border(边框)** - 围绕在内边距和内容外的边框。
- **Padding(内边距)** - 清除内容周围的区域，内边距是透明的。

- **Content(内容)** - 盒子的内容，显示文本和图像。

**重要:** 当您指定一个 CSS 元素的宽度和高度属性时，你只是设置内容区域的宽度和高度。要知道，完整大小的元素，你还必须添加内边距，边框和外边距。

下面的例子中的元素的总宽度为300px：

```
div {
    width: 300px;
    border: 25px solid green;
    padding: 25px;
    margin: 25px;
}
```

让我们自己算算：
300px (宽)
+ 50px (左 + 右填充)
+ 50px (左 + 右边框)
+ 50px (左 + 右边距)
= 450px

试想一下，你只有 250 像素的空间。让我们设置总宽度为 250 像素的元素:

```
div {
    width: 220px;
    padding: 10px;
    border: 5px solid gray;
    margin: 0;
}
```

最终元素的总宽度计算公式是这样的：

总元素的宽度=宽度+左填充+右填充+左边框+右边框+左边距+右边距

元素的总高度最终计算公式是这样的：

总元素的高度=高度+顶部填充+底部填充+上边框+下边框+上边距+下边距

# 3、html

超文本标记语言（英语：HyperText Markup Language，简称：HTML）是一种用于创建网页的标准标记语言。

您可以使用 HTML 来建立自己的 WEB 站点，HTML 运行在浏览器上，由浏览器来解析。

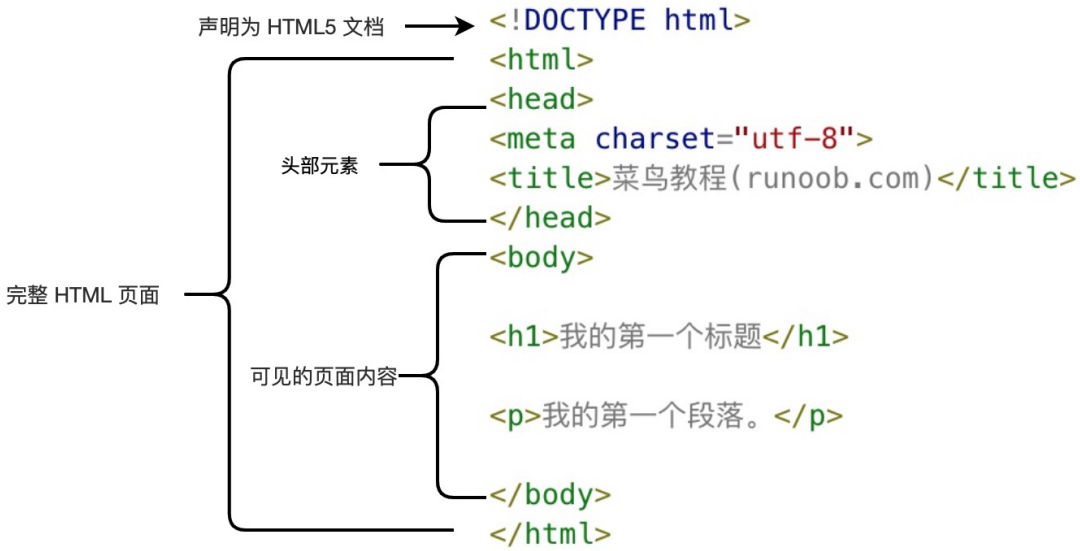
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>

<h1>我的第一个标题</h1>

<p>我的第一个段落。</p>

```

```
</body>
</html>
```




HTML 是用来描述网页的一种语言。

- HTML 指的是超文本标记语言: **H**yper**T**ext **M**arkup **L**anguage
- HTML 不是一种编程语言，而是一种**标记**语言
- 标记语言是一套**标记标签** (markup tag)
- HTML 使用标记标签来**描述**网页
- HTML 文档包含了HTML **标签**及**文本**内容
- HTML文档也叫做 **web 页面**

HTML 标记标签通常被称为 HTML 标签 (HTML tag)。

- HTML 标签是由*尖括号*包围的关键词，比如
- HTML 标签通常是*成对出现*的，比如 **和**
- 标签对中的第一个标签是*开始标签*，第二个标签是*结束标签*
- 开始和结束标签也被称为*开放标签*和*闭合标签*

HTML 标题（Heading）是通过

---

标签来定义的。HTML 段落是通过标签

来定义的。HTML 链接是通过标签 来定义的。HTML 图像是通过标签 来定义的。

表格示例：

```
<table border="1">
    <tr>
        <td>row 1, cell 1</td>
        <td>row 1, cell 2</td>
    </tr>
    <tr>
        <td>row 2, cell 1</td>
        <td>row 2, cell 2</td>
    </tr>
</table>
```

div 元素是用于分组 HTML 元素的块级元素。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>

<div id="container" style="width:500px">

<div id="header" style="background-color:#FFA500;">
<h1 style="margin-bottom:0;">主要的网页标题</h1></div>

<div id="menu" style="background-
color:#FFD700;height:200px;width:100px;float:left;">
<b>菜单</b><br>
HTML<br>
CSS<br>
JavaScript</div>

<div id="content" style="background-
color:#EEEEEE;height:200px;width:400px;float:left;">
内容在这里</div>

<div id="footer" style="background-color:#FFA500;clear:both;text-
align:center;">
版权 © runoob.com</div>

</div>

</body>
</html>
```

使用 HTML

标签是创建布局的一种简单的方式。

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
```

```
<table width="500" border="0">
<tr>
<td colspan="2" style="background-color:#FFA500;">
<h1>主要的网页标题</h1>
</td>
</tr>

<tr>
<td style="background-color:#FFD700;width:100px;">
<b>菜单</b><br>
HTML<br>
CSS<br>
JavaScript
</td>
<td style="background-color:#eeeeee;height:200px;width:400px;">
内容在这里</td>
</tr>

<tr>
<td colspan="2" style="background-color:#FFA500;text-align:center;">
版权 © runoob.com</td>
</tr>
</table>

</body>
</html>
```

文本域通过 <input type="text"> 标签来设定，当用户要在表单中键入字母、数字等内容时，就会用到文本域。

```
<form>
First name: <input type="text" name="firstname"><br>
Last name: <input type="text" name="lastname">
</form>
```

密码字段通过标签 <input type="password"> 来定义：

```
<form>
Password: <input type="password" name="pwd">
</form>
```

# 五、软件测试方法

测试名称：黑盒测试（Black Box） 测试内容：黑盒测试是把测试对象看做一个黑盒子，利用黑盒测试法进行动态测试时，需要测试软件产品已经实现的功能是否符合功能设计要求，不需测试软件产品的内部结构和处理过程。 黑盒测试注重于测试软件的功能性需求，也即黑盒测试使软件工程师派生出执行程序所有功能需求的输入条件。黑盒测试并不是白盒测试的替代品，而是用于辅助白盒测试发现其他类型的错误。 黑盒测试更多内容请查看之前的一篇博文:黑盒测试用例设计

测试名称：白盒测试（White Box） 测试内容：设计者可以看到软件系统的内部结构，并且使用软件的内部知识来指导测试数据及方法的选择。 白盒测试通常被认为是单元测试与集成测试，期中有六种测试方法：语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖。 更多白盒测试的内容请查看此处链接

测试名称：灰盒测试（Gray Box） 测试内容：介于黑盒和白盒之间，是一种综合测试的方法，他将白盒测试和黑盒测试结合在一起，构成一种无缝测试技术。 灰盒测试是基于程序运行时的外部表现又结合程序内部逻辑结构来设计测试用例，执行程序并采集程序路径执行信息和外部用户接口结果的测试技术。灰盒测试法旨在验证软件满足外部指标以及软件的所有通道或路径都进行了检验。

单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试。

集成测试：在运行（可能是不完整）的应用中保证软件单元被结合后能正常操作的测试执行的阶段

系统测试：当应用作为整体运行时的测试执行阶段

黑盒测试 (Black box testing) —— 不考虑内部设计和代码，根据需求和功能进行测试。

白盒测试 (White box testing) —— 根据应用软件的代码的内部逻辑，按照代码的语句、分支、路径和条件进行测试。

功能测试（functional testing）——对一个应用软件的功能模块进行黑盒测试。这种测试应当由测试人员进行。但这并不意味着程序员在推出软件之前不进行代码检查。（这一原则适用于所有的测试阶段。）

系统测试 —— 针对全部需求说明进行黑盒测试，包括系统中所有的部件。

回归测试 (regression testing) —— 每当软件经过了整理、修改、或者其环境发生变化，都重复进行测试。很难说需要进行多少次回归测试，特别是是到了开发周期的最后阶段。进行此种测试，特别适于使用自动测试工具。

负荷试验 (load testing) —— 在大负荷条件下对应用软件进行测试。例如测试一个网站在不同负荷情况下的状况，以确定在什么情况下系统响应速度下降或是出现故障。

压力测试 (stress testing) —— 经常可以与"负荷测试"或"性能测试"相互代替。这种测试是用来检查系统在下列条件下的情况：在非正常的巨大负荷下、某些动作和输入大量重复、输入大数、对数据库进行非常复杂的查询，等等。

性能测试 (performance testing) —— 经常可以与"压力测试"或"负荷测试"相互代替。理想的"性能测试"(也包括其他任何类型的测试) 都应在质量保障和测试计划的文档终予以规定。

可用性测试 (usability testing) —— 是专为"对用户友好"的特性进行测试。这是一种主观的感觉，取决于最终用户或顾客。可以进行用户会见、检查、对用户会议录像、或者使用其他技术。程序员和测试人员通常不参加可用性测试。

安装/卸载测试 (install/uninstall testing) —— 对安装/卸载进行测试 (包括全部、部分、升级操作)。

安全测试 (security testing) —— 测试系统在应付非授权的内部/外部访问、故意的损坏时的防护情况。这需要精密复杂的测试技术。

兼容性测试 (compatability testing) —— 测试在特殊的硬件/软件/操作系统/网络环境下的软件表现。

α 测试 (alpha testing) —— 在开发一个应用软件即将完成时所进行的测试。此时还允许有较小的设计修改。通常由最终用户或其他人进行这种测试，而不是由程序员和测试人员来进行。

β 测试 (beta testing) —— 当开发和测试已基本完成，需要在正式发行之前最后寻找毛病而进行的测试。通常由最终用户或其他人进行这种测试，而不是由程序员和测试人员来进行。

# 六、Linux常用命令

# 七、关系型数据库基础

# 八、中信银行开发规范

## 九、中信银行安全技术规范

## 十、中信银行设计规范

## 十一、中信银行发展战略、企业文化及合规内控要求