

课程目标

能够熟练使用Math类中的常见方法

能够熟练使用System类中的常见方法

能够理解Object类的常见方法作用

能够熟练使用Objects类的常见方法

能够熟练使用BigInteger类的常见方法

能够熟练使用BigDecimal类的常见方法

1 Math类

1.1 概述

tips: 了解内容

查看API文档，我们可以看到API文档中关于Math类的定义如下：

```
java.lang  
类 Math  
  
java.lang.Object  
└─ java.lang.Math
```

```
public final class Math  
extends Object
```

Math类所在包为java.lang包，因此在使用的时候不需要进行导包。并且Math类被final修饰了，因此该类是不能被继承的。

Math类包含执行基本数字运算的方法，我们可以使用Math类完成基本的数学运算。

要想使用Math类我们就需要先创建该类的对象，那么创建对象就需要借助于构造方法。因此我们就需要首先查看一下API文档，看看API文档中针对Math类有没有提供对应的构造方法。通过API文档来查看

一下Math类的成员，如下所示：

字段摘要

static double	<u>E</u>	比任何其他值都更接近 e (即自然对数的底数) 的 double 值。
static double	<u>PI</u>	比任何其他值都更接近 π (即圆的周长与直径之比) 的 double 值。

方法摘要

static double	<u>abs</u> (double a)	返回 double 值的绝对值。
static float	<u>abs</u> (float a)	返回 float 值的绝对值。
static int	<u>abs</u> (int a)	返回 int 值的绝对值。
static long	<u>abs</u> (long a)	返回 long 值的绝对值。
static double	<u>acos</u> (double a)	返回一个值的反余弦; 返回的角度范围在 0.0 到 π 之间。
static double	<u>asin</u> (double a)	返回一个值的反正弦; 返回的角度范围在 $-\pi/2$ 到 $\pi/2$ 之间。
static double	<u>atan</u> (double a)	返回一个值的反正切; 返回的角度范围在 $-\pi/2$ 到 $\pi/2$ 之间。
static double	<u>atan2</u> (double y, double x)	将矩形坐标 (x, y) 转换成极坐标 (r, <i>theta</i>), 返回所得角 <i>theta</i> 。
static double	<u>cbrt</u> (double a)	返回 double 值的立方根。
static double	<u>ceil</u> (double a)	返回最小的 (最接近负无穷大) double 值, 该值大于等于参数, 并等于某个整数。
static double	<u>copySign</u> (double magnitude, double sign)	返回带有第二个浮点参数符号的第一个浮点参数。
static float	<u>copySign</u> (float magnitude, float sign)	返回带有第二个浮点参数符号的第一个浮点参数。
static double	<u>cos</u> (double a)	返回角的三角余弦。
static double	<u>cosh</u> (double x)	返回 double 值的双曲线余弦。
static double	<u>exp</u> (double a)	返回欧拉数 e 的 double 次幂的值。
static double	<u>expm1</u> (double x)	返回 $e^x - 1$ 。

在API文档中没有体现可用的构造方法, 因此我们就不能直接通过new关键字去创建Math类的对象。同时我们发现Math类中的方法都是静态的, 因此在使用的时候我们可以直接通过类名去调用。在Math类中

定义了很多数学运算的方法, 但是我们不可能将所有的方法学习一遍, 我们主要学习的就是一些常见的方法。

1.2 常见方法

tips: 重点讲解内容

常见方法介绍

我们要学习的Math的常见方法如下所示：

<code>public static int abs(int a)</code>	// 返回参数的绝对值
<code>public static double ceil(double a)</code>	// 返回大于或等于参数的最小整数
<code>public static double floor(double a)</code>	// 返回小于或等于参数的最大整数
<code>public static int round(float a)</code>	// 按照四舍五入返回最接近参数的
<code>int</code> 类型的值	
<code>public static int max(int a,int b)</code>	// 获取两个int值中的较大值
<code>public static int min(int a,int b)</code>	// 获取两个int值中的较小值
<code>public static double pow (double a,double b)</code>	// 计算a的b次幂的值
<code>public static double random()</code>	// 返回一个[0.0,1.0)的随机值

案例演示

接下来我们就来演示一些这些方法的执行效果，如下所示：

```
public class MathDemo01 {  
  
    public static void main(String[] args) {  
  
        // public static int abs(int a)          返回参数的绝对值  
        System.out.println("-2的绝对值为: " + Math.abs(-2));  
        System.out.println("2的绝对值为: " + Math.abs(2));  
  
        // public static double ceil(double a)  返回大于或等于参数的最小整数  
        System.out.println("大于或等于23.45的最小整数位: " + Math.ceil(23.45));  
        System.out.println("大于或等于-23.45的最小整数位: " + Math.ceil(-23.45));  
  
        // public static double floor(double a) 返回小于或等于参数的最大整数  
        System.out.println("小于或等于23.45的最大整数位: " + Math.floor(23.45));  
        System.out.println("小于或等于-23.45的最大整数位: " +  
Math.floor(-23.45));  
  
        // public static int round(float a)      按照四舍五入返回最接近参数的int  
        System.out.println("23.45四舍五入的结果为: " + Math.round(23.45));  
        System.out.println("23.55四舍五入的结果为: " + Math.round(23.55));  
  
        // public static int max(int a,int b)    返回两个int值中的较大值  
        System.out.println("23和45的最大值为: " + Math.max(23, 45));  
  
        // public static int min(int a,int b)    返回两个int值中的较小值  
        System.out.println("12和34的最小值为: " + Math.min(12, 34));  
  
        // public static double pow (double a,double b)返回a的b次幂的值  
        System.out.println("2的3次幂计算结果为: " + Math.pow(2,3));  
  
        // public static double random()返回值为double的正值, [0.0,1.0)
```

```

        System.out.println("获取到的0-1之间的随机数为: " + Math.random());
    }

}

```

运行程序进行测试，控制台输出结果如下：

```

-2的绝对值为: 2
2的绝对值为: 2
大于或等于23.45的最小整数位: 24.0
大于或等于-23.45的最小整数位: -23.0
小于或等于23.45的最大整数位: 23.0
小于或等于-23.45的最大整数位: -24.0
23.45四舍五入的结果为: 23
23.55四舍五入的结果为: 24
23和45的最大值为: 45
12和34的最小值为: 12
2的3次幂计算结果为: 8.0
获取到的0-1之间的随机数为: 0.7322484131745958

```

1.3 算法小题(质数)

需求：

判断一个数是否为一个质数

代码实现：

```

public class MathDemo2 {
    public static void main(String[] args) {
        //判断一个数是否为一个质数
        System.out.println(isPrime(997));
        //997 2~996 995次
    }

    public static boolean isPrime(int number) {
        int count = 0;
        for (int i = 2; i <= Math.sqrt(number); i++) {
            count++;
            if (number % i == 0) {
                return false;
            }
        }
        System.out.println(count);
        return true;
    }
}

```

1.4 算法小题(自幂数)

自幂数，一个n位自然数等于自身各个数位上数字的n次幂之和

举例1：三位数 $1^3 + 5^3 + 3^3 = 153$

举例2：四位数 $1^4 + 6^4 + 3^4 + 4^3 = 1634$

如果自幂数是：

- 一位自幂数，也叫做：独身数
- 三位自幂数：水仙花数 四位自幂数：四叶玫瑰数
- 五位自幂数：五角星数 六位自幂数：六合数
- 七位自幂数：北斗七星数 八位自幂数：八仙数
- 九位自幂数：九九重阳数 十位自幂数：十全十美数

要求1：统计一共有多少个水仙花数。

要求2：（课后作业）证明没有两位的自幂数。

要求3：（课后作业）分别统计有多少个四叶玫瑰数和五角星数。（答案：都是3个）

```
//水仙花数:100 ~ 999
int count = 0;
//得到每一个三位数
for (int i = 100; i <= 999; i++) {
    //个位 十位 百位
    int ge = i % 10;
    int shi = i / 10 % 10;
    int bai = i / 100 % 10;
    //判断：
    //每一位的三次方之和 跟本身 进行比较。
    double sum = Math.pow(ge, 3) + Math.pow(shi, 3) + Math.pow(bai, 3);
    if (sum == i) {
        count++;
        //System.out.println(i);

        System.out.println(count);
    }
}
```

1.5 课后练习

要求2：（课后作业）证明没有两位的自幂数。

要求3：（课后作业）分别统计有多少个四叶玫瑰数和五角星数。（答案：都是3个）

2 System类

2.1 概述

tips: 了解内容

查看API文档，我们可以看到API文档中关于System类的定义如下：

java.lang
类 System

[java.lang.Object](#)
└ **java.lang.System**

```
public final class System
extends Object
```

System类所在包为java.lang包，因此在使用的时候不需要进行导包。并且System类被final修饰了，因此该类是不能被继承的。

System包含了系统操作的一些常用的方法。比如获取当前时间所对应的毫秒值，再比如终止当前JVM等等。

要想使用System类我们就需要先创建该类的对象，那么创建对象就需要借助于构造方法。因此我们就需要首先查看一下API文档，看看API文档中针对System类有没有提供对应的构造方法。通过API文档来

查看一下System类的成员，如下所示：

Nested Class Summary

Nested Classes		
Modifier and Type	Class	描述
static interface	System.Logger	System.Logger实例日志消息将路由到底层日志框架LoggerFinder使用。
static class	System.LoggerFinder	LoggerFinder服务负责创建，管理和配置记录器到其使用的基础框架。

Field Summary

Fields		
Modifier and Type	Field	描述
static PrintStream	err	"标准"错误输出流。
static InputStream	in	"标准"输入流。
static PrintStream	out	"标准"输出流。

方法摘要

所有方法		
Modifier and Type	方法	描述
static void	arraycopy(Object src, int srcPos, Object dest, int destPos, int length)	将指定源数组中的数组从指定位置复制到目标数组的指定位置。
static String	clearProperty(String key)	删除指定键指定的系统属性。

在API文档中没有体现可用的构造方法，因此我们就不能直接通过new关键字去创建System类的对象。同时我们发现System类中的方法都是静态的，因此在使用的时候我们可以直接通过类名去调用（Nested Class Summary内部类或者内部接口的描述）。

2.2 常见方法

常见方法介绍

我们要学习的System类中的常见方法如下所示:

```
public static long currentTimeMillis()           // 获取当前时间所对应的毫秒值 (当前时间为0时区所对应的时间即就是英国格林尼治天文台旧址所在位置)
public static void exit(int status)              // 终止当前正在运行的Java虚拟机, 0表示正常退出, 非零表示异常退出
public static native void arraycopy(Object src,  int  srcPos, Object dest, int destPos, int length); // 进行数值元素copy
```

案例演示

接下来我们就来通过一些案例演示一下这些方法的特点。

案例1: 演示currentTimeMillis方法

```
public class SystemDemo01 {

    public static void main(String[] args) {

        // 获取当前时间所对应的毫秒值
        long millis = System.currentTimeMillis();

        // 输出结果
        System.out.println("当前时间所对应的毫秒值为: " + millis);

    }

}
```

运行程序进行测试, 控制台的输出结果如下:

当前时间所对应的毫秒值为: 1576050298343

获取到当前时间的毫秒值的意义: 我们常常来需要统计某一段代码的执行时间。此时我们就可以在执行这段代码之前获取一次时间, 在执行完毕以后再次获取一次系统时间, 然后计算两个时间的差值,

这个差值就是这段代码执行完毕以后所需要的时间。如下代码所示:

```
public class SystemDemo2 {
    public static void main(String[] args) {
        //判断1~100000之间有多少个质数
    }
}
```

```

        long start = System.currentTimeMillis();

        for (int i = 1; i <= 100000; i++) {
            boolean flag = isPrime2(i);
            if (flag) {
                System.out.println(i);
            }
        }
        long end = System.currentTimeMillis();
        //获取程序运行的总时间
        System.out.println(end - start); //方式一： 1514 毫秒 方式二： 71毫秒
    }

    //以前判断是否为质数的方式
    public static boolean isPrime1(int number) {
        for (int i = 2; i < number; i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }

    //改进之后判断是否为质数的方式（效率高）
    public static boolean isPrime2(int number) {
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }
}

```

案例2：演示exit方法

```

public class SystemDemo01 {

    public static void main(String[] args) {

        // 输出
        System.out.println("程序开始执行了.....");

        // 终止JVM
        System.exit(0);

        // 输出
        System.out.println("程序终止了.....");

    }
}

```



```
}
```

运行程序进行测试，控制台输出结果如下：

```
程序开始执行了.....
```

此时可以看到在控制台只输出了"程序开始了..."，由于JVM终止了，因此输出"程序终止了..."这段代码没有被执行。

案例3：演示arraycopy方法

方法参数说明：

```
// src:      源数组
// srcPos:   源数值的开始位置
// dest:     目标数组
// destPos:  目标数组开始位置
// length:   要复制的元素个数
public static native void arraycopy(Object src,  int  srcPos, Object dest, int
destPos, int length);
```

代码如下所示：

```
public class SystemDemo01 {

    public static void main(String[] args) {

        // 定义源数组
        int[] srcArray = {23 , 45 , 67 , 89 , 14 , 56 } ;

        // 定义目标数组
        int[] desArray = new int[10] ;

        // 进行数组元素的copy：把srcArray数组中从0索引开始的3个元素，从desArray数组
        中的1索引开始复制过去
        System.arraycopy(srcArray , 0 , desArray , 1 , 3);

        // 遍历目标数组
        for(int x = 0 ; x < desArray.length ; x++) {
            if(x != desArray.length - 1) {
                System.out.print(desArray[x] + ", ");
            }else {
                System.out.println(desArray[x]);
            }
        }
    }
}
```

```

    }

}
}

```

运行程序进行测试，控制台输出结果如下所示：

```
0, 23, 45, 67, 0, 0, 0, 0, 0, 0
```

通过控制台输出结果我们可以看到，数组元素的确进行复制了。

使用这个方法我们也可以完成数组元素的删除操作，如下所示：

```

public class SystemDemo02 {
    public static void main(String[] args) {
        // 定义一个数组
        int[] srcArray = {23, 45, 67, 89, 14, 56};
        // 删除数组中第3个元素(67)：要删除67这个元素，我们只需要将67后面的其他元素依次向前进行移动即可
        System.arraycopy(srcArray, 3, srcArray, 2, 3);
        // 遍历srcArray数组
        for(int x = 0; x < srcArray.length; x++) {
            if(x != srcArray.length - 1) {
                System.out.print(srcArray[x] + ", ");
            } else {
                System.out.println(srcArray[x]);
            }
        }
    }
}

```

运行程序进行测试，控制台的输出结果如下所示：

```
23, 45, 89, 14, 56, 56
```

通过控制台输出结果我们可以看到此时多出了一个56元素，此时我们只需要将最后一个位置设置为0即可。如下所示：

```

public class SystemDemo02 {
    public static void main(String[] args) {
        // 定义一个数组
        int[] srcArray = {23, 45, 67, 89, 14, 56};
        // 删除数组中第3个元素(67)：要删除67这个元素，我们只需要将67后面的其他元素依

```

次向前进行移动即可

```
System.arraycopy(srcArray , 3 , srcArray , 2 , 3);
// 将最后一个位置的元素设置为0
srcArray[srcArray.length - 1] = 0 ;
// 遍历srcArray数组
for(int x = 0 ; x < srcArray.length ; x++) {
    if(x != srcArray.length - 1 ) {
        System.out.print(srcArray[x] + ", ");
    }else {
        System.out.println(srcArray[x]);
    }
}
}
```

运行程序进行测试，控制台输出结果如下所示：

```
23, 45, 89, 14, 56, 0
```

此时我们可以看到元素"67"已经被删除掉了。67后面的其他元素依次向前进行移动了一位。

arraycopy方法底层细节：

- 1.如果数据源数组和目的地数组都是基本数据类型，那么两者的类型必须保持一致，否则会报错
- 2.在拷贝的时候需要考虑数组的长度，如果超出范围也会报错
- 3.如果数据源数组和目的地数组都是引用数据类型，那么子类类型可以赋值给父类类型

代码示例：

```
public class SystemDemo3 {
    public static void main(String[] args) {
        //public static void arraycopy(数据源数组, 起始索引, 目的地数组, 起始索引,
        拷贝个数) 数组拷贝
        //细节:
        //1.如果数据源数组和目的地数组都是基本数据类型，那么两者的类型必须保持一致，
        否则会报错
        //2.在拷贝的时候需要考虑数组的长度，如果超出范围也会报错
        //3.如果数据源数组和目的地数组都是引用数据类型，那么子类类型可以赋值给父类类
        型

        Student s1 = new Student("zhangsan", 23);
        Student s2 = new Student("lisi", 24);
        Student s3 = new Student("wangwu", 25);

        Student[] arr1 = {s1, s2, s3};
        Person[] arr2 = new Person[3];
        //把arr1中对象的地址值赋值给arr2中
```

```

        System.arraycopy(arr1, 0, arr2, 0, 3);

        //遍历数组arr2
        for (int i = 0; i < arr2.length; i++) {
            Student stu = (Student) arr2[i];
            System.out.println(stu.getName() + "," + stu.getAge());
        }
    }
}

class Person {
    private String name;
    private int age;

    public Person() {
    }

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

    /**
     * 获取
     *
     * @return name
     */
    public String getName() {
        return name;
    }

    /**
     * 设置
     *
     * @param name
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * 获取
     *
     * @return age
     */
    public int getAge() {
        return age;
    }

    /**
     * 设置
     *
     * @param age

```

```

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

    public String toString() {
        return "Person{name = " + name + ", age = " + age + "}";
    }
}

class Student extends Person {

    public Student() {
    }

    public Student(String name, int age) {
        super(name, age);
    }
}

```

3 Runtime

3.1 概述

Runtime表示Java中运行时对象，可以获取到程序运行时设计到的一些信息

3.2 常见方法

常见方法介绍

我们要学习的Object类中的常见方法如下所示：

<code>public static Runtime getRuntime()</code>	//当前系统的运行环境对象
<code>public void exit(int status)</code>	//停止虚拟机
<code>public int availableProcessors()</code>	//获得CPU的线程数
<code>public long maxMemory()</code>	//JVM能从系统中获取总内存大小（单位byte）
<code>public long totalMemory()</code>	//JVM已经从系统中获取总内存大小（单位
byte)	
<code>public long freeMemory()</code>	//JVM剩余内存大小（单位byte）
<code>public Process exec(String command)</code>	//运行cmd命令

代码示例：

```

public class RunTimeDemo1 {
    public static void main(String[] args) throws IOException {
        /*

```

```

        public static Runtime getRuntime() 当前系统的运行环境对象
        public void exit(int status) 停止虚拟机
        public int availableProcessors() 获得CPU的线程数
        public long maxMemory() JVM能从系统中获取总内存大小(单位byte)
        public long totalMemory() JVM已经从系统中获取总内存大小(单位byte)
        public long freeMemory() JVM剩余内存大小(单位byte)
        public Process exec(string command) 运行cmd命令
    */

    //1.获取Runtime的对象
    //Runtime r1 =Runtime.getRuntime();

    //2.exit 停止虚拟机
    //Runtime.getRuntime().exit(0);
    //System.out.println("看看我执行了吗?");

    //3.获得CPU的线程数
    System.out.println(Runtime.getRuntime().availableProcessors()); //8
    //4.总内存大小,单位byte字节
    System.out.println(Runtime.getRuntime().maxMemory() / 1024 /
1024); //4064
    //5.已经获取的总内存大小,单位byte字节
    System.out.println(Runtime.getRuntime().totalMemory() / 1024 /
1024); //254
    //6.剩余内存大小
    System.out.println(Runtime.getRuntime().freeMemory() / 1024 /
1024); //251

    //7.运行cmd命令
    //shutdown :关机
    //加上参数才能执行
    //-s :默认在1分钟之后关机
    //-s -t 指定时间 : 指定关机时间
    //-a :取消关机操作
    //-r: 关机并重启
    Runtime.getRuntime().exec("shutdown -s -t 3600");

}
}

```

3.3 恶搞好基友

需求:

界面上方按钮默认隐藏

界面中间有一个提示文本和三个按钮

当你的好基友点击中间三个按钮的时候就在N秒之后关机，不同的按钮N的值不一样

任意一个按钮被点击之后，上方了按钮出现。当点击上方按钮之后取消关机任务



```
public class Test {  
    public static void main(String[] args) {  
        new MyJframe();  
    }  
}
```

```
public class MyJframe extends JFrame implements ActionListener {  
  
    JButton yesBut = new JButton("帅爆了");  
    JButton midBut = new JButton("一般般吧");  
    JButton noBut = new JButton("不帅，有点磕碜");  
    JButton dadBut = new JButton("饶了我吧! ");  
  
    //决定了上方的按钮是否展示
```

```

boolean flag = false;

public MyJframe() {
    initJFrame();

    initView();

    //显示
    this.setVisible(true);
}

private void initView() {

    this.getContentPane().removeAll();

    if (flag) {
        //展示按钮
        dadBut.setBounds(50, 20, 100, 30);
        dadBut.addActionListener(this);
        this.getContentPane().add(dadBut);
    }

    JLabel text = new JLabel("你觉得自己帅吗? ");
    text.setFont(new Font("微软雅黑", 0, 30));
    text.setBounds(120, 150, 300, 50);

    yesBut.setBounds(200, 250, 100, 30);
    midBut.setBounds(200, 325, 100, 30);
    noBut.setBounds(160, 400, 180, 30);

    yesBut.addActionListener(this);
    midBut.addActionListener(this);
    noBut.addActionListener(this);

    this.getContentPane().add(text);
    this.getContentPane().add(yesBut);
    this.getContentPane().add(midBut);
    this.getContentPane().add(noBut);

    this.getContentPane().repaint();
}

private void initJFrame() {
    //设置宽高
    this.setSize(500, 600);
    //设置标题
    this.setTitle("恶搞好基友");
    //设置关闭模式

```



```

        this.setDefaultCloseOperation(3);
        //置顶
        this.setAlwaysOnTop(true);
        //居中
        this.setLocationRelativeTo(null);
        //取消内部默认布局
        this.setLayout(null);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if (obj == yesBut) {
            //给好基友一个弹框
            showJDialog("xxx, 你太自信了, 给你一点小惩罚");
            try {
                Runtime.getRuntime().exec("shutdown -s -t 3600");
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }
            flag = true;
            initView();
        } else if (obj == midBut) {
            System.out.println("你的好基友点击了一般般吧");

            //给好基友一个弹框
            showJDialog("xxx, 你还是太自信了, 也要给你一点小惩罚");

            try {
                Runtime.getRuntime().exec("shutdown -s -t 7200");
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }

            flag = true;
            initView();
        } else if (obj == noBut) {
            System.out.println("你的好基友点击了不帅");

            //给好基友一个弹框
            showJDialog("xxx, 你还是有一点自知之明的, 也要给你一点小惩罚");

            try {
                Runtime.getRuntime().exec("shutdown -s -t 1800");
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }

            flag = true;
            initView();
        }
    }

```

```

    } else if (obj == dadBut) {
        //给好基友一个弹框
        showJDialog("xxx, 这次就饶了你~");

        try {
            Runtime.getRuntime().exec("shutdown -a");
        } catch (IOException ioException) {
            ioException.printStackTrace();
        }
    }
}

public void showJDialog(String content) {
    //创建一个弹框对象
    JDialog jDialog = new JDialog();
    //给弹框设置大小
    jDialog.setSize(200, 150);
    //让弹框置顶
    jDialog.setAlwaysOnTop(true);
    //让弹框居中
    jDialog.setLocationRelativeTo(null);
    //弹框不关闭永远无法操作下面的界面
    jDialog.setModal(true);

    //创建JLabel对象管理文字并添加到弹框当中
    JLabel warning = new JLabel(content);
    warning.setBounds(0, 0, 200, 150);
    jDialog.getContentPane().add(warning);

    //让弹框展示出来
    jDialog.setVisible(true);
}
}

```

4 Object类

4.1 概述

tips: 重点讲解内容

查看API文档，我们可以看到API文档中关于Object类的定义如下：

```
public class Object
```

类 Object 是类层次结构的根类。每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法。

Object类所在包是java.lang包。Object 是类层次结构的根，每个类都可以将 Object 作为超类。所有类都直接或者间接的继承自该类；换句话说，该类所具备的方法，其他所有类都继承了。

查看API文档我们可以看到，在Object类中提供了一个无参构造方法，如下所示：

构造方法摘要

[Object\(\)](#)

但是一般情况下我们很少去主动的创建Object类的对象，调用其对应的方法。更多的是创建Object类的某个子类对象，然后通过子类对象调用Object类中的方法。

4.2 常见方法

tips：重点讲解内容

常见方法介绍

我们要学习的Object类中的常见方法如下所示：

```
public String toString()           //返回该对象的字符串表示形式(可以看做是对
象的内存地址值)
public boolean equals(Object obj)  //比较两个对象地址值是否相等；true表示相
同，false表示不相同
protected Object clone()          //对象克隆
```

案例演示

接下来我们就来通过一些案例演示一下这些方法的特点。

案例1：演示toString方法

实现步骤：

1. 创建一个学生类，提供两个成员变量（name，age）；并且提供对应的无参构造方法和有参构造方法以及get/set方法
2. 创建一个测试类（ObjectDemo01），在测试类的main方法中去创建学生对象，然后调用该对象的toString方法获取该对象的字符串表现形式，并将结果进行输出

如下所示：

Student类

```
public class Student {  
  
    private String name ;        // 姓名  
    private String age ;        // 年龄  
  
    // 无参构造方法和有参构造方法以及get和set方法略  
    ...  
  
}
```

ObjectDemo01测试类

```
public class ObjectDemo01 {  
  
    public static void main(String[] args) {  
  
        // 创建学生对象  
        Student s1 = new Student("itheima" , "14") ;  
  
        // 调用toString方法获取s1对象的字符串表现形式  
        String result1 = s1.toString();  
  
        // 输出结果  
        System.out.println("s1对象的字符串表现形式为: " + result1);  
  
    }  
  
}
```

运行程序进行测试，控制台输出结果如下所示：

```
s1对象的字符串表现形式为: com.itheima.api.system.demo04.Student@3f3afe78
```

为什么控制台输出的结果为：com.itheima.api.system.demo04.Student@3f3afe78；此时我们可以查看一下Object类中toString方法的源码，如下所示：

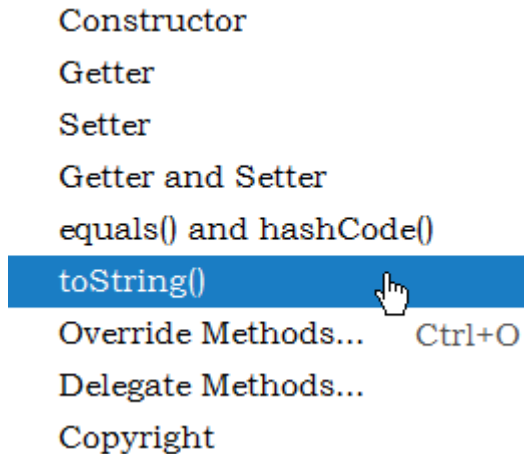
```
public String toString() {        // Object类中toString方法的源码定义  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

其中getClass().getName()对应的结果就是：com.itheima.api.system.demo04.Student；
Integer.toHexString(hashCode())对应的结果就是3f3afe78。

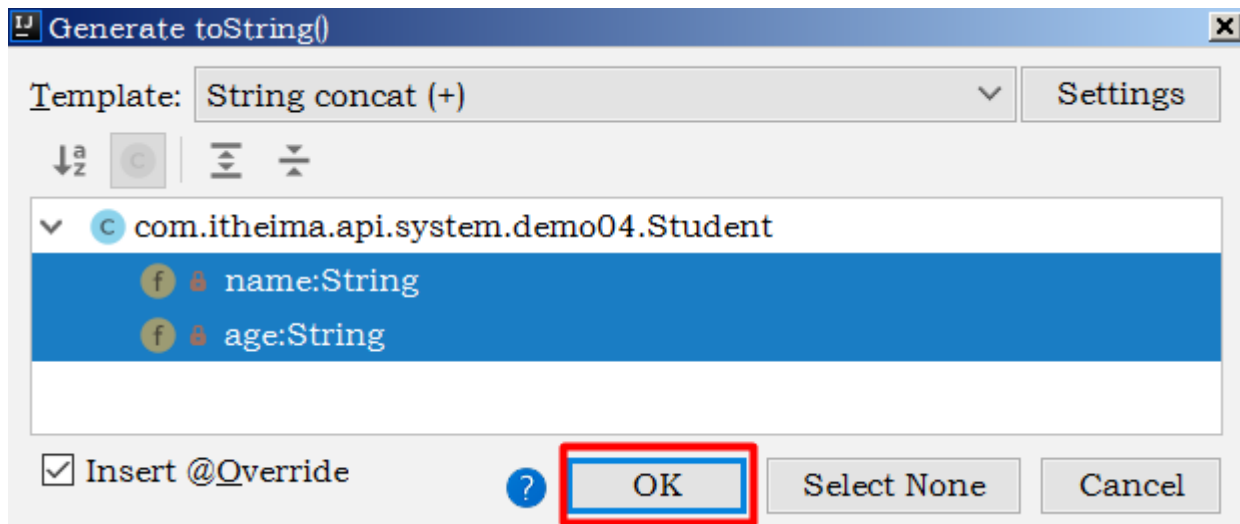
我们常常将"com.itheima.api.system.demo04.Student@3f3afe78"这一部分称之为对象的内存地址值。但是一般情况下获取对象的内存地址值没有太大的意义。获取对象的成员变量的字符串拼接形式才

算有意义，怎么实现呢？此时我们就需要在Student类中重写Object的toString方法。我们可以通过idea开发工具进行实现，具体步骤如下所示：

1. 在空白处使用快捷键：alt + insert。此时会弹出如下的对话框



2. 选择toString，此时会弹出如下的对话框



同时选择name和age属性，点击OK。此时就会完成toString方法的重写，代码如下所示：

```
@Override
public String toString() {
    return "Student{" +
        "name='" + name + '\'' +
        ", age='" + age + '\'' +
        '}';
}
```

这段代码就是把Student类中的成员变量进行了字符串的拼接。重写完毕以后，再次运行程序，控制台输出结果如下所示：

```
s1对象的字符串表现形式为：Student{name='itheima', age='14'}
```

此时我们就可以清楚的查看Student的成员变量值，因此重写toString方法的意义就是以良好的格式，更方便的展示对象中的属性值

我们再来查看一下如下代码的输出：

```
// 创建学生对象
Student s1 = new Student("itheima" , "14") ;

// 直接输出对象s1
System.out.println(s1);
```

运行程序进行测试，控制台输出结果如下所示：

```
Student{name='itheima', age='14'}
```

我们可以看到和刚才的输出结果是一致的。那么此时也就证明直接输出一个对象，那么会默认调用对象的toString方法，因此如上代码的等同于如下代码：

```
// 创建学生对象
Student s1 = new Student("itheima" , "14") ;

// 调用s1的toString方法，把结果进行输出
System.out.println(s1.toString());
```

因此后期为了方便进行测试，我们常常是通过输出语句直接输出一个对象的名称。

小结：

1. 在通过输出语句输出一个对象时，默认调用的就是toString()方法
2. 输出地址值一般没有意义，我们可以通过重写toString方法去输出对应的成员变量信息（快捷键：atl + insert，空白处 右键 -> Generate -> 选择toString）
3. toString方法的作用：以良好的格式，更方便的展示对象中的属性值
4. 一般情况下Jdk所提供的类都会重写Object类中的toString方法

案例2：演示equals方法

实现步骤：

1. 在测试类 (ObjectDemo02) 的main方法中, 创建两个学生对象, 然后比较两个对象是否相同

代码如下所示:

```
public class ObjectDemo02 {  
  
    public static void main(String[] args) {  
  
        // 创建两个学生对象  
        Student s1 = new Student("itheima" , "14") ;  
        Student s2 = new Student("itheima" , "14") ;  
  
        // 比较两个对象是否相等  
        System.out.println(s1 == s2);  
  
    }  
  
}
```

运行程序进行测试, 控制台的输出结果如下所示:

```
false
```

因为"=="号比较的是对象的地址值, 而我们通过new关键字创建了两个对象, 它们的地址值是不相同的。因此比较结果就是false。

我们尝试调用Object类中的equals方法进行比较, 代码如下所示:

```
// 调用equals方法比较两个对象是否相等  
boolean result = s1.equals(s2);  
  
// 输出结果  
System.out.println(result);
```

运行程序进行测试, 控制台的输出结果为:

```
false
```

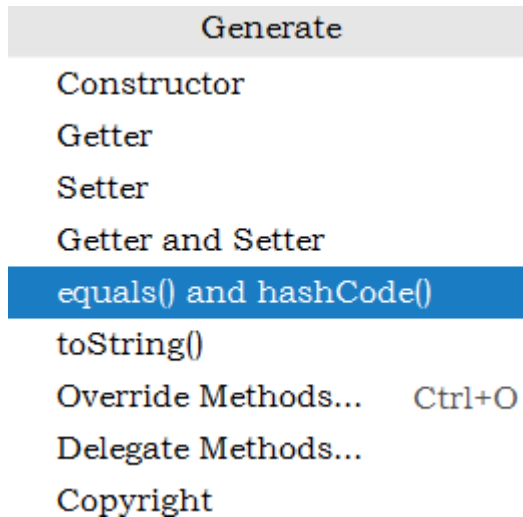
为什么结果还是false呢? 我们可以查看一下Object类中equals方法的源码, 如下所示:

```
public boolean equals(Object obj) {    // Object类中的equals方法的源码  
    return (this == obj);  
}
```

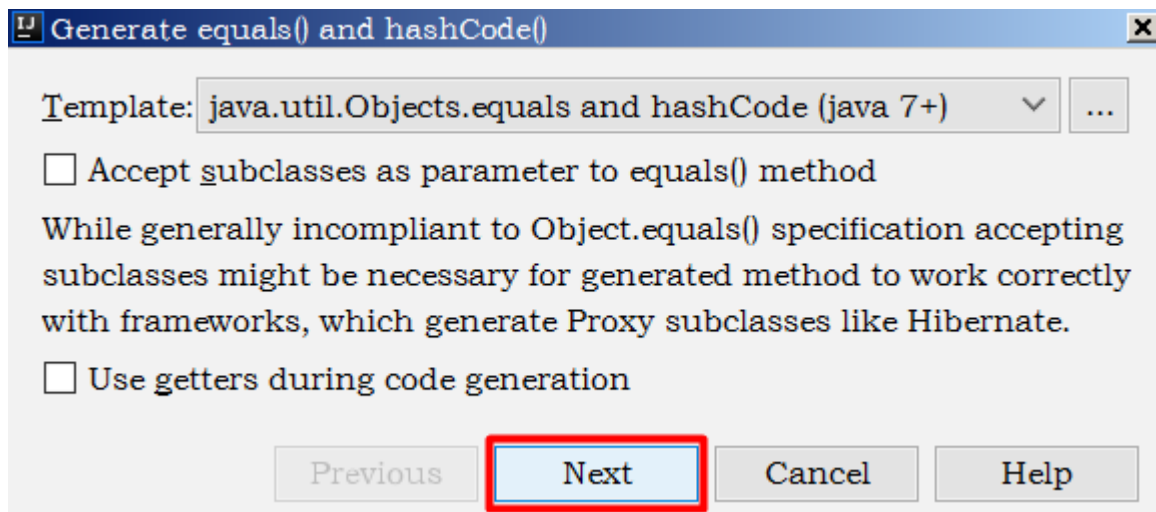
通过源码我们可以发现默认情况下equals方法比较的也是对象的地址值。比较内存地址值一般情况下是没有意义的，我们希望比较的是对象的属性，如果两个对象的属性相同，我们认为就是同一个对象；

那么要比较对象的属性，我们就需要在Student类中重写Object类中的equals方法。equals方法的重写，我们也可以使用idea开发工具完成，具体的操作如下所示：

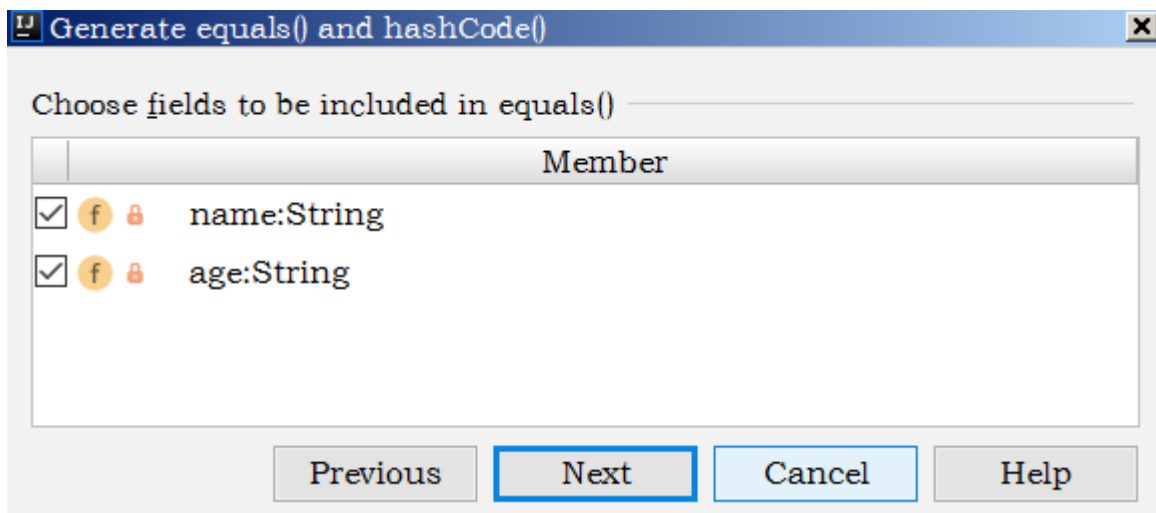
1. 在空白处使用快捷键：alt + insert。此时会弹出如下的对话框



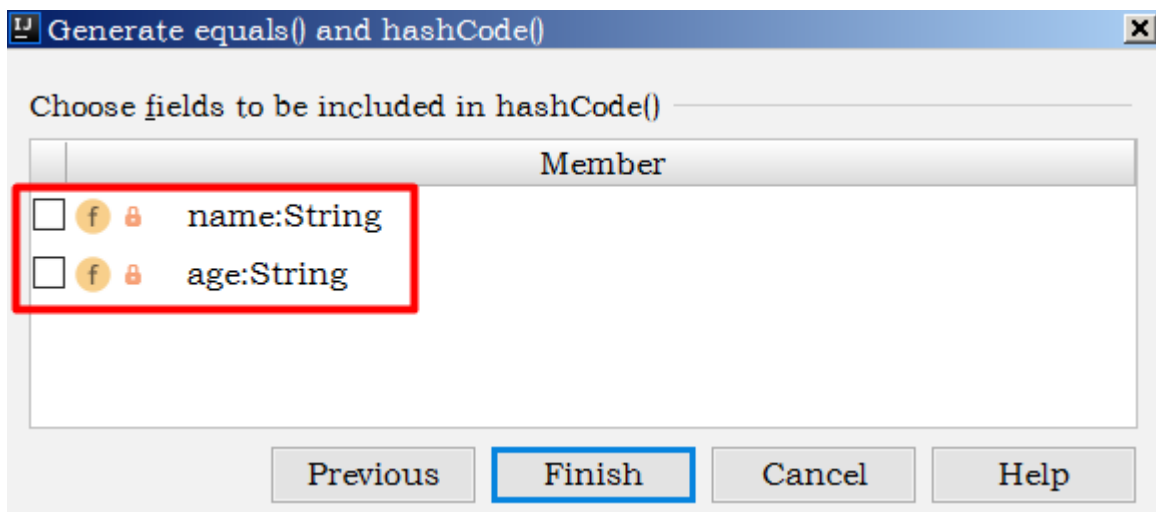
2. 选择equals() and hashCode()方法，此时会弹出如下的对话框



点击next，会弹出如下对话框：



选择name和age属性点击next，此时就会弹出如下对话框：



取消name和age属性（因为此时选择的是在生成hashCode方法时所涉及到的属性，关于hashCode方法后期再做重点介绍），点击Finish完成生成操作。生成的equals方法和hashCode方法如下：

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student student = (Student) o;
    return Objects.equals(name, student.name) && Objects.equals(age,
student.age);    // 比较的是对象的名字属性值和age属性值
}

@Override
public int hashCode() {
    return 0;
}
```

hashCode方法我们暂时使用不到，可以将hashCode方法删除。重写完毕以后运行程序进行测试，控制台输出结果如下所示：

```
true
```

此时equals方法比较的是对象的成员变量值，而s1和s2两个对象的成员变量值都是相同的。因此比较完毕以后的结果就是true。

小结：

1. 默认情况下equals方法比较的是对象的地址值
2. 比较对象的地址值是没有意义的，因此一般情况下我们都会重写Object类中的equals方法

案例2：对象克隆

把A对象的属性值完全拷贝给B对象，也叫对象拷贝,对象复制

对象克隆的分类：

深克隆和浅克隆

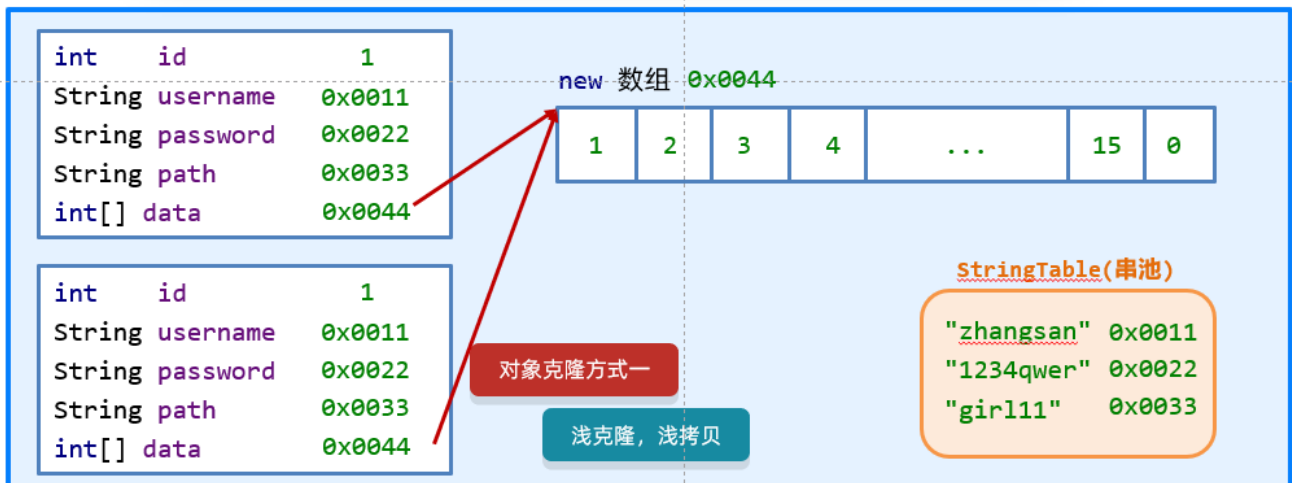
浅克隆：

不管对象内部的属性是基本数据类型还是引用数据类型，都完全拷贝过来

基本数据类型拷贝过来的是具体的数据，引用数据类型拷贝过来的是地址值。

Object类默认的是浅克隆

堆内存



深克隆：

基本数据类型拷贝过来，字符串复用，引用数据类型会重新创建新的

堆内存



代码实现：

```
package com.itheima.a04objectdemo;

public class ObjectDemo4 {
    public static void main(String[] args) throws CloneNotSupportedException {
        // protected object clone(int a) 对象克隆

        //1.先创建一个对象
        int[] data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0};
        User u1 = new User(1, "zhangsan", "1234qwer", "girl11", data);

        //2.克隆对象
        //细节:
        //方法在底层会帮我们创建一个对象,并把原对象中的数据拷贝过去。
        //书写细节:
        //1.重写Object中的clone方法
        //2.让javabean类实现Cloneable接口
        //3.创建原对象并调用clone就可以了
        //User u2 =(User)u1.clone();

        //验证一件事情: Object中的克隆是浅克隆
        //想要进行深克隆, 就需要重写clone方法并修改里面的方法体
        //int[] arr = u1.getData();
        //arr[0] = 100;

        //System.out.println(u1);
        //System.out.println(u2);

        //以后一般会用第三方工具进行克隆
        //1.第三方写的代码导入到项目中
        //2.编写代码
        //Gson gson =new Gson();
        //把对象变成一个字符串
        //String s=gson.toJson(u1);
```

```

        //再把字符串变回对象就可以了
        //User user = gson.fromJson(s, User.class);

        //int[] arr=u1.getData();
        //arr[0] = 100;

        //打印对象
        //System.out.println(user);

    }
}

package com.itheima.a04objectdemo;

import java.util.StringJoiner;

//Cloneable
//如果一个接口里面没有抽象方法
//表示当前的接口是一个标记性接口
//现在Cloneable表示一旦实现了，那么当前类的对象就可以被克隆
//如果没有实现，当前类的对象就不能克隆
public class User implements Cloneable {
    private int id;
    private String username;
    private String password;
    private String path;
    private int[] data;

    public User() {
    }

    public User(int id, String username, String password, String path, int[]
data) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.path = path;
        this.data = data;
    }

    /**
     * 获取
     *
     * @return id
     */
    public int getId() {
        return id;
    }
}

```

```

/**
 * 设置
 *
 * @param id
 */
public void setId(int id) {
    this.id = id;
}

/**
 * 获取
 *
 * @return username
 */
public String getUsername() {
    return username;
}

/**
 * 设置
 *
 * @param username
 */
public void setUsername(String username) {
    this.username = username;
}

/**
 * 获取
 *
 * @return password
 */
public String getPassword() {
    return password;
}

/**
 * 设置
 *
 * @param password
 */
public void setPassword(String password) {
    this.password = password;
}

/**
 * 获取
 *
 * @return path
 */
public String getPath() {
    return path;
}

```

```

    }

    /**
     * 设置
     *
     * @param path
     */
    public void setPath(String path) {
        this.path = path;
    }

    /**
     * 获取
     *
     * @return data
     */
    public int[] getData() {
        return data;
    }

    /**
     * 设置
     *
     * @param data
     */
    public void setData(int[] data) {
        this.data = data;
    }

    public String toString() {
        return "角色编号为: " + id + ", 用户名为: " + username + "密码为: " +
password + ", 游戏图片为:" + path + ", 进度:" + arrToString();
    }

    public String arrToString() {
        StringJoiner sj = new StringJoiner(", ", "[", "]");

        for (int i = 0; i < data.length; i++) {
            sj.add(data[i] + "");
        }
        return sj.toString();
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        //调用父类中的clone方法
        //相当于让Java帮我们克隆一个对象，并把克隆之后的对象返回出去。

        //先把被克隆对象中的数组获取出来
        int[] data = this.data;
        //创建新的数组
        int[] newData = new int[data.length];
    }

```

```

        //拷贝数组中的数据
        for (int i = 0; i < data.length; i++) {
            newData[i] = data[i];
        }
        //调用父类中的方法克隆对象
        User u=(User)super.clone();
        //因为父类中的克隆方法是浅克隆，替换克隆出来对象中的数组地址值
        u.data =newData;
        return u;
    }
}

```

5 Objects类

5.1 概述

tips: 了解内容

查看API文档，我们可以看到API文档中关于Objects类的定义如下：

Module java.base
软件包 java.util
Class Objects
java.lang.Object
java.util.Objects

```
public final class Objects
extends Object
```

Objects类所在包是在java.util包下，因此在使用的时候需要进行导包。并且Objects类是被final修饰的，因此该类不能被继承。

Objects类提供了一些对象常见操作的方法。比如判断对象是否相等，判断对象是否为null等等。

接下来我们来查看一下API文档，看一下Objects类中的成员，如下所示：

方法摘要

所有方法 静态方法 具体的方法

Modifier and Type	方法	描述
static int	checkFromIndexSize(int fromIndex, int size, int length)	检查是否在子范围从 fromIndex (含) 至 fromIndex + size (不包括) 是范围界限内 0 (包括) 到 length (不包括)。
static int	checkFromToIndex(int fromIndex, int toIndex, int length)	检查是否在子范围从 fromIndex (含) 至 toIndex (不包括) 是范围界限内 0 (包括) 到 length (不包括)。
static int	checkIndex(int index, int length)	检查 index 是否在范围从 0 (含) 到 length (排他) 的范围内。

我们可以发现Objects类中无无参构造方法，因此我们不能使用new关键字去创建Objects的对象。同时我们可以发现Objects类中所提供的方法都是静态的。因此我们可以通过类名直接去调用这些方法。

5.2 常见方法

tips: 重点讲解内容

常见方法介绍

我们要重点学习的Objects类中的常见方法如下所示：

```

public static String toString(Object o)           // 获取对象的字符串
表现形式
public static boolean equals(Object a, Object b) // 比较两个对象是否
相等
public static boolean isNull(Object obj)         // 判断对象是否为
null
public static boolean nonNull(Object obj)        // 判断对象是否不为
null

```

我们要了解的Objects类中的常见方法如下所示：

```

public static <T> T requireNonNull(T obj)           // 检查对象是否不为
null,如果为null直接抛出异常；如果不是null返回该对象；
public static <T> T requireNonNullElse(T obj, T defaultObj) // 检查对象是否不为
null, 如果不为null, 返回该对象；如果为null返回defaultObj值
public static <T> T requireNonNullElseGet(T obj, Supplier<? extends T>
supplier) // 检查对象是否不为null, 如果不为null, 返回该对象；如果
// 为null,返回由Supplier所提供的值

```

上述方法中的T可以理解为是Object类型。

案例演示

接下来我们就通过一些案例演示一下Objects类中的这些方法特点。

案例1：演示重点学习方法

实现步骤：

1. 创建一个学生类，提供两个成员变量（name，age）；并且提供对应的无参构造方法和有参构造方法以及get/set方法，并且重写toString方法和equals方法
2. 创建一个测试类（ObjectsDemo01），在该类中编写测试代码

如下所示：

Student类

```

public class Student {

    private String name ;    // 姓名
    private String age ;    // 年龄

    // 其他代码略
    ...

}

```



```
public class ObjectsDemo01 {

    public static void main(String[] args) {

        // 调用方法
        method_04() ;

    }

    // 测试nonNull方法
    public static void method_04() {

        // 创建一个学生对象
        Student s1 = new Student("itheima" , "14") ;

        // 调用Objects类中的nonNull方法
        boolean result = Objects.nonNull(s1);

        // 输出结果
        System.out.println(result);

    }

    // 测试isNull方法
    public static void method_03() {

        // 创建一个学生对象
        Student s1 = new Student("itheima" , "14") ;

        // 调用Objects类中的isNull方法
        boolean result = Objects.isNull(s1);

        // 输出结果
        System.out.println(result);

    }

    // 测试equals方法
    public static void method_02() {

        // 创建两个学生对象
        Student s1 = new Student("itheima" , "14") ;
        Student s2 = new Student("itheima" , "14") ;

        // 调用Objects类中的equals方法, 比较两个对象是否相等
        boolean result = Objects.equals(s1, s2);    // 如果Student没有重写
Object类中的equals方法, 此处比较的还是对象的地址值

        // 输出结果
        System.out.println(result);

    }

}
```

```

    }

    // 测试toString方法
    public static void method_01() {

        // 创建一个学生对象
        Student s1 = new Student("itheima" , "14") ;

        // 调用Objects中的toString方法,获取s1对象的字符串表现形式
        String result = Objects.toString(s1);          // 如果Student没有重写Object
类中的toString方法, 此处还是返回的对象的地址值

        // 输出结果
        System.out.println(result);

    }

}

```

案例2：演示需要了解的方法

```

public class ObjectsDemo02 {

    public static void main(String[] args) {

        // 调用方法
        method_03();

    }

    // 演示requireNonNullElseGet
    public static void method_03() {

        // 创建一个学生对象
        Student s1 = new Student("itheima" , "14") ;

        // 调用Objects对象的requireNonNullElseGet方法,该方法的第二个参数是Supplier
类型的, 查看源码我们发现Supplier是一个函数式接口,
        // 那么我们就可以为其传递一个Lambda表达式, 而在Supplier接口中所定义的方法是
无参有返回值的方法, 因此具体调用所传入的Lambda表达式如下所示
        Student student = Objects.requireNonNullElseGet(s1, () -> {
            return new Student("itcast", "14");
        });

        // 输出
        System.out.println(student);

    }

    // 演示requireNonNullElse

```

```

public static void method_02() {

    // 创建一个学生对象
    Student s1 = new Student("itheima" , "14") ;

    // 调用Objects对象的requireNonNullElse方法
    Student student = Objects.requireNonNullElse(s1, new Student("itcast",
"14"));

    // 输出
    System.out.println(student);

}

// 演示requireNonNull
public static void method_01() {

    // 创建一个学生对象
    Student s1 = new Student("itheima" , "14") ;

    // 调用Objects对象的requireNonNull方法
    Student student = Objects.requireNonNull(s1);

    // 输出
    System.out.println(student);

}

}

```

注：了解性的方法可以可以作为扩展视频进行下发。

6 BigInteger类

6.1 引入

平时在存储整数的时候，Java中默认是int类型，int类型有取值范围：-2147483648 ~ 2147483647。如果数字过大，我们可以使用long类型，但是如果long类型也表示不下怎么办呢？

就需要用到BigInteger，可以理解为：大的整数。

有多大呢？理论上最大到42亿的21亿次方

基本上在内存撑爆之前，都无法达到这个上限。

6.2 概述

查看API文档，我们可以看到API文档中关于BigInteger类的定义如下：

java.math

类 BigInteger

[java.lang.Object](#)

└ [java.lang.Number](#)

└ [java.math.BigInteger](#)

所有已实现的接口:

[Serializable](#), [Comparable](#)<[BigInteger](#)>

```
public class BigInteger
extends Number
implements Comparable<BigInteger>
```

BigInteger所在包是在java.math包下，因此在使用的时候就需要进行导包。我们可以使用BigInteger类进行大整数的计算

6.3 常见方法

构造方法

```
public BigInteger(int num, Random rnd)    //获取随机大整数，范围：[0 ~ 2的num次
方-1]
public BigInteger(String val)             //获取指定的大整数
public BigInteger(String val, int radix)   //获取指定进制的大整数
```

下面这个不是构造，而是一个静态方法获取BigInteger对象

```
public static BigInteger valueOf(long val) //静态方法获取BigInteger的对象，内部
有优化
```

构造方法小结:

- 如果BigInteger表示的数字没有超出long的范围，可以用静态方法获取。
- 如果BigInteger表示的超出long的范围，可以用构造方法获取。
- 对象一旦创建，BigInteger内部记录的值不能发生改变。
- 只要进行计算都会产生一个新的BigInteger对象

常见成员方法

BigDecimal类中使用最多的还是提供的进行四则运算的方法，如下:

```
public BigInteger add(BigInteger val)      //加法
public BigInteger subtract(BigInteger val)  //减法
public BigInteger multiply(BigInteger val)  //乘法
public BigInteger divide(BigInteger val)    //除法
public BigInteger[] divideAndRemainder(BigInteger val) //除法，获取商和余数
public boolean equals(Object x)            //比较是否相同
```

<code>public BigInteger pow(int exponent)</code>	//次幂、次方
<code>public BigInteger max/min(BigInteger val)</code>	//返回较大值/较小值
<code>public int intValue(BigInteger val)</code>	//转为int类型整数, 超出范围数据有误

代码实现：

```
package com.itheima.a06bigintegerdemo;

import java.math.BigInteger;

public class BigIntegerDemo1 {
    public static void main(String[] args) {
        /*
num次方-11
        public BigInteger(int num, Random rnd) 获取随机大整数, 范围:[0~ 2的
        public BigInteger(String val) 获取指定的大整数
        public BigInteger(String val, int radix) 获取指定进制的大整数

        public static BigInteger valueOf(long val) 静态方法获取BigInteger的
对象, 内部有优化

        细节:
        对象一旦创建里面的数据不能发生改变。
        */

        //1.获取一个随机的大整数
        /* Random r=new Random();
        for (int i = e; i < 100; i++) {
            BigInteger bd1 = new BigInteger(4,r);
            System.out.println(bd1);//[@ ~ 15]}
        }
        */

        //2.获取一个指定的大整数, 可以超出long的取值范围
        //细节:字符串中必须是整数, 否则会报错
        /* BigInteger bd2 = new BigInteger("1.1");
        System.out.println(bd2);
        */

        /*
        BigInteger bd3 = new BigInteger("abc");
        System.out.println(bd3);
        */

        //3.获取指定进制的大整数
        //细节:
        //1.字符串中的数字必须是整数
        //2.字符串中的数字必须要跟进制吻合。
        //比如二进制中, 那么只能写0和1, 写其他的就报错。
    }
}
```

```

BigInteger bd4 = new BigInteger("123", 2);
System.out.println(bd4);

//4.静态方法获取BigInteger的对象，内部有优化
//细节：
//1.能表示范围比较小，只能在long的取值范围之内，如果超出long的范围就不行了。
//2.在内部对常用的数字：-16 ~ 16 进行了优化。
// 提前把-16~16 先创建好BigInteger的对象，如果多次获取不会重新创建新的。
BigInteger bd5 = BigInteger.valueOf(16);
BigInteger bd6 = BigInteger.valueOf(16);
System.out.println(bd5 == bd6);//true

BigInteger bd7 = BigInteger.valueOf(17);
BigInteger bd8 = BigInteger.valueOf(17);
System.out.println(bd7 == bd8);//false

//5.对象一旦创建内部的数据不能发生改变
BigInteger bd9 =BigInteger.valueOf(1);
BigInteger bd10 =BigInteger.valueOf(2);
//此时，不会修改参与计算的BigInteger对象中的值，而是产生了一个新的BigInteger
对象记录
BigInteger result=bd9.add(bd10);
System.out.println(result);//3

}
}

```

```

package com.itheima.a06bigintegerdemo;

import java.math.BigInteger;

public class BigIntegerDemo2 {
    public static void main(String[] args) {
        /*
            public BigInteger add(BigInteger val) 加法
            public BigInteger subtract(BigInteger val) 减法
            public BigInteger multiply(BigInteger val) 乘法
            public BigInteger divide(BigInteger val) 除法，获取商
            public BigInteger[] divideAndRemainder(BigInteger val) 除法，获取商
            和余数

            public boolean equals(Object x) 比较是否相同
            public BigInteger pow(int exponent) 次幂
            public BigInteger max/min(BigInteger val) 返回较大值/较小值
            public int intValue(BigInteger val) 转为int类型整数，超出范围数据有误
        */

        //1.创建两个BigInteger对象
    }
}

```

```

BigInteger bd1 = BigInteger.valueOf(10);
BigInteger bd2 = BigInteger.valueOf(5);

//2.加法
BigInteger bd3 = bd1.add(bd2);
System.out.println(bd3);

//3.除法, 获取商和余数
BigInteger[] arr = bd1.divideAndRemainder(bd2);
System.out.println(arr[0]);
System.out.println(arr[1]);

//4.比较是否相同
boolean result = bd1.equals(bd2);
System.out.println(result);

//5.次幂
BigInteger bd4 = bd1.pow(2);
System.out.println(bd4);

//6.max
BigInteger bd5 = bd1.max(bd2);

//7.转为int类型整数, 超出范围数据有误
/* BigInteger bd6 = BigInteger.valueOf(2147483647L);
   int i = bd6.intValue();
   System.out.println(i);
   */

BigInteger bd6 = BigInteger.valueOf(200);
double v = bd6.doubleValue();
System.out.println(v); //200.0
}
}

```

6.4 底层存储方式:

对于计算机而言, 其实是没有数据类型的概念的, 都是0101010101, 数据类型是编程语言自己规定的, 所以在实际存储的时候, 先把具体的数字变成二进制, 每32个bit为一组, 存储在数组中。

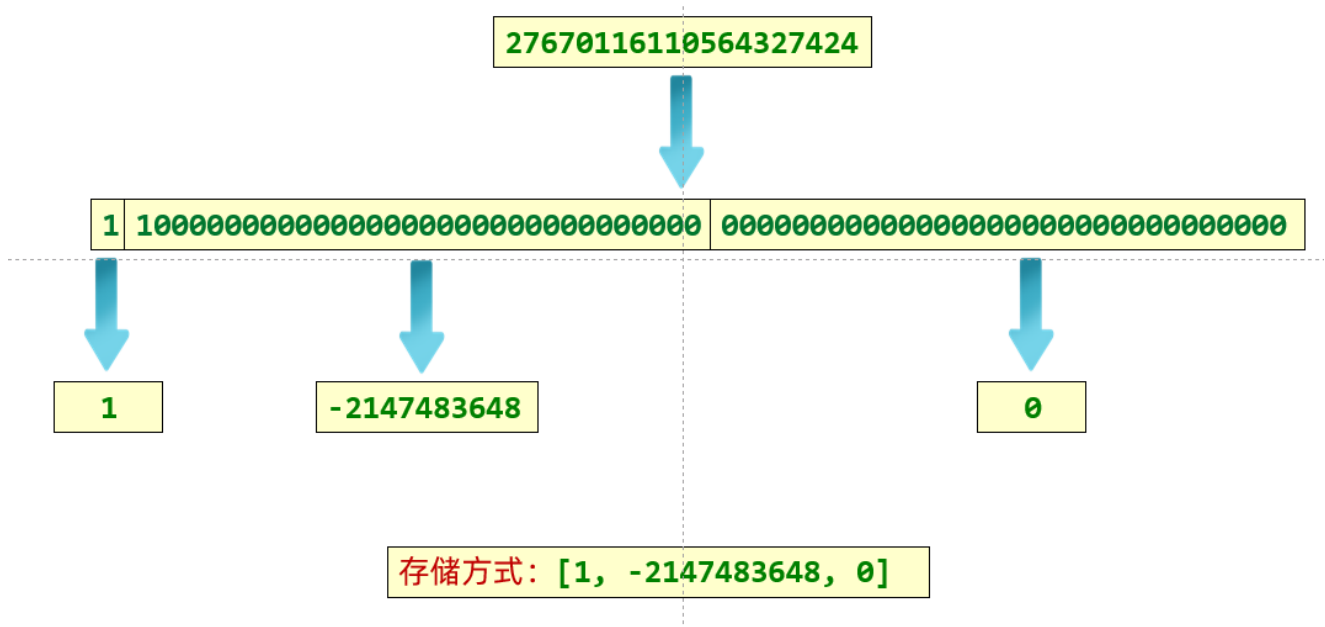
数组中最多能存储元素个数: 21亿多

数组中每一位能表示的数字: 42亿多

理论上, BigInteger能表示的最大数字为: 42亿的21亿次方。

但是还没到这个数字, 电脑的内存就会撑爆, 所以一般认为BigInteger是无限的。

存储方式如图所示:



7 BigDecimal类

7.1 引入

首先我们分析一下如下程序的执行结果：

```
public class BigDecimalDemo01 {  
  
    public static void main(String[] args) {  
        System.out.println(0.09 + 0.01);  
    }  
  
}
```

这段代码比较简单，就是计算0.09和0.01之和，并且将其结果在控制台进行输出。那么按照我们的想法在控制台输出的结果应该为0.1。那么实际的运行结果是什么呢？我们来运行一下程序，控制台的输出

结果如下所示：

```
0.09999999999999999
```

这样的结果其实就是一个丢失精度的结果。为什么会产生精度丢失呢？

在使用float或者double类型的数据在进行数学运算的时候，很有可能会产生精度丢失问题。我们都知道计算机底层在进行运算的时候，使用的都是二进制数据； 当我们在程序中写了一个十进制数据， 在

进行运算的时候，计算机将这个十进制数据转换成二进制数据，然后再进行运算，计算完毕以后计算机会把运算的结果再转换成十进制数据给我们展示； 如果我们使用的是整数类型的数据进行计算， 那

么在把十进制数据转换成二进制数据的时候不会存在精度问题； 如果我们的数据是一个浮点类型的数据，有的时候计算机并不会将这个数据完全转换成一个二进制数据，而是将这个将其转换成一个无限的

趋近于这个十进数的二进制数据； 这样使用一个不太准确的数据进行运算的时候， 最终就会造成精度丢失； 为了提高精度， Java就给我们提供了BigDecimal供我们进行数据运算。

7.2 概述

查看API文档，我们可以看到API文档中关于BigDecimal类的定义如下：

`java.math`

类 `BigDecimal`

`java.lang.Object`

└ `java.lang.Number`

└ `java.math.BigDecimal`

所有已实现的接口：

`Serializable`, `Comparable<BigDecimal>`

```
public class BigDecimal
extends Number
implements Comparable<BigDecimal>
```

BigDecimal所在包是在java.math包下，因此在使用的时候就需要进行导包。我们可以使用BigDecimal类进行更加精准的数据计算。

7.3 常见方法

构造方法

要用BigDecimal类，那么就需要首先学习一下如何去创建BigDecimal的对象。通过查看API文档，我们可以发现Jdk中针对BigDecimal类提供了很多的构造方法，但是最常用的构造方法是：

<code>BigDecimal</code> (int val)	将 int 转换为 BigDecimal。
<code>BigDecimal</code> (long val)	将 long 转换为 BigDecimal。
<code>BigDecimal</code> (String val)	将 BigDecimal 的字符串表示形式转换为 BigDecimal。

了解完常见的构造方法以后，我们接下来就重点介绍一下常见的成员方法。

常见成员方法

BigDecimal类中使用最多的还是提供的进行四则运算的方法，如下：

```

public BigDecimal add(BigDecimal value)        // 加法运算
public BigDecimal subtract(BigDecimal value)    // 减法运算
public BigDecimal multiply(BigDecimal value)    // 乘法运算
public BigDecimal divide(BigDecimal value)      // 触发运算

```

接下来我们就通过一些案例演示一下这些成员方法的使用。

案例1：演示基本的四则运算

代码如下所示：

```

public class BigDecimalDemo01 {

    public static void main(String[] args) {

        // 创建两个BigDecimal对象
        BigDecimal b1 = new BigDecimal("0.3") ;
        BigDecimal b2 = new BigDecimal("4") ;

        // 调用方法进行b1和b2的四则运算，并将其运算结果在控制台进行输出
        System.out.println(b1.add(b2));        // 进行加法运算
        System.out.println(b1.subtract(b2));    // 进行减法运算
        System.out.println(b1.multiply(b2));    // 进行乘法运算
        System.out.println(b1.divide(b2));      // 进行除法运算

    }

}

```

运行程序进行测试，控制台输出结果如下：

```

4.3
-3.7
1.2
0.075

```

此时我们可以看到使用BigDecimal类来完成浮点数的计算不会存在损失精度的问题。

案例2：演示除法的特殊情况

如果使用BigDecimal类型的数据进行除法运算的时候，得到的结果是一个无限循环小数，那么就会报错：ArithmeticException。如下代码所示：

```

public class BigDecimalDemo02 {

    public static void main(String[] args) {

```

```

        // 创建两个BigDecimal对象
        BigDecimal b1 = new BigDecimal("1") ;
        BigDecimal b2 = new BigDecimal("3") ;

        // 调用方法进行b1和b2的除法运算，并且将计算结果在控制台进行输出
        System.out.println(b1.divide(b2));

    }

}

```

运行程序进行测试，控制台输出结果如下所示：

```

Exception in thread "main" java.lang.ArithmeticException: Non-terminating
decimal expansion; no exact representable decimal result.
    at java.base/java.math.BigDecimal.divide(BigDecimal.java:1716)
    at
com.itheima.api.bigdecimal.demo02.BigDecimalDemo02.main(BigDecimalDemo02.java:1
4)

```

针对这个问题怎么解决，此时我们就需要使用到BigDecimal类中另外一个divide方法，如下所示：

```

BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)

```

上述divide方法参数说明：

```

divisor:          除数对应的BigDecimal对象;
scale:            精确的位数;
roundingMode:     取舍模式;
取舍模式被封装到了RoundingMode这个枚举类中（关于枚举我们后期再做重点讲解），在这个枚
举类中定义了很多种取舍方式。最常见的取舍方式有如下几个：
UP(直接进1)， FLOOR(直接删除)， HALF_UP(4舍五入)，我们可以通过如下格式直接访问这些
取舍模式：枚举类名.变量名

```

接下来我们就来演示一下这些取舍模式，代码如下所示：

```

public class BigDecimalDemo02 {

    public static void main(String[] args) {

        // 调用方法
        method_03() ;
    }
}

```

```

}

// 演示取舍模式HALF_UP
public static void method_03() {

    // 创建两个BigDecimal对象
    BigDecimal b1 = new BigDecimal("0.3") ;
    BigDecimal b2 = new BigDecimal("4") ;

    // 调用方法进行b1和b2的除法运算，并且将计算结果在控制台进行输出
    System.out.println(b1.divide(b2 , 2 , RoundingMode.HALF_UP));

}

// 演示取舍模式FLOOR
public static void method_02() {

    // 创建两个BigDecimal对象
    BigDecimal b1 = new BigDecimal("1") ;
    BigDecimal b2 = new BigDecimal("3") ;

    // 调用方法进行b1和b2的除法运算，并且将计算结果在控制台进行输出
    System.out.println(b1.divide(b2 , 2 , RoundingMode.FLOOR));

}

// 演示取舍模式UP
public static void method_01() {

    // 创建两个BigDecimal对象
    BigDecimal b1 = new BigDecimal("1") ;
    BigDecimal b2 = new BigDecimal("3") ;

    // 调用方法进行b1和b2的除法运算，并且将计算结果在控制台进行输出
    System.out.println(b1.divide(b2 , 2 , RoundingMode.UP));

}

}

```

小结：后期在进行两个数的除法运算的时候，我们常常使用的是可以设置取舍模式的divide方法。

7.4 底层存储方式：

把数据看成字符串，遍历得到里面的每一个字符，把这些字符在ASCII码表上的值，都存储到数组中。

```
BigDecimal bd = new BigDecimal("0.226");
```

```
[ 48, 46, 50, 50, 54]
```

```
BigDecimal bd = new BigDecimal("123.226");
```

```
[ 49, 50, 51, 46, 50, 50, 54]
```

```
BigDecimal bd = new BigDecimal("-1.5");
```

```
[ 45, 49, 46, 53]
```