Java语言概述

Java语言发展历史

1996年，JDK 1.0

2004年，发布里程碑式版本：JDK 1.5，为突出重要性，更名为**JDK 5.0**

2009年，Oracle公司收购SUN

2011年，JDK 7.0

2014年，**JDK 8.0**，是继JDK 5.0以来变化最大的版本

Java技术体系平台

Java SE(Java Standard Edition)

标准版，Java核心API

Java EE(Java Enterprise Edition)

企业版，Web应用开发，包含Servlet，Jsp等

Java ME(Java Micro Edition)

小型版，移动终端

Java语言的特点

1 面向对象

- 两个基本概念：类、对象
- 三大特性：封装、继承、多态

2 健壮性

去掉指针、内存的申请与释放等，提供一个相对安全的内存管理和访问机制

3 跨平台性

在操作系统上安装Java虚拟机(Java Virtual Machine)，由**JVM**来负责Java程序在该系统中的运行

JDK & JRE & JVM

- JDK = JRE + 开发工具集（eg. Javac编译工具等）
- JRE = JVM + Java SE 标准类库

Java代码的运行步骤

1 将Java代码编写到拓展名为.java的文件中

2 通过javac命令对该java文件**编译**，得到拓展名为.class的字节码文件

3 通过java命令对class文件运行，得到结果

注释（Comment）

```
//这是单行注释

/*
这是多行注释
这是多行注释
这是多行注释
...
*/
```

```
/**
文档注释，可以用javadoc解析，生成一套以网页文件形式的说明文档
命令行：javadoc -d setName -author -version javaFileName
@author arron
@version v1.0
*/
```

Java基本语法

关键字 & 保留字 & 标志符

关键字(Keyword)：class、interface、int、float、if、switch等

保留字(Reserved word)：现有Java版本未使用，以后可能使用，例如goto、const等

标志符(Identifier)：由英文字母，数字，_，$ 组成，其中数字不可以作为开头

命名规范(Naming Notations)

包名：xxxyyyzzz

类名、接口名：XxxYyyZzz

变量名、方法名：xxxYyyZzz

常量名：XXX_YYY_ZZZ

变量的使用

按数据类型分类

> 1 byte = 8 bit，表示数范围 -128～127
>
> 声明long类型整数变量，必须以"l"或"L"结尾
>
> 定义float类型浮点变量，必须以"f"或"F"结尾
>
> 通常，定义整型变量用int，定义浮点型变量用double

1 基本数据类型

- 数值型：byte(1 byte)、short(2)、int(4)、long(8)、float(4)、double(8)
- 字符型：char(2)
- 布尔型：boolean(1 bit)

> char定义必须使用单引号
>
> 换行符：\n

> 制表符：\t
>
> 引号：\"
>
> boolean只能取true、false

2 引用数据类型

- 类（class）
- 接口（interface）
- 数组（array）

基本数据类型之间的运算规则

1 自动类型提升

byte，short，char->int->long->float->double

当byte，short，char三种类型变量运算时，结果为int

整型变量，默认为int

浮点型变量，默认为double

2 强制类型转换

自动类型提升的逆运算（大容量->小容量）

```
//强转符()
double d = 12.9;
int i = (int)d;//12
```

可能导致精度损失

按声明位置分类

1 成员变量（类内，方法体外）

- 实例变量：不以static修饰
- 类变量：以static修饰

2 局部变量（方法体内）

- 形参（方法，构造器中定义）
- 方法局部变量
- 代码块局部变量

String类型变量

1 String属于引用数据类型

2 有String类型的'+'代表**连接**

3 没有String类型的'+'代表**加法**

> String定义必须使用双引号
>
> ASCII 码：
>
> A = 65
>
> a = 97

进制

二进制（binary）：以0B或0b开头

八进制（octal）：以0开头

十进制（decimal）

十六进制（hex）：以0X或0x开头

原码 & 反码 & 移码

- 对于正数，原码、反码、补码相同
- 对于负数，

-14的原码：10001110

-14的补码：11110001（符号位不变，取反）

-14的移码：11110010（反码+1）

> 计算机底层以补码的方式来存储数据

运算符

算术运算符：+ - * / % ++ --

赋值运算符：= += -= *= /= %=

比较运算符：== != < > <= >= instanceof

逻辑运算符：&逻辑与 | ！^ &&短路与 ||

> 两边均为boolean
>
> 左边为false，&继续执行右边，&&不执行
>
> 左边为true，|继续执行右边，||不执行
>
> 所以推荐使用&&和||

位运算符：<< >> >>>(无符号右移) & | ^ ~

三元运算符：(条件表达式)？表达式1：表达式2

程序流程控制

- 顺序结构
- 分支结构：if-else if-else、switch-case
- 循环结构：while、do-while、for

```
//输出10000以内的质数
public class PrimeNumber {
    public static void main(String[] args) {
        boolean isFlag = true;

        long startTime = System.currentTimeMillis();

        for(int i = 2;i < 10000;i++){
            for(int j = 2;j <= Math.sqrt(i);j++){
                if(i % j == 0){
                    isFlag = false;
                    break;
                }
            }
            if(isFlag == true){
                System.out.println("The Prime Number is: " + i);
            }else{
                isFlag = true;
            }
        }

        long endTime = System.currentTimeMillis();
        System.out.println("Time consuming: " + (endTime - startTime) + "ms");
    }
}
```

Scanner类的使用

```java
import java.util.Scanner;

class ScannerTest{
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    int num = scanner.nextInt();
    System.out.println(num);
  }
}
```

数组

一维数组

```java
//一维数组的声明
int[] array1;
//静态初始化
array1 = new int[]{1,2,3,4};
//动态初始化
String[] names = new String[5];
//获取数组的长度
System.out.println (names.length);
//遍历数组
for(int i = 0;i < names.length;i++){
  System.out.println(names[i]);
}
```

默认初始化值：

- 数组元素是整型：0
- 数组元素是浮点型：0.0
- 数组元素是char型：0或'\u0000'
- 数组元素是boolean型：false
- 数组元素是引用数据类型：null

二维数组

```java
//二维数组的声明
int[][] arr1;
//静态初始化
arr1 = new int[][]{{1,2,3},{4},{5,6}};
//动态初始化1
String[][] arr2 = new String[3][2];
//动态初始化2
String[][] arr3 = new String[3][];
```

```
//获取数组的长度
System.out.println (arr2.length);//3
System.out.println (arr1[0].length);//3
//遍历数组
for(int i = 0;i < arr2.length;i++){
  for(int j = 0;j < arr2[i].length;j++){
    System.out.print(arr2[i][j]+"\t");
  }
  System.out.println();
}
```

动态初始化1的默认初始化值：

- 外层元素arr[0]：地址值
- 内层元素arr[0] [0]：同一维数组

动态初始化2的默认初始化值：

- 外层元素arr[0]：null
- 内层元素arr[0] [0]：不能调用，否则报错

面向对象

基本概念

面向对象&面向过程

- 面向对象OOP（Object Oriented Programming）：强调具备了功能的对象，以类/对象为最小单位
- 面向过程POP（Process Oriented Programming）：强调功能行为，以函数为最小单位

类&对象

- 类（class）：是对一类事物的描述，是抽象的定义
- 对象（Object）：是实际存在的该类事物的每个个体，也称为实例（instance），类的实例化也就是创建类的对象
- 匿名对象（anonymous object）：创建对象时没有显式地赋给该对象一个变量名，只能调用一次，e.g. new Person()

形参&实参

- 形参：方法声明时的参数
- 实参：方法调用时的实际传给形参的参数值

> Java的实参值如何传入方法：

> 1.形参是基本数据类型，将实参类型的**数据值**传给形参
>
> 2.形参是引用数据类型，将实参类型的**地址值**传给形参

方法的重载（overload）& 方法的重写（override）

方法的重载

两同一不同：同一个类下、相同方法名；参数列表不同（个数或类型不同）

e.g. print(boolean)，print(char)，print(int)...

方法的重写

定义：在子类中可以根据需要对从父类中继承来的方法进行改造。程序执行时，子类的方法将覆盖父类的方法

方法的声明格式：

权限修饰符 返回值类型 方法名(形参){

    //方法体

}

说明：

- 子类重写方法的权限修饰符**大于等于**父类
- 返回值类型：父类为void，子类为void；父类为基本数据类型，子类为相同的基本数据类型；父类为引用数据类型A类，子类为A类或A类的子类
- 方法名和形参列表相同
- 子类重写的方法抛出的异常**小于等于**父类
- 方法体不一致

区分方法的重载和重写

1. 二者的概念
2. 重载和重写的规则
3. 重载不表现为多态性，重写表现为多态性

构造器（constructor）

作用：创建对象，给对象初始化

格式：权限修饰符 类名(形参){ };

说明：

- 如果没有显式的定义类的构造器，则系统默认提供一个空参的构造器
- 一个类中如果有多个构造器，彼此构成重载
- 一旦显式定义类的构造器，系统将不再提供默认的空参构造器
- 一个类中，至少有一个构造器

封装（Encapsulation）

高内聚低耦合

- 高内聚：类的内部数据操作细节自己完成，不允许外部干涉
- 低耦合：仅对外暴露少量的方法用于使用

设计思想

隐藏对象内部的复杂性，只对外公开简单的接口

比如，将类的属性私有化(private)，提供公共的(public)方法来获取(getXxx)和设置(setXxx)此属性的值

```
class Person{
  private int age;
  public int getAge(){
    return age;
  }
  public void setAge(int age){
    this.age = age;
  }
}
```

权限修饰符

权限修饰符	类内部	同一个包	不同包子类	同一个工程
private	✅			
default（缺省）	✅	✅		
protected	✅	✅	✅	
public	✅	✅	✅	✅

继承（Inheritance）

优点

- 减少代码的冗余，提高代码的复用性
- 便于功能的拓展
- 为之后多态性的使用，提供了前提

格式：class A extends B{ }

说明

- 当子类A继承父类B以后，子类A就获取了B中的结构（属性，方法）。特殊地，父类中private的属性、方法，子类也获取了，只是因为封装性的影响，使得子类不能直接调用父类的结构
- 子类继承父类以后，还可以声明自己特有的属性，方法，实现功能的拓展
- Java中的类只支持**单继承**，接口可以多继承

多态（Polymorphism）

理解为：一个事物的多种形态，实现代码的通用性

对象的多态性：父类的引用指向子类的对象。e.g. Person p = new Man();

使用

前提：类的继承关系；方法的重写

当调用子类、父类同名同参数的方法时，实际执行子类中重写父类的方法

体现

- Object类中定义的public boolean equals(Object obj)
- JDBC：使用Java程序操作（获取数据库连接，CRUD）数据库（MySQL，SQLServer，Oracle）
- 抽象类、接口的使用（抽象类、接口不可以实例化）

部分关键字和其他

this

- 理解为：当前对象的
- 用来修饰：属性、方法、构造器

super

- 理解为：父类的
- 用来修饰：属性、方法、构造器

instanceof

- 作用：a instanceof A：判断a是否是A的实例，返回true/false
- 使用场景：为避免向下转型时出现ClassCastException的异常，一般先判断
- 例子：class A extends B，如果a instanceof A是true，则a instanceof B也是true

static

用来修饰属性、方法、代码块、内部类。**随着类的加载而加载**

- static 属性：静态属性。当创建了类的多个对象，多个对象共享同一个静态变量。当通过某一个对象修改静态变量时，会导致其他对象调用此静态变量是已经修改过了的

- static 方法：静态方法。随着类的加载而加载，可以通过“类.静态方法“调用，不需要new对象。静态方法中，只能调用静态的方法或属性，不能使用this，super

> 1.开发中，如何确定一个**属性**是否声明为static?
>
> 属性是可以被多个对象所共享的，不会随着对象的不同而不同的，例如：银行利率，最低存款金额
>
> 2.开发中，如何确定一个**方法**是否声明为static?
>
> 操作静态属性的方法，通常设置为static

- static 代码块：静态代码块。随着类的加载而加载，只执行一次

final

用来修饰类、方法、属性

- 修饰类：该类不可以被其他类继承
- 修饰方法：该方法不可以被重写
- 修饰属性：该变量为常量
- static final 属性：全局常量

包装类（wrapper）

作用：使基本数据类型有了类的特点

基本数据类型	**包装类**
byte	Byte
short	Short
int	**Integer**
long	Long
float	Float
double	Double
boolean	Boolean
char	**Character**

相互转换关系：

基本数据类型to包装类：自动装箱

包装类to基本数据类型：自动拆箱

```
//自动装箱
Integer total = 99;
//自动拆箱
int totalCount = total;
```

基本数据类型toString类：'+'

String类to基本数据类型：例子：Integer.parseInt()

包装类toString类：toString()

== & equals()

- ==既可以比较基本数据类型（比较值），也可以比较引用数据类型（比较内存地址）
- equals()只能比较引用数据类型
- 像String，Date，File，Wrapper等都重写了Object类中的equals()方法，重写以后，比较的不是两个引用地址，而是“实体内容”

toString()

- 当我们输出一个对象的引用时，实际上是当前对象.toString()
- 像String，Date，File，Wrapper等都重写了Object类中的toString()方法，重写以后，使得返回实体内容

abstract

用来修饰类、方法

- abstract 类：抽象类。该类不可以实例化。一定有构造器，便于子类实例化时调用
- abstract 方法：抽象方法。

  只有方法的声明，没有方法体。e.g. public abstract void eat();

  包含抽象方法的类，一定是抽象类；抽象类中可以没有抽象方法

  若子类重写了父类中所有的抽象方法，可实例化；如果没有，子类需要用abstract修饰

接口（interface）

说明

- Java中，接口和类是并列关系
- 定义接口中的成员：JDK7之前，只能定义全局常量和抽象方法；JDK8之后，还可以定义静态方法，默认方法
- 接口中不能定义构造器，意味着接口不可以实例化
- Java开发中，接口通过让类去实现（implement）。如果实现类覆盖了接口中所有的抽象方法，则

可以实例化；如果没有，则仍为一个抽象类

- Java可以实现多个接口。e.g. class A extends B implement C,D,E{ }
- 接口与接口之间可以继承，而且可以多继承
- 接口的具体使用，体现多态性

```
interface A{
  int x = 0;
}
class B{
  int x = 1;
}
class C extends B implements A{
  public void printX(){
    System.out.println(super.X);//1
    System.out.println(A.x);//0
  }
  //...other methods
}
```

抽象类&接口

- 相同点：不能实例化，都可以被继承
- 不同点

	抽象类	接口
构造器	有	不能声明
继承性	单继承	多继承

异常处理

Error

定义：JVM无法解决的严重问题

如：JVM系统内部错误，资源耗尽

- 栈溢出：java.lang.StackOverFlowError
- 堆溢出：java.lang.OutOfMemeryError

Exception

定义：其他因编程错误或偶然的外在因素导致的

编译时异常：

- IOException
    - FileNotFoundException

运行时异常：

- AtithmeticException
- InputMismatchException
- NumberFormatException
- ClassCastException
- ArrayIndexOutOfBoundsException
- NullPointerException

抓抛模型

1.“抛”过程

两种方式：系统生成；手动生成throw new Exception( )

程序一旦出现异常，会在异常代码处生成一个异常类的对象，并将此对象抛出。一旦抛出对象后，其后的代码就不再执行

2.“抓”过程

异常的两种处理方式：

- try-catch-finally（处理）
- throws 异常类型（甩给上一级）

try-catch-finally

```
public class ExceptionTest{
  public void test(){
    String str = "abc";
    try{
      //可能出现异常的语句
      int num = Integer.parseInt(str);
    }catch(NumberFormatException e){
      //处理方式1
      e.printStackTrace();
    }catch(Exception e){
      //处理方式2，不处理，已执行上述异常处理语句，已跳出try-catch
      e.printStackTrace();
    }

    finally{
      //必须执行的语句
    }
  }
```

```
}
```

说明：

- catch中的类型如果存在子父类关系，则需要子类一定声明在父类上面，否则会报错
- 常用的异常对象处理方式：e.getMessage() 以及 e.printStackTrace()
- finally的使用场景：数据库连接，输入输出流，网络编程等

throw&throws

- throw：表示抛出一个异常类的对象，**生成异常对象**的过程，声明在方法体内
- throws：属于**异常处理**的一种方式，声明在方法的声明处

```
class Student{
  private int id;
  //异常处理的方式之一：throws
  public void regist(int id) throws Exception {
    if(id>0) {
      this.id = id;
    }else {
      //手动抛出异常
      throw new Exception("invalid input");
    }
  }
}
```

多线程基本概念

程序（program）

一段静态的代码

进程（process）

- 程序的一次执行过程，或者是正在运行的程序，如：运行中的QQ
- 程序是静态的，进程是动态的
- 进程是资源分配的单位

线程（thread）

- 进程可细分为线程，是一个程序内部的一条执行路径
- 若一个进程同一时间并行执行多个线程，就是支持多线程的
- 生命周期：新建--就绪--执行--（阻塞）--死亡

多线程（multi-thread）

优点

- 提高应用程序的响应，对图形化界面更有意义，增强用户体验
- 提高CPU的利用率
- 改善程序结构

应用场合

- 程序需要同时执行两个或多个任务
- 程序需要实现一些需要等待的任务时，比如：用户输入，文件读写，网络操作，搜索等
- 需要一些后台运行的程序时

生成线程

1. 通过继承：extends Thread

```
class MyThread extends Thread{
    public void run(){
        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0)
            System.out.println(i);
        }
    }
}


public class ThreadTest {
    public static void main(String[] args) {
        MyThread mt = new MyThread();
        //调用start () 方法：① 启动当前的线程 ② 调用当前线程的run () 方法
        mt.start();

        //如下操作仍是在main线程中执行的
        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0)
                System.out.println(i + "******  main()    ********");
        }

    }
}
```

2. 通过接口：implements Runnable

```
class MThread implements Runnable{

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if(i % 2 ==0){
```

```
                System.out.println(Thread.currentThread().getName() + ":" +
i);
            }
        }
    }
}

public class ThreadTest2 {
    public static void main(String[] args) {
        MThread mThread = new MThread();
        Thread t1 = new Thread(mThread);
        t1.start();

        Thread t2 = new Thread(mThread);
        t2.start();
    }
}
```

说明：

1. 在开发中，优先选择实现接口的方式来创建线程。因为实现的方式没有类的单继承性的局限性；实现的方式更适合来处理多个线程有共享数据的情况。
2. 两种方式的关系：public class Thread implements Runnable{ }
3. 相同点：都需要重写run()

Java同步机制来解决线程安全的问题

死锁

理解：

- 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
- 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续
- 使用同步时，避免出现死锁

同步机制

同步代码块

```
synchronized (同步监视器){
  //需要被同步的代码
}
```

说明：

- 操作共享数据的代码，即为需要同步的代码
- 共享数据：多个线程共同操作的变量
- 同步监视器：锁。任何一个类的对象，都可以充当锁，但是多个线程必须要共用同一把锁

```
package Thread;

//synchronized(同步监视器){
//      需要被同步的代码
//}

class Window3 implements Runnable{
    private int ticket = 100;
    Object obj = new Object();

    @Override
    public void run() {
        while(true){
            //this也可以作为对象
            synchronized(obj){
                if(ticket > 0){

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(Thread.currentThread().getName() + ":卖
票，票号为：" + ticket);
                    ticket--;
                }else{
                    break;
                }
            }
        }
    }
}

public class WindowTest3 {
    public static void main(String[] args) {
        Window3 w = new Window3();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");
```

```
        t1.start();
        t2.start();
        t3.start();

    }
}
```

同步方法

```
public synchronized void show(){
  // 需要被同步的代码
}
```

说明：

- 同步方法仍然涉及到同步监视器，只是不需要显式的声明
- 非static的同步方法，同步监视器为**this**
- static的同步方法，同步监视器为**当前类本身**

Lock（JDK5.0增加）

实现ReentrantLock类，调用lock()和unlock()

synchronized和Lock的异同点

相同点：二者都可以解决线程安全的问题

不同点：

1. synchronized机制在执行完相应的同步代码后，自动释放同步监视器
2. Lock需要手动地启动同步lock()，同时结束同步也需要手动实现unlock()

```
package Thread;

//解决线程安全问题方式三：Lock

import java.util.concurrent.locks.ReentrantLock;

class WindowLock implements Runnable{
    private int ticket = 100;

    //1.实例化ReentrantLock
    private ReentrantLock lock = new ReentrantLock();
    @Override
    public void run() {
```

```
        while(true){
            try{
                //2.调用锁定方法lock ()
                lock.lock();

                if(ticket > 0){

                    try {
                        Thread.sleep(100);
                    }catch (InterruptedException e){
                        e.printStackTrace();
                    }

                    System.out.println(Thread.currentThread().getName() + ":卖
票，票号为：" + ticket);
                    ticket--;
                }else {
                    break;
                }
            }finally {
                //3.调用解锁方法unlock ()
                lock.unlock();
            }

        }
    }
}

public class LockTest1 {
    public static void main(String[] args) {
        WindowLock wl = new WindowLock();

        Thread t1 =new Thread(wl);
        Thread t2 =new Thread(wl);
        Thread t3 =new Thread(wl);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

优先使用的顺序：

Lock--同步代码块--同步方法

线程通信

案例：使用两个线程打印1-100，交替打印

涉及到的方法：wait() & notify() & notifyAll()

- wait（）：一旦执行此方法，当前线程就会进入阻塞状态并释放同步监视器
- notify（）：一旦执行此方法，就会唤醒被wait的一个线程。如果有多个线程被wait，就会唤醒优先级高的
- notifyAll（）：一旦执行此方法，就会唤醒所有被wait的线程

说明：

- wait（），notify（），notifyAll（）三个方法必须使用在同步代码块或同步方法中
- wait（），notify（），notifyAll（）三个方法的调用者必须是同步代码块或同步方法中的同步监视器。否则，会出现IllegalManitorStateException
- wait（），notify（），notifyAll（）三个方法都定义在java.lang.Object类中

sleep() & wait()

相同点：一旦执行方法，都可以使当前的线程进入阻塞状态

不同点：

1. 声明的位置不同，Thread类声明sleep（），Object类声明wait（）
2. 调用范围不同，sleep（）可以在任何需要的场景下调用，wait（）必须在同步代码块或同步方法中
3. sleep（）不释放同步监视器，wait（）释放同步监视器

创建线程的其他方式

实现Callable接口（JDK5增加）

如何理解Callable接口比Runnable接口强大?

1. call（）可以有返回值
2. call（）可以抛出异常，被外面的操作捕获，获取异常信息
3. Callable是支持范型的

线程池

思路：

提前创建好多个线程放入线程池中，使用时直接获取，使用完放回池中，可以避免频繁地创建销毁，实现重复利用

优点：

1. 提高响应速度（减少创建新线程的时间）
2. 降低资源消耗
3. 便于线程管理

一些参数：
corePoolSize：核心池的大小

maximumPoolSize：最大线程数

keepAliveTime：最多保持时间

常用类

枚举类

说明：

1. 类的对象有限个，确定的，我们称之为枚举类
2. 当需要定义一组常量时，建议使用枚举类

使用enum关键字定义枚举类

```
enum Season{
  SPRING("spring","spring season");
  SUMMER("summer","summer season");

  // 声明season对象的属性：private final修饰
  private final String SeasonName;
  private final String SeasonDesc;

  // 私有化类的构造器，并给对象属性赋值
  private Season(String seasonName,String seasonDesc){
    this.seasonName = seasonName;
    this.seasonDesc = seasonDesc;
  }

  // ...一些public的方法
}
```

Date类

基本概念

两个构造器：

Date():创建当前时间的Date对象

Date(long time):创建指定毫秒数的Date对象

> System.currentTimeMills():返回当前时间（毫秒数）

两个方法：

toString():显示当前的年月日时分秒

getTime():获取当前Date对象的毫秒数（时间戳）

相关的API

- SimpleDateFormat类
- Calendar抽象类
- LocalDate类
- LocalTime类
- LocalDateTime类
- DataTimeFormatter类

BigInteger类&BigDecimal类

BigInteger可以表示不可变的任意精度的整数

要求数字精度比较高，用到java.math.BigDecimal类

String类

说明

- String声明为final的，不可被继承
- String实现了Serializable接口：表示支持序列化；实现了Comparable接口：表示String可以比较大小
- String内部定义了final char[] value用于存储字符串数据
- String代表**不可变**的字符序列。体现：对当前字符串重新赋值时，需要重新指定内存区域
- 通过字面量的方式（区别于new）给一个字符串赋值，此时的字符串值声明在字符串常量池中
- 字符串常量池中是不会存储相同内容的字符串的

String & StringBuffer & StringBuilder

相同点：底层使用char[]存储

不同点：

String：不可变的字符序列

StringBuffer：可变的字符序列，线程安全的，效率低(synchronized)

StringBuilder：可变的字符序列，线程不安全的，效率高

效率：StringBuilder>StringBuffer>String

StringBuffer & StringBuilder的方法

增：append(***)

删：delete(int start,int end)

改：setCharAt(int n, char ch) / replace(int,int,String)

查：charAt(int n)

长度：length()

遍历：for + charAt()

Java 比较器

使用背景

Java中的对象，正常情况下，只能进行等于/不等于比较，不能使用大于/小于。但在开发中，经常需要比较对象的大小。

实现

- 实现Comparable接口
- 实现Comparator接口

自然排序：使用Comparable接口

1. 像String，Wrapper等实现了Comparable接口，重写了compareTo（obj）方法，进行了**从小到大**的排序
2. 重写compareTo（obj）的规则：

如果当前对象this大于形参对象obj，则返回正整数

如果当前对象this小于形参对象obj，则返回负整数

如果当前对象this等于形参对象obj，则返回0

e.g. S1.compareTo(S2); //1,则S1>S2

3. 对于自定义类，如果需要排序，我们可以自定义类实现Comparable接口，重写compareTo（obj）方法

```
@override
public int compareTo(obj o){
  if(o instanceof Goods){
    Goods goods = (Goods)o;
    if(this.price > goods.price){
      return 1;
    }else if(this.price < goods.price){
      return -1;
```

```
    }else{
      return 0;
    }
  }
  throw new RuntimeException("Invalid!");
}
```

定制排序：实现Comparator接口

1. 背景：当实现了Comparable接口，但排序规则不适合当前的操作
2. 重写compare(Object o1,Object o2)方法：

如果方法返回正整数，则表示o1>o2;

如果方法返回负整数，则表示o1<o2;

如果方法返回0，则表示o1==o2.

两种排序方式的比较

1. Comparable接口的方式一旦确定，保证Comparable接口实现类的对象在任何位置都可以比较大小
2. Comparator接口属于临时性的比较

集合

集合与数组

定义：集合与数组都是对多个数据进行存储操作的结构，简称**Java容器**

数组存储的特点：一旦初始化以后，其长度就确定了；数组一旦定义好，其元素的类型也确定了

数组存储的缺点：长度不可修改；数组中提供的方法很有限

集合的分类

- 集合框架
  - Collection接口：单列集合，用来存储一个一个的对象
    - List接口：存储有序，可重复的数据（类似“动态”数组）--ArrayList、LinkedList、Vector
    - Set接口：存储无序，不可重复的数据（类似高中“集合”）--HashSet、LinkedHashSet、TreeSet
  - Map接口

Collection接口

遍历Collection元素的方法：

1. 迭代器接口：Iterator

```
public void test(){
  Collection collection = new ArrayList();
  collection.add(123456);
  Iterator iterator = collection.iterator();
  // 判断是否还有下一个元素
  while (iterator.hasNext()){
    System.out.println(iterator.next());
  }
}
```

2. foreach（内部仍然调用了迭代器）

```
//for(集合元素类型 局部变量：集合对象)
for(Object obj:collection){
  System.out.println(obj);
}
```

Collection子接口：List

存储有序的，可重复的数据

- ArrayList：List接口的主要实现类，线程不安全，效率高；底层使用Object[] elementData存储
- LinkedList：对于频繁地插入删除操作，使用此类的效率比ArrayList高，底层使用双向链表存储
- Vector：List接口的古老实现类，线程安全，效率低，底层使用Object[] elementData存储

常用方法：

增：add(Object obj)

删：remove(int index)/remove(Object obj)

改：set(int index,Object element)

查：get(int index)

插：add(int index, Object element)

长度：size()

遍历：Iterator迭代器、foreach、普通循环

> 存储元素的要求：添加的对象，所在的类要重写equals（）方法

Collection子接口：Set

存储无序的，不可重复的数据

以HashSet为例

无序性：不等于随机性，存储的数据在底层数组中并非按照索引的顺序添加，而是根据数据的哈希值确定的

不可重复性：保证添加的元素按照equals（）判断时，不能返回true，即相同的元素只能添加一个

元素添加过程：

- 1.向HashSet中添加元素a，首先调用元素a所在类的hashcode（），计算a的哈希值，此哈希值接着通过某种算法计算出HashSet底层数组中的存放位置
- 2.判断数组此位置上是否已有元素
    - 2.1 没有元素，则元素a添加成功（情况一）
    - 2.2 有元素b，则比较元素a和元素b的哈希值
        - 2.2.1 如果hash值不同，则a添加成功（情况二）
        - 2.2.2 如果hash值相同，调用a所在类的equals（）
            - 2.2.2.1 equals（）返回true，a添加失败
            - 2.2.2.2 equals（）返回false，a添加成功（情况三）

情况二和情况三：

jdk7：元素a放到数组中，指向原来的元素

jdk8：原来的元素在数组中不变，指向a

HashSet底层：数组+链表（jdk7）

分类：

- HashSet：Set接口的主要实现类，线程不安全的，可以存储null值
- LinkedHashSet：HashSet的子类
    - 遍历其内部数据时，可以按照添加的顺序遍历（因为在添加数据的同时，每个数据会维护两个引用，分别记录前后的两个数据）
    - 对于频繁的遍历操作，LinkedHashSet效率高于HashSet
- TreeSet：可以按照添加对象的指定属性，进行排序。底层是红黑树，与排序相关，应用到Comparable

存储对象所在类的要求：

HashSet/LinkedHashSet：向Set中添加的数据，其所在的类一定要重写hashCode（）和equals（）

Map接口

双列数据，存储key-value对的数据

分类：

- HashMap：Map的主要实现类，线程不安全，效率高，可以存储null的key，value
- LinkedHashMap：HashMap的子类。
  - 保证在遍历map元素时，可以按照添加的顺序实现遍历（因为在原HashMap的基础上，添加了一对指针，指向前一个元素和后一个元素）
  - 对于频繁的遍历操作，效率高于HashMap
- TreeMap：保证按照添加的key-value对进行排序，实现排序遍历。此时考虑key的自然排序或定制排序，底层使用红黑树
- HashTable：古老的Map实现类，线程安全，效率低，不能存储null的key或value
- Properties：HashTable子类，常用来处理配置文件。key和value都是String类型

HashMap的底层：

数组+链表（jdk7）

数组+链表+红黑树（jdk8）

HashMap的实现原理

一、在jdk7中的实现原理

HashMap map = new HashMap();

在实例化以后，底层创建了长度为16的一维数组Entry[] table

map.put(key1,value1);

Map.put(key2,value2);...

首先调用key1所在类的hashCode（）计算key1的hash值，此hash值经过某种算法计算以后，得到在Entry[]中的存放位置

- 1 如果此位置上的数据为空，则key1-value1添加成功
- 2 如果此位置上的数据不为空，则比较key1与其他数据的hash值
  - 2.1 key1的hash值与其他数据的hash值不同，则添加成功
  - 2.2 key1的hash值与其他数据的hash值相同，则继续比较equals（）
    - 2.2.1 返回false，key1-value1添加成功
    - 2.2.2 返回true，value1替换value x

二、HashMap在jdk8相比于jdk7的不同

- new HashMap（）：底层没有创建长度为16的数组
- jdk8底层的数组是：Node[]，而不是Entry[]
- 首次调用put（），底层创建长度为16的数组
- 七上八下

Collections工具类

作用：操作Collection和Map的工具类

常用方法：

- reverse(List)
- shuffle(List)：对List集合元素进行随机排序
- sort(List)
- swap(Listing,int)

说明：ArrayList和HashMap线程不安全，我们可以使用synchronizedList(List list)和synchronizedMap(Map map)

IO流

File类

理解

- File类的一个对象，代表一个文件或者文件目录
- File类声明在java.io包下
- File类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小写等写法，并未涉及到写入或读取文件的操作。如果需要写入或读取文件内容，必须使用IO流来完成
- 后续File类的对象常会作为参数传递到流的构造器中，指明读取或写入的“终点”

File的实例化

常用的构造器：

File(String filePath)

File(String parentPath,String childPath)

File(File parentFile,String childPath)

路径的分类：

相对路径：相较于某个路径下，指明的路径

绝对路径：包含盘符在内的文件或文件目录的路径

IO流

流的分类

1. 操作数据单位：字节流，字符流
2. 数据的流向：输入流、输出流

3. 流的角色：节点流、处理流

重要的流结构

抽象基类	InputStream	OutputStream	Reader	Writer
节点流	FileInputStream	FileOutputStream	FileReader	FileWriter
缓冲流	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter

输入、输出的标准化过程

输入过程：

1. 创建File类的对象，指明读取的数据来源（文件一定要存在）
2. 创建对应的输入流，将File类的对象作为参数，传入流的构造器中
3. 具体的读入过程：创建相应的byte[]或char[]
4. 关闭流资源

说明：程序中出现的异常需要使用try-catch-finally处理

```
File file1 = new File("input.txt");
File file2 = new File("output.txt");
//
FileInputStream fis = new FileInputStream(file1);
FileOutputStream fos = new FileOutputStream(file2);
// 缓冲流增加效率
BufferedInputStream bis = new BufferedInputStream(fis);
BufferedOutputStream bos = new BufferedOutputStream(fos);
// 处理
byte[] buffer = new byte[10];
int len;
while((len=bis.read(buffer))!=-1){
  bos.write(buffer,0,len);
}
//try-catch-finally
bis.close();
bos.close();
```

输出过程：

1. 创建File类对象，指明写出数据的位置
2. 创建输出流
3. write(char[]/byte[] buffer,0,len)
4. 关闭

> 1. 对于文本文件（.txt,.java,.c,.cpp），使用字符流处理
> 2. 对于非文本文件（.jpg,.mp3,.mp4,.doc），使用字节流处理

缓冲流

作用：提高流的读取，写入的速度

原因：内部提供了一个缓冲区，默认情况是8kb

flush（）：刷新缓冲区，将缓冲区的内容写出

转换流

作用：提供字节流和字符流之间的转换

属于字符流

- InputStreamReader：字节输入流to字符输入流
  - 解码：字节，字节数组to字符数组，字符串
- OutputStreamWriter：字符输出流to字节输出流
  - 编码：字符数组、字符串to字节数组、字节

对象流

- ObjectOutputStream：序列化过程
  - 内存中的对象to存储中的文件、通过网络传输出去
- ObjectInputStream：反序列化过程
  - 存储中的文件、通过网络接收过来to内存中的对象

注解Annotation

理解

代码里的特殊标记，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理

框架 = 注解 + 反射机制 + 设计模式

使用案例

- 自动生成的文档注释
- JDK内置的基本注解
  - @Override
  - @Deprecated表示所修饰的元素（类，方法等）已过时
  - @SuppressWarnings抑制编译器警告
- 调用框架，配置文件

元注解

对现有的注解进行解释说明的注解

jdk提供的4种元注解：

@Retention：指定修饰的Annotation的生命周期：SOURSE/CLASS（默认）/RUNTIME（只有RUNTIME，才能通过反射获取）

@Target：用于修饰哪些程序元素

@Documented：表示被修饰的Annotation在被javadoc解析时保留

@Inherited：具有继承性

如何获取注解信息：通过反射来进行获取调用

前提：元注解Retention声明的生命周期为RUNTIME