java 1.8参考文档：

<https://docs.oracle.com/javase/8/docs/api/index.html>

本文档使用模板

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
|  | 构造 |  |
|  | 普通 |  |

#### String，StringBuffer，StringBuild

##### String

1.1 String类两种实例化方式

1. 直接赋值 可能创建0或者1个对象，如果在java String池中已有，则不创建，没有则创建。只开辟一块内存空间，自动入池
2. 构造方法 可能创建1个或2个对象，如果在java String池中已有，则只在堆中创建一个，没有则在java String常量池中创建一份，堆中也创建一个。开辟两块空间，其中一块成为垃圾，不会自动入池，需使用intern()方法手动入池

1.2 String常用方法：

|  |  |  |
| --- | --- | --- |
| 操作分类 | 类型 | 函数 |
| 字符串比较 | 普通 | boolean equals(String str) |
| 普通 | boolean equalsIgnoreCase(String str) |
| 普通 | int compareTo(String str) |
| 字符串查找 | 普通 | boolean contains(String str) |
| 普通 | int indexOf(String str) |
| 普通 | int indexOf(String str, int fromIndex) |
| 普通 | int last**I**ndexOf(String str) |
| 字符串替换 | 普通 | String repalce(String regex, String str) |
| 普通 | String replaceAll(String regex, String str)--效果同上 |
| 普通 | String replaceFirst(String regex, String str) |
| 字符串截取 | 普通 | String sub**s**tring(int fromIndex) |
| 普通 | String sub**s**tring(int fromIndex,int endIndex)--其中endIndex <= str.length() |
| 字符串拆分 | 普通 | String[] split(String regex) |
| 其他 | 普通 | boolean isEmpty() |
| 普通 | int length() |
| 普通 | String trim() |
| 普通 | String toLowerCase() |
| 普通 | String toUpperCase() |
| 普通 | String intern() |
| 普通 | String concat(String str) |
|  | 普通 | static String format(String format, Object… args) |

##### StringBuffer

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| StringBuffer(String str) | 构造 | 接收字符串，同时多预留16个字符空间 |
| public String toString() | 普通 | StringBuffer转化成字符串 |
| StringBuffer append(String str) | 普通 | 将String类型的字符串拼接到StringBuffer对象后面 |
| public StringBuffer reverse() | 普通 | 反转(和String的有区别的方法) |
| public StringBuffer insert(int offset, 数据类型); | 普通 | 数据插入(和String的有区别的方法)，数据类型可以是：基本数据类型，CharSequence,String,Object |
| public StringBuffer delete(int start, int end); | 普通 | 删除部分数据(和String的有区别的方法) |
| public StringBuffer replace(int start, int end, [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) str) | 普通 | 替换 |

结构：

|  |  |
| --- | --- |
| public **final** class String  extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)  implements [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)<[String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)>, [**CharSequence**](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html) | public **final** class StringBuffer  extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)  implements [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [**CharSequence**](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html) |

##### StringBuilder

StringBuilder和StringBuffer区别：

StringBuilder线程不同步，所以性能也比较好；StringBuffer线程同步（方法上都加了Synchronizes），所以性能没不同步的好。

##### 三者区别

源码看出，字符串其实是以字符数组的形式保存的。字符串方法中，改变字符串内容则修改字符数组，并用新字符数组new String。所以大量的字符串操作必导致内存损耗大，时间效率低。StringBuffer中的方法也是对字符数组操作，但不会new String,只对本对象改变，再返回this。

相对执行效率：StringBuilder > StringBuffer > String

但是String str = "hello"+ "world"的效率就比 StringBuilder st = new StringBuilder().append("hello").append("world")要高

所以，适合的使用场景：

String：适用于少量的字符串操作的情况

StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

另，

都是final类，无法被继承。

三者都实现CharSequence接口，所以引用类型都可以使用CharSequence。

String作为形参，不会被调用的函数改变，StringBuffer会被改变

基本数据类型作为形参，调用函数前后，基本数据值不变，

对象数据类型作为形参，调用函数前后，对象内容会被改变，**字符串除外**。但若是改变对象数据类型的形参的指向，再改变形参的内容，则对实参没有任何影响。如无法理解，可参照<https://www.cnblogs.com/dolphin0520/p/3736238.html>最后。

|  |
| --- |
| **public static void** main(String[] args) {  String string = **"hello"**;  System.***out***.println(*change*(string));   StringBuffer sb = **new** StringBuffer(string);  System.***out***.println(*change*(sb));  } **public static** String change(String string){  string = **"world"**;  **return** string; } **public static** StringBuffer change(StringBuffer sb){  sb.append(**"world"**);  **return** sb; } |

#### 访问修饰符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 同一包内 | | 不同包内 | |
| 修饰符 | 当前类 | 非当前类（含子孙类） | 子孙类 | 非子孙类 |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| default | Y | Y | N | N |
| private | Y | N | N | N |

注意点：

1. 子孙类访问父类都使用使用super.父类属性或方法，super关键字可省略。

2. public:调用不同包内的java文件内的public类，需要import进来。同一包内public类调用不需要import。

3. private:只能在本类{}范围中使用，若想获得或修改，可在类中另外定义函数返回。(getInfo或者toString)

#### static，final，public static final

##### static

* 修饰类：

只能修饰内部类，新建内部类的方法变成：

外部类名.内部类名 内部类对象 = new 外部类名.内部类名()

不加static，外部类名.内部类名 内部类对象 = new 外部类名().new内部类名()

* 修饰变量：

1. 静态变量也被称为类变量。
2. static关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。
3. 局部变量不能被声明为static变量。
4. 保存在全局数据区。

* 修饰方法：

1. static方法主要用来操作static属性。
2. static方法中不能直接访问非static变量和方法。因为，非静态的变量是依赖于对象存在的，对象必须实例化之后，它的变量才会在内存中存在。反过来，非static方法可以调用static变量和方法。也可以通过对象访问static变量和方法。
3. **static方法必须被实现**，而不能是抽象的abstract。
4. 不能以任何方式引用this和super。

* 修饰代码块：

参照代码块，主要作用是初始化静态属性，可以节约资源。

什么情况下用：

1. 很多对象都使用到一个变量，而且一个里面有修改，其他都要用更新过的值（公共属性，只有一份）。
2. 方便访问变量时。例如为了方便方法的调用，Java API中的Math类中所有的方法都是静态的。

##### final

修饰类：不能被继承。final类中的成员变量可以根据需要设为final，但是要注意final类中的所有成员方法都会被隐式地指定为final方法。

修饰变量：如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。必须指定初始值，变量名习惯大写。

修饰方法：不能被子类覆写。

##### public static final

用来定义变量，表示一个全局常量。

#### this，super

this和super都是在类定义的时候使用的。

1. this的用法

1. 调用本类中的类变量和类方法，
2. 在构造函数中，用this调用本类中的其他构造方法，调用时要放在构造方法的首行。

|  |
| --- |
| **class** A {  **public int i** ;  **public** String **str** ;  **public** A(){}  **public** A(**int** i){  **this**.**i** = i ;  }  **public** A(**int** i, String str){  **this**(i);  **this**.**str** = str ;  } } |

1. 可以在类函数内返回当前对象的引用

|  |
| --- |
| **class** A {  **public** A(){}  **public** A fun(){  **return this**;  } } |

注意：因为this代表当前被实例化的对象，所以必须放在非静态（非static）方法里面。

2. super的用法

要使用被覆写的父类变量和父类方法的时候使用：super.变量/方法名（注意权限限制）

提示：

每个子类构造方法的第一条语句，都是隐含地调用super()，如果父类没有这种形式的构造函数，那么在编译的时候就会报错。

3. this和super的区别：

|  |  |  |
| --- | --- | --- |
| 区别 | this | super |
| 定义 | 表示本类的实例化对象 | 表示父类的实例化对象 |
| 使用 | this.属性、this.方法、this()、return this | super.属性、super.方法、super() |
| 调用构造 | 调用本类构造，必须写在首行 | 子类调用父类构造，必须放在首行 |
| 查找范围 | 先从本类中查找，找不到去父类找 | 直接在子类中查找父类 |
| 特殊 | 表示当前对象，如return this; | 无 |

#### 构造函数

每个类都有构造方法，若是没有定义，则默认提供，它把所有成员初始化为0

一旦定义了有参构造方法，默认构造方法就会被覆盖而失效，将不能在new对象时，调用无参构造（定义了带参的构造函数之后，要想使用默认构造，可以自己定义一个不带参构造）

类的构造器不是静态方法。运行时，使用到哪个类那个类才被加载，加载完之后才能用new新建对象。static是在类加载时执行，构造器在new对象时才会执行。

private修饰的构造无法在外部用new新建对象（用在单例模式）

子类不能继承父类的构造器。

子类构造器中，必然会默认先调用父类无参构造（先有老子，才有儿子，通过构造出生），也可以手动调用：super()。父类若没有无参构造，必须在子类构造中，使用super显式调用父类带参构造。

**范例：**说明子类实例化时，会先调用父类构造函数。

|  |
| --- |
| **abstract class** A { *//抽象类* **public** A (){  **this**.print();  }  **public abstract void** print() ; }  **class** B **extends** A {  **private int x** = 100 ;  **public** B (**int** x){  **this**.**x** = x ;  }  **public void** print(){  System.***out***.println(**x**) ;  } }  **public class** Client {  **public static void** main(String [] args) {  **new** B(30);  } } |
| **输出：0** |

构造代码块先于构造函数执行

**范例：**

|  |
| --- |
| **class** Demo {  **public** Demo(){  System.***out***.println(**"构造"**) ;  }  {  System.***out***.println(**"代码块"**) ;  } }   **public class** Client {  **public static void** main(String [] args) {  **new** Demo() ;  } } |
| **输出：**  代码块  构造 |

#### 封装，继承与多态

##### 封装

是将类的某些信息通过权限控制隐藏在类的内部，不允许外部程序直接访问，只能通过该类提供的方法来实现对隐藏信息的操作和访问。

##### 继承

主类也可以继承和被继承。

##### 多态

Java多态体现在两方面

方法多态：重载，覆写

对象多态：向上转型，向下转型

多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

* 1. 重载，覆写

重载和覆写方法名都相同。

重载：一个类中多态，参数列表必须修改，其他可改可不改。

覆写：父子类中多态，参数列表不能修改，子类中的方法访问修饰符比父类大或者相同，返回值和异常比父类小或者相同(即为父类的子类)。

其他区别：final或static修饰的方法可以重载，但不可以覆写（static修饰的方法覆写实为隐藏，见下文隐藏概念）。

* 1. 向上转型和向下转型

Java的引用变量有两个类型，一个是编译时类型，一个是运行时类型

编译时类型：由声明该变量时使用的类型决定

运行时类型：由该变量指向的对象类型决定

如果编译时类型和运行时类型不一致，会出现所谓的多态。即父类引用指向子类对象。亦即向上转型。

向上转型概念：

父类引用指向子类对象，如Father obj = new Son() ;

向下转型概念：

把之前向上转型的父类引用强制转化为子类引用，如

Father obj = new Son() ;

Son objS = (Son)obj ;

必须先有向上转型，才会有后面的向下转型。

向上转型之后，访问类属性，静态方法，非静态方法情况不同：

访问该引用的**类属性值**时，值由声明该变量类型的类决定**（隐藏），**无强制转换时子类要访问父类的属性使用super关键字；

访问该引用的**静态方法**时，方法行为也由声明该变量类型的类决定**（隐藏），**无强制转换时子类要访问父类的方法使用super关键字；

访问该引用的**非静态方法**时（多态），方法行为由new的对象实际类型所决定**（覆写）**。有覆写则调用子类覆写的方法；子类有而父类没有的方法则编译错误。如要使用子类特定方法，需要向下转型。

（所以日后开发中尽量不要在子类中定义父类中没有的方法，子类最好只是对父类定义好的方法进行完善）

作用：

向上转型：当父类有多个子类，调用子类中方法是以子类作为参数时，利用向上转型减少重复代码量；

向上转型：当需要使用子类有而父类没有的方法时，就需要先向下转型，再调用，不然会编译错误。

* 1. 隐藏

概念：父类和子类拥有相同名字的属性或者方法（ 方法隐藏只有一种形式，就是父类和子类存在相同的静态方法）时，父类的同名的属性或者方法形式上不见了，实际是还是存在的。当发生隐藏的时候，引用的声明类型是什么类，就调用对应类的属性或者方法，而不会发生动态绑定。

隐藏与覆写的区别：

子类和父类拥有相同名字的非静态方法，称为子类覆写父类方法，向上转型时访问子类的方法；

子类和父类的拥有相同名字的属性或者静态方法，称为隐藏，向上转型时访问父类的属性或静态方法；

RTTI(run time type identification，运行时类型检查)只针对覆盖，不针对隐藏：因为覆盖是动态绑定，是受RTTI约束的，隐藏不受RTTI约束。（在Java中，除了static方法和final方法，其他所有的方法都是动态绑定）。

隐藏与覆写的相同点：

方法隐藏时，权限修饰符和异常条件和覆写一样。

**范例：**说明隐藏和覆盖的区别。

|  |
| --- |
| **public class** Client {  **public static void** main(String[] args) {  Shape shape = **new** Circle();  System.***out***.println(shape.**name**);  shape.printType();  shape.*printName*();  } }  **class** Shape {  **public** String **name** = **"shape"**;  **public** Shape(){  System.***out***.println(**"shape constructor"**);  }  **public void** printType() {  System.***out***.println(**"this is shape"**);  }  **public static void** printName() {  System.***out***.println(**"shape"**);  } }  **class** Circle **extends** Shape {  **public** String **name** = **"circle"**;  **public** Circle() {  System.***out***.println(**"circle constructor"**);  }  **public void** printType() {  System.***out***.println(**"this is circle"**);  }  **public static void** printName() {  System.***out***.println(**"circle"**);  } } |
| **输出：**  **shape constructor**  **circle constructor**  **shape**  **this is circle**  **shape** |

#### 抽象方法，抽象类与接口

1. 抽象方法：

方法定义时以分号结尾，例如：public abstract sample();

不能与  
inal，static，native，synchronized共用。

2. 抽象类

**范例：**

|  |
| --- |
| **abstract class** A { *//抽象类* **private** String **name** ; *//抽象类可以包含属性和构造函数* **public** A (String name){  **this**.**name** = name ;  }  **public abstract void** funA () ; *//抽象类可包含可不包含抽象函数* **public void** funB (){ *//可包含可不包含非抽象函数* System.***out***.println(**name**) ;  } } |

3. 接口：

**范例：**

|  |
| --- |
| **abstract class** A {  **public abstract void** printA() ; } **interface** B {  **public static final** String ***MSG*** = **"全局常量"** ;   **public abstract void** printB () ; } **interface** C **extends** B{} *//接口继承接口单继承，关键字extends* **class** D **extends** A **implements** B,C{  *//类继承接口多继承，关键字implements，且可以同时继承类和抽象类，关键字extends* **public void** printA(){} *//实现父类中所有的抽象方法* **public void** printB(){} }  **public class** Client {  **public static void** main(String [] args) {  A a = **new** D() ;  a.printA() ;  a.printB() ; *//报错* B b = **new** D() ;  b.printB() ; *//正常* } } |

**范例：**Phone继承usb接口，Computer使用接口

|  |
| --- |
| **interface** USB {  **public void** install() ;  **public void** work() ; }  **class** Phone **implements** USB {  **public void** install(){}  **public void** work(){  System.***out***.println(**"work"**) ;  } }  **class** Computer {  **public void** plugin(USB usb){  usb.install() ;  usb.work() ;  } }  **public class** Client {  **public static void** main(String [] args) {  Computer com = **new** Computer() ;  com.plugin(**new** Phone()) ;  } } |

实战中接口作用：

指定操作标准

表示一种能力（标识接口）

将服务器端的远程方法视图暴露给客户端：分布式开发。

标记接口：没有任何方法的接口。

例：

public interface EventListener {}

作用：

1. 可以使用一个标记接口来建立一组接口的父接口。
2. **向一个类添加数据类型**
3. 抽象类和接口异同

4.1 语法层面

|  |  |  |
| --- | --- | --- |
| **区别点** | **抽象类** | **接口** |
| 类定义关键字 | abstract | interface。默认abstract修饰。 |
| 类访问权限 | 默认public，可以protected，但不可以default，private | 默认且只能是public |
| 类修饰符 | 不能是final，外部抽象类不能static，内部抽象类可以 | 不能是final，外部接口不能static，内部接口可以 |
| 组成 | 除了可以包含抽象方法外，和一般类无区别 | 只能有全局常量和抽象方法（JDK 1.8以前） |
| 构造函数 | 抽象类有构造函数 | 接口没有构造函数 |
| 属性 | 无限制 | 默认且只能是public static final，不可以用其他修饰 |
| 方法 | 抽象类可包含可不包含抽象方法。 JDK1.8之前，默认权限为protected； JDK1.8，默认权限为default。 | 默认abstract修饰。 JDK 1.8以前默认且必须是public的抽象方法； JDK 1.8时可以是public的抽象方法，也可以是default的普通方法（有方法体）； JDK 1.9时可以是private的 |
| 继承 | 可继承一般类（单继承，extends），抽象类（单继承，extends），接口（多继承，implements） | 只能继承接口（多继承，extends） |
| 被继承 | 子类若不是抽象类，必须实现所有父类抽象方法，若是抽象类无此限制 | 实现类若不是抽象类，必须实现接口中所有的方法，若是抽象类无此限制 |
| 实例化对象 | 不可以 | 不可以 |

4.2 设计层面上的区别

　　1）抽象类是对一种事物的抽象，即对类抽象，而接口是对行为的抽象（可以有多个行为，即实现多个接口）。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。举例，飞机和鸟是不同类的事物，但是它们都有一个共性，就是都会飞。那么在设计的时候，可以将飞机设计为一个类Airplane，将鸟设计为一个类Bird，但是不能将 飞行 这个特性也设计为类，因此它只是一个行为特性，并不是对一类事物的抽象描述。此时可以将 飞行 设计为一个接口Fly，包含方法fly( )，然后Airplane和Bird分别根据自己的需要实现Fly这个接口。然后至于有不同种类的飞机，比如战斗机、民用飞机等直接继承Airplane即可，对于鸟也是类似的，不同种类的鸟直接继承Bird类即可。从这里可以看出，继承是一个"是不是"的关系，而 接口 实现则是"有没有"的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现则是有没有、具备不具备的关系，比如鸟是否能飞（或者是否具备飞行这个特点），能飞行则可以实现这个接口，不能飞行就不实现这个接口。

　　2）设计层面不同，抽象类作为很多子类的父类，它是一种模板式设计。而接口是一种行为规范，它是一种辐射式设计。什么是模板式设计？最简单例子，大家都用过ppt里面的模板，如果用模板A设计了ppt B和ppt C，ppt B和ppt C公共的部分就是模板A了，如果它们的公共部分需要改动，则只需要改动模板A就可以了，不需要重新对ppt B和ppt C进行改动。而辐射式设计，比如某个电梯都装了某种报警器，一旦要更新报警器，就必须全部更新。也就是说对于抽象类，如果需要添加新的方法，可以直接在抽象类中添加具体的已实现方法，子类可以不进行变更；而对于接口则不行，如果接口进行了变更，则所有实现这个接口的类都必须进行相应的改动。

#### 内部类和匿名内部类

<http://www.cnblogs.com/dolphin0520/p/3811445.html> （了解特性即可，暂不深究原理）

内部类在类中新添加一个内部类成员，破坏了程序结构，但是内部类有自己的优点：

可以方便的访问外部类的私有操作，外部类也可以方便的访问内部类的私有操作。

内部类访问外部类私有属性方法：

可以直接用 属性 访问，也可以用 外部类.this.属性 的方式。

外部类访问内部类私有属性方法：

在外部类中实例化内部类，再调用实例对象属性

实例化内部类对象的方式：

外部类.内部类 引用名称 = new 外部类().new 内部类()

若内部类被static修饰，则

外部类.内部类 = new 外部类.内部类()

**范例：**

|  |
| --- |
| **class** Outer {  **private** String **outer** = **"outer 属性"**;  **class** Inner {  **private** String **inner** = **"inner 属性"**;  **public void** printInner(){  *//内部类中可以不加任何东西，直接访问外部类属性。外部类中必须实例化内部类才能访问内部类属性*  System.***out***.println(**outer**);  *//按照常理访问本类属性都要加this，所以可以用 外部类.this.属性 的方式：*  System.***out***.println(Outer.**this**.**outer**); }  }  **public void** printOuter() {  System.***out***.println(**new** Inner().**inner**);  } }  **public class** HelloWorld {  **public static void** main(String [] args) **throws** Exception{  **new** Outer().printOuter();  Outer.Inner ot = **new** Outer().**new** Inner(); *//在不同类上实例化内部类对象* ot.printInner();  } } |

**范例：**内部类用static修饰

|  |
| --- |
| **class** Outer {  **private** String **outer** = **"outer属性"**;  **static class** Inner {  **private** String **inner** = **"inner属性"**;  **public void** printInner(){  *//内部类static修饰之后，相当于一个外部类，所以不 new外部类对象的话只能访问外部类的static属性。此处可以直接使用静态变量outer* System.***out***.println(**new Outer().outer**);  }  }  **public void** printOuter() {  System.***out***.println(**new** Inner().**inner**);  } }  **public class** HelloWorld {  **public static void** main(String [] args) **throws** Exception{  **new** Outer().printOuter();  Outer.Inner ot = **new Outer.Inner()**; *//注意此处new对象使用括号位置* ot.printInner();  } } |

**范例：**方法中定义内部类

|  |
| --- |
| **class** Outer {  **private** String **outer** = **"outer shuxing"**;  **public void** fun(**final int** x){  **final** String str = **"fun inner class"**;  **class** Inner {  **public void** print(){  System.***out***.println(Outer.**this**.**outer**); //方法中内部类访问外部类私有属性  System.***out***.println(x);  System.***out***.println(str);  }  }  **new** Inner().print();  }  }  **public class** HelloWorld {  **public static void** main(String [] args) **throws** Exception{  Outer ot = **new** Outer();  ot.fun(1);  } } |

注意：

方法中定义的内部类，只能访问方法中用final修饰的参数和变量。

实例化内部类方式？

匿名内部类：

|  |
| --- |
| **interface** Message {  **public void** print(); }  **class** Demo {  **public static void** get(Message msg){  msg.print() ;  } }   **public class** Client {  **public static void** main(String [] args) {  Demo.*get*(   **new** Message(){ *//直接new接口* **public void** print(){} *//需要实现接口函数* }   ) ;  } } |

注意：

1. 匿名内部类中使用的外部局部变量为什么只能是final变量。

jdk1.8中，不用显式地将这些变量声明为final。编译器会自动加上final修饰

#### Object

所有的类默认隐式继承Object，也可以多此一举显示继承。

一般在新建类中覆写三个方法toString(), equals(), hashcode()。

有2个protected的方法：clone()和finalize()。

有3个final的方法（不可被覆写）：getClass(),notify(),wait()。

Object中的方法：

##### equals()

方法定义：public boolean equals(Object obj)

一般对象的比较会覆写，但是一旦覆写equals(),就必须同时覆写hashCode()：

|  |
| --- |
| **class** A {  **private int num** ;  **public** A(){}  **public** A(**int** num){  **this**.**num** = num ;  }   **public int** getNum() {  **return num**;  }   **public void** setNum(**int** num) {  **this**.**num** = num;  }   @Override  **public boolean** equals(Object obj){  *//三个判断：被比较对象是否为空，两对象是不是==，否则比较属性* **if**(obj == **null**){ *//排除与空值比较，否则会报NullPointerException异常* **return false** ;  }  **if**(**this** == obj){  **return true** ;  }  **if**(obj.getClass() == (**this**.getClass())){ *//排除不同类型的比较，否则会报ClassCastException异常  //Class对象比较可用equals()，可用==* A a = (A)obj ;  **if** (**this**.**num** == a.**num**) {  **return true** ;  }  }  **return false** ;  } } |

注意点：

为什么不用isinstanceof来进行类型判断？？？

*//B是A的子类，则b instanceof A = true* Object obj = **new** B();  
 A a1 = (A)obj;  
 System.***out***.println(obj.getClass() + **"\n"** + a1.getClass());

为什么覆写equals()就要覆写hashCode()？：

https://www.cnblogs.com/dolphin0520/p/3681042.html

String的equals源码

|  |
| --- |
| **public boolean** equals(Object anObject) {*//anObject是传进来的要进行比较的对象  //如果当前对象和传进来要进行比较的对象anObject是同一个对象（即地址相同eg同一辆汽车（只有一辆））则返回true* **if** (**this** == anObject) {  **return true**;  }  **if** (anObject **instanceof** String) {*//如果传进来的需要进行比较的对象anObject是String类的实例，则把anObject转换成String类型* String anotherString = (String) anObject;  *//value是一个private final char value[];  //String类的构造函数已经给value[]初始化了  //value.length代表原先要比较对象的字符个数* **int** n = value.length;  *//如果两者的字符个数不相等，意味着两者不可能相等，所以返回false；否则，依次遍历比较两者的每一个字符，若每一个字符都相等则相等，否则不想等* **if** (n == anotherString.**value**.length) {  **char** v1[] = value;  **char** v2[] = anotherString.**value**;  **int** i = 0;  **while** (n-- != 0) {  **if** (v1[i] != v2[i])  **return false**;  i++;  }  **return true**;  }  }  **return false**; } |

##### toString()

方法定义：public String toString()

Object的toString()源码：

|  |
| --- |
| **public** String toString() {  **return** getClass().getName() + **"@"** + Integer.*toHexString*(hashCode()); } |

已覆写此方法的常用类：String，Integer，Long，Date。

##### getClass()

方法定义：public final Class<?> getClass()

返回对象的运行时类, 不可重写，要调用的话，一般和getName()联合使用，例 a.getClass().getName();

##### hashCode()

方法定义：public int hashCode()

返回类对象的hashcode

不同的对象的hashcode值可能会相同。两个对象的hashcode值不等，必定是不同对象。

==与equals()的区别就在于此。

覆写equals()方法就要同时覆写hashCode()方法，覆写hashCode():

|  |
| --- |
|  |

##### clone()

方法定义：protected Object clone() throws CloneNotSupportedException

new出来的对象是全新的，有时候需要复制一个已被改变过（如属性值）的对象，就需要用到clone()方法。

方法定义：protected native Object clone() throws CloneNotSupportedException;（异常为不支持的克隆异常）

类必须继承Cloneable这个标识接口（没有任何方法，只表示一种能力）。clone 方法在 object 类中定义。再覆写Object里面的clone()才能实现克隆。但是Object类本身不实现Cloneable接口，所以直接调用Object类的clone()方法都会抛异常。而且，clone()是一个本地方法，这意味着它是由 c 或 c++ 或 其他本地语言实现的。

**范例：**基本实现

|  |
| --- |
| **class** A **implements** Cloneable{  **protected** Object clone() **throws** CloneNotSupportedException{  **return super**.clone();  } } **public class** Test {  **public static void** main(String[] args) **throws** Exception{  A a = **new** A();  A a1 = (A)a.clone();  } } |

##### notify()

方法定义：public final void notify()

唤醒在此对象监视器上等待的单个线程。

##### notifyAll()

方法定义：public final void notifyAll()

唤醒在此对象监视器上等待的所有线程。

##### wait()

方法定义：

public final void wait() throws InterruptedException

public final void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException

导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

##### finalize()

方法定义：protected void finalize() throws Throwable

Throwable包含Error和Exception，Error是程序不能处理的，但是此处只是为了强调finalize()方法的完整性，不管出现异常还是错误，都不会中断程序的执行。

没有引用指向的对象会被当做垃圾回收，调用finalize()方法是对象被回收之前最后做的一个操作。

|  |
| --- |
| **class** A {  @Override  **protected void** finalize() **throws** Throwable {  System.***out***.println(**"垃圾被处理前的最后操作"**);  **throw new** Exception(**"异常不影响程序执行"**);  } } **public class** Test {  **public static void** main(String[] args) {  A a = **new** A();  a = **null**;  System.*gc*();  } } |

#### java常用包

|  |  |  |
| --- | --- | --- |
| 包名称 | 说明 | 常见类和接口 |
| java.lang |  |  |
| java.lang.reflect |  |  |
| java.util | 工具包，包含常用类库，日期操作 |  |
| java.text | 文本处理类库 |  |
| java.sql |  |  |
| java.net | 网络编程 |  |
| java.io |  |  |
| java.awt |  |  |
| java.swing |  |  |
| java.applet |  |  |

**java.lang –**

接口：

Runnable

类：

Object

String

Math

System

Thread

java.lang.reflect –

反射机制的包

**java.util –**

接口：

类集框架

Collection

List

Map

类：

Date

**java.sql –**

**java.io –**

接口：

类：

File

FileInputStream

FileOutputStream

Reader

Writer

**java.net –**

**java.awt –**

**java.swing –**

#### 线程

##### 实现多线程

每一个主方法中至少启动两个线程：main和gc。

* 1. Thread类

**范例：**

|  |
| --- |
| **class** MyThread **extends** Thread{  @Override  **public void** run(){  **for**(**int** i = 0 ; i < 5 ; i++){  System.***out***.println(i);  }  } }  **public class** Test {  **public static void** main(String[] args){  MyThread mt1 = **new** MyThread();  MyThread mt2 = **new** MyThread();  mt1.start();  mt2.start();  } } |

* 1. Runnable接口。还是会用到Thread的start()方法，可以利用Thread的构造public Thread(Runnable target)。

**范例：**

|  |
| --- |
| **class** MyThread **implements** Runnable{  @Override  **public void** run(){  **for**(**int** i = 0 ; i < 5 ; i++){  System.***out***.println(flag + **":"** + i);  }  } }  **public class** Test {  **public static void** main(String[] args){  MyThread mt = **new** MyThread();  Thread t1 = **new** Thread(mt);  Thread t2 = **new** Thread(mt);  t1.start();  t2.start();  } } |

##### 多线程为什么使用start()而不是run()

run()只是简单调用线程类对象里面的方法，start()方法才能启动一个线程，一旦得到资源就可以直接运行run()。

##### Thread和Runnable区别

1. 继承Thread类后不能继承其他类（JAVA单继承局限），继承Runnable接口后还可以继承其他类和接口，程序可扩展性高
2. Thread继承了Runnable接口
3. Thread可以接受Runnable对象作为参数
4. 到最后运行都需要用到Thread类的start()方法
5. Thread实现多线程，需要new多个线程类对象，Runnable只需new一个，占用着同一个对象引用，所以可以更好实现类中的资源共享

**范例：**两种方式实现卖票功能。因为Thread继承自Runnable，所以可以在Thread的构造public Thread(Runnable target)中向上转型。

|  |  |
| --- | --- |
| **class** MyThread **implements** Runnable{  **private int ticket** = 5;  @Override  **public void** run(){  **for**(**int** i = 1 ; i <= 5 ; i++){  **this**.**ticket**--;  **if**(**ticket** >= 0) {  System.***out***.println(  **"某线程买完票之后的即时余票: "** + **this**.**ticket**  );  } } }  } **public class** Test {  **public static void** main(String[] args){  MyThread mt = **new** MyThread();  Thread t1 = **new** Thread(mt);  Thread t2 = **new** Thread(mt);  t1.start();  t2.start();  } } | **class** MyThread **extends** Thread{  **private int ticket** = 5;  @Override  **public void** run(){  **for**(**int** i = 1 ; i <= 5 ; i++){  **this**.**ticket**--;  **if**(**ticket** >= 0) {  System.***out***.println(  **"某线程买完票之后的即时余票: "** + **this**.**ticket**  );  } } }  } **public class** Test {  **public static void** main(String[] args){  MyThread mt = **new** MyThread();  Thread t1 = **new** Thread(mt); //向上转型  Thread t2 = **new** Thread(mt);   t1.start();  t2.start();  } } |

##### 线程的常用方法

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public Thread(Runnable target,String name) | 构造 | 接收Runnable接口子类对象来实例化线程类对象，同时设置线程名称 |
| public final void setName(String name) | 普通 | 设置线程名称，系统默认设置为Thread-0，Thread-1,Thread-2… |
| public final String getName() | 普通 | 获取线程名称 |
| public static Thread currentThread() | 普通 | 取得当前线程对象，一般使用myThread.currentThread().getName() |
| public static void sleep(long millis) throws InterruptedException | 普通 | 线程休眠（毫秒）。在定义线程类时使用，需要用try catch捕捉异常 |

##### 线程优先级

每个线程都有自己的优先级，一般用1-10表示，10最高，表示被执行的几率最高。

默认一个线程的优先级和创建它的线程优先级相同，首次创建默认优先级5。

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public static final int MAX\_PRIORITY | 常量 | 最高优先级，数值为10 |
| public static final int NORM\_PRIORITY | 常量 | 中等优先级，数值为5。主方法优先级为5。 |
| public static final int MIN\_PRIORITY | 常量 | 最低优先级，数值为1 |
| public final void setPrority(int newPriority) | 普通 | 设置线程优先级 |
| public final int getPriority() | 普通 | 取得线程优先级 |

##### 线程同步

一个线程进入一个对象的一个synchronized()方法后，其他线程不能进入此对象的其他方法。

方式一：同步代码块，对当前对象的代码块进行同步。

|  |
| --- |
| **class** MyThread **extends** Thread{  **private int ticket** = 5;  @Override  **public void** run(){  **for**(**int** i = 1 ; i < 10 ; i++){  **synchronized**(**this**) {  **if** (**ticket** > 0) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"卖票："** + **this**.**ticket**--);  }  }  }  } }  **public class** Test {  **public static void** main(String[] args){  MyThread mt = **new** MyThread();  Thread t1 = **new** Thread(mt);  Thread t2 = **new** Thread(mt);  t1.start();  t2.start();  } } |

方式二：同步方法，就是把代码块放进一个同步方法，synchronize放在返回值前面。

|  |
| --- |
| **class** MyThread **extends** Thread{  **private int ticket** = 5;  @Override  **public void** run() {  **for** (**int** i = 1; i < 10; i++) {  sale();  }  }  **public synchronized void** sale(){  **if** (**ticket** > 0) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"卖票："** + **this**.**ticket**--);  }  } } |

##### 线程死锁

关键点：

1. 资源只有一份
2. 一个进程对已获得的资源不放，再去请求下一个资源

|  |
| --- |
| **class** MyThread **implements** Runnable{  **private** String **string1**;  **private** String **string2**;  **public** MyThread (String string1,String string2){  **this**.**string1** = string1;  **this**.**string2** = string2;  }  @Override  **public void** run(){  **synchronized** (**string1**) {  **try**{  Thread.*sleep*(1000); *//加延时必死锁，不加有可能线程1全部执行完再执行线程2就不会构成死锁了* }**catch**(InterruptedException e){  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName() + **"锁住"** + **this**.**string1**);  **synchronized** (**string2**){  */\*  \*资源1已经被线程1锁了，线程1想再把资源2锁上，操作资源2。  \*在这同时  \*资源2也已经被线程2锁了，线程2想再把资源1锁上，操作资源1。  \*线程1把资源1锁了，等线程2把资源2放出来，线程2把资源2锁了，等线程1把资源1放出来，所以就构成了死锁  \*/* System.***out***.println(Thread.*currentThread*().getName() + **"锁住"** + **this**.**string2**);  }  }  } } **public class** Test {  **public static void** main(String[] args) {  String string1 = **"资源1"**;  String string2 = **"资源2"**;  MyThread mt1 = **new** MyThread(string1,string2);  MyThread mt2 = **new** MyThread(string2,string1);  **new** Thread(mt1,**"线程1"**).start();  **new** Thread(mt2,**"线程2"**).start();   } } |

##### 线程状态

5个：创建（New），就绪（Runnable），运行（Running），阻塞（Blocked），死亡（Dead）。

1. 创建（New）：new一个线程对象。但线程并没有运行，此时线程还不是活着的（not alive）；
2. 就绪（Runnable）：调用实例的start()方法，但要等CPU给分配到资源才会运行，此时线程是活着的（alive）；
3. 运行（Running）： 一旦获取CPU(被JVM选中)，线程就进入运行(running)状态，线程的run()方法才开始被执行；在运行状态的线程执行自己的run()方法中的操作，可能会因等待某种资源而阻塞，或者完成任务而死亡，此时线程活着（alive）；
4. 阻塞（Blocked）：通过调用join()、sleep()、wait()或者资源被暂用使线程处于阻塞(blocked)状态，此时线程活着（alive）；
5. 死亡（Dead）：线程的run()方法运行完毕或被中断或被异常退出就到达死亡(dead)状态。不具有继续运行的能力。对于一个处于Dead状态的线程调用start()方法，会出现一个运行期(runtime exception)的异常；此时线程活不是活着的（not alive）。

##### 其他

P311

1. object.wait()

当一个线程执行到wait()方法时，他就进入到一个和该对象相关的等待池(Waiting Pool)中，同时失去了对象的机锁—暂时的，wait后还要返还对象锁。当前线程必须拥有当前对象的锁，如果当前线程不是此锁的拥有者，会抛出IllegalMonitorStateException异常,所以wait()必须在synchronized block中调用。

1. object.notify() notifyAll()

唤醒在当前对象等待池中等待的第一个线程/所有线程。notify()/notifyAll()也必须拥有相同对象锁，否则也会抛出IllegalMonitorStateException异常。

#### 数据库操作

|  |
| --- |
| **import** java.sql.\*;  **public** **class** study{  **private** **static** **final** String *DBDRIVER* = "com.mysql.jdbc.Driver";  **private** **static** **final** String *DBURL* = "jdbc:mysql://localhost:3306/test1";  **private** **static** **final** String *DBUSER* = "root";  **private** **static** **final** String *DBPW* = "0000";    **public** **static** **void** main(String[] args) **throws** Exception{  Class.*forName*(*DBDRIVER*);  Connection conn = DriverManager.*getConnection*(*DBURL*,*DBUSER*,*DBPW*);  Statement stmt = conn.createStatement();  //创建数据库操作对象，开发中Statement一般不用  String sql1 = "insert into student (name) values ('Tom')";  **int** len = stmt.executeUpdate(sql1);  //执行数据库增加，删除，修改等更新操作使用  String sql2 = "SHOW TABLES;";  ResultSet res = stmt.executeQuery(sql2);  //执行查询操作使用  **while**(res.next()){  String name = res.getString("name");  //也可用索引res.getString(1)，索引从1开始  //常用方法：select后面不加\*，写上名字，后面使用索引  System.*out*.println(name);  }  conn.close();  }  } |
| **由于使用statement时，sql语句中出现 错误的时候会引起错误严重会导致漏洞，所以一般使用PreparedStatement** |

注意点:

报错看到里面有sql，一般都是sql语句出错。

#### 类集框架

Collection和Map的最大区别：

Collection设置数据的目的主要是为了输出，Map设置数据的目的主要是为了查找。

##### Collection接口

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类集框架 | | | | | |
| **Collection**  add(),remove(),iterator() | | | | **Map** put(key,value),get(key), **set<Map.Entry<KV>> EntrySet()** | |
| **List** （有序，可重复，扩展函数get(),set()） | | **Set** （无序，不重复，完全继承Collection无扩展） | | **HashMap** （key或value都可为null） | **Hashtable** （key或value不可为null） |
| **ArrayList** | **Vector** 1.stack（有push(),pop()方法，先进后出）父类 2.只有Vector有enumeration()方法 | **HashSet** （无序，不重复） | **TreeSet** （有序，不重复） 使用comparable接口里的compareTo()方法 |  |  |

单值保存（指每一次只保存一个对象）的最大父接口，一般很少用，多用子接口List和set。

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| **public boolean add(E e)** | **普通** | **向集合中保存数据** |
| **public Iterator<E> itreator()** | **普通** | **为Iterator接口实例化** |
| public void clear() | 普通 | 清空集合 |
| public boolean contains(Object o) | 普通 | 查询集合中是否包含指定对象 |
| public boolean remove(Object o) | 普通 | 删除对象，需要equals() |
| public boolean isEmpty() | 普通 | 判断集合是否为空集合 |
| public int size() | 普通 | 返回集合长度 |
| public Object[] toArray() | 普通 | 将集合以对象数组的形式返回 |
| default void forEach(Consumer<? super T> action) | 普通 | 继承自[Iterable](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)<E> |

###### 1.1 List接口

collection最常用的子接口。扩充的方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| **public E get(int index)** | **普通** | **取得指定索引位置上的对象** |
| **public E set(int index,E element)** | **普通** | **修改指定索引位置上的对象** |
| public ListIterator<E>listIterator() | 普通 | 为ListIterator接口实例化 |

List常用子类：ArrayList（90%使用），Vector(10%使用)，

* + 1. ArrayList(90%)

**范例：**ArrayList常用方法使用

|  |
| --- |
| **import** java.util.\*;  **public class** Test {  **public static void** main(String [] args){  List<String> all = **new** ArrayList<String>();  all.add(**"hello"**);  all.add(**"hello"**);  all.add(**"world"**);  System.***out***.println(all); **for**(**int** i = 0 ; i <= all.size(); i++){  System.***out***.println(all.get(i));  }  } } |
| [hello, hello, world] |

**范例：**自定义对象添加到ArrayList。

注意点：

*有使用到对象比较的方法的（例：contains(),remove()），类中一定要覆写equals()方法，不覆写的话，比较的只是地址,无法完成正常功能*

*Object自带equals参数是Object类型的，所以覆写一定要用Object类型接收参数*

*需要判断的1.判断是否空 2.判断是否相等== 3.判断是否是同一个类的对象（排除不同类对象，属性却一样的情况） 4.比较属性*

*Object类型没有Person的属性，所以要向下转型，以便下面的属性比较*

|  |
| --- |
| **class** Person{  **private** String **name**;  **private int age**;  **public** Person(){}  **public** Person(String name, **int** age){  **this**.**name** = name;  **this**.**age** = age;  }  @Override  **public boolean** equals(Object obj){  Person per = (Person)obj;  **if**(obj == **null**){  **return false**;  }**else if**(**this** == obj){  **return true**;  }**else if**(!(obj **instanceof** Person)){  **return false**;  }**else if**(**this**.**name**.equals(per.**name**) & **this**.**age** == per.**age**){  **return true**;  }**else**{  **return false**;  }  } }  **public class** HelloWorld {  **public static void** main(String [] args) **throws** Exception{  List<Person> list = **new** ArrayList<Person>();  Person person1 = **new** Person(**"zhangsan"**,20);  Person person2 = **new** Person(**"zhangsan"**,20);  list.add(person1);  *//因为覆写了equals()方法，所以比较的是内容而不是地址*  System.***out***.println(list.contains(person2));System.***out***.println(list.remove(person2));   } } |
| **true**  **true** |

new ArrayList ()和Lists.newArrayList()几乎一模一样, 唯一区别是Lists[.](http://guava-libraries.googlecode.com/svn/trunk/javadoc/com/google/common/collect/Lists.html#newArrayList())newArrayList[()](http://guava-libraries.googlecode.com/svn/trunk/javadoc/com/google/common/collect/Lists.html#newArrayList())会自动推导(不是"倒")尖括号里的数据类型 (其实是javac帮你做的)。

* + 1. Vector(10%)

与ArrayList区别在于：

1. ArrayList异步处理，Vector同步处理。相应的同步处理数据安全性相对较高，性能相对较差。
2. 只有Vector支持Enumeration，其他任何都不支持。
   * 1. LinkedList

常用方法：

LinkedList特有方法

addFirst()

addLast()

添加元素

getFirst()

getLast()

获取元素，但不删除元素，如果集合中没有元素，会出现NoSuchException//？？？

removeFirst()

removeLast()

也可以获取元素，但是元素会被删除，如果集合中没有元素，会出现NoSuchException

与ArrayList区别：

ArrayList通过下标随机访问元素块，但是插入、删除比较慢；

LinkedList 插入、删除和移动元素快比价快，但是通过下标访问元素快效率低。

###### Set接口

完全继承了collection，未做扩展。常用的子类：HashSet（90%使用）, TreeSet（10%使用）.

* + 1. HashSet(90%)

散列（无序）存放的子类

**范例：**

|  |
| --- |
| **public static void** main(String[] args){  Set<String> all = **new** HashSet<String>();  all.add(**"hello"**);  all.add(**"hello"**);  all.add(**"world"**);  System.***out***.println(all); } |
| **[**world,hello] //输出是无序，不重复的 |

* + 1. TreeSet(10%)

和HashSet比多了排序功能。数据添加到TreeSet对象中时，会自动对数据进行排序。

依靠Comparable接口的compareTo方法返回的数据来区分是否为重复数据。所以自定义类中有多个参数会出错：需要覆写compareTo方法，比较所有属性，被比较属性的顺序自己定义（int compareTo()继承自Comparable<T>接口）。

**范例：**使用TreeSet保存Person类(有3个属性，String name, int age,double eng,按照年龄，工资，名字排序)

|  |
| --- |
| **class** Person **implements** Comparable<Person>{  **private** String **name**;  **private int age**;  **private double eng**;  **public** Person(){}  **public** Person(String name, **int** age,**double** eng){  **this**.**name** = name;  **this**.**age** = age;  **this**.**eng** = eng;  }  @Override  **public** String toString(){  **return this**.**age** + **" "** + **this**.**eng** + **" "** + **this**.**name**;  }  @Override  **public int** compareTo(Person per){  **if**(**this**.**age** > per.**age**){  **return** 1;  }**else if**(**this**.**age** < per.**age**){  **return** -1;  }**else**{  **if**(**this**.**eng** > per.**eng**){  **return** 1;  }**else if**(**this**.**eng** < per.**eng**){  **return** -1;  }**else**{  **return this**.**name**.compareTo(per.**name**);  }  }  } }  **public class** HelloWorld {  **public static void** main(String [] args) **throws** Exception{  Set<Person> set = **new** TreeSet<Person>();  set.add(**new** Person(**"zhangsan"**,20,10.0));  set.add(**new** Person(**"zhangsan"**,20,10.1));  set.add(**new** Person(**"lisi"**,20,10.1));  set.add(**new** Person(**"zhangsan"**,20,10.0));  Iterator<Person> ite = set.iterator();  **while**(ite.hasNext()){  System.***out***.println(ite.next());  }  } } |

List和Set的区别：

List有序可重复，Set无序不重复。判断重复与否，依赖于Object的两个方法：

public int hashCode();

public boolean equals(Object obj);

先比较hashcode,一致的话再用equals()方法进行属性的依次判断。

##### Map接口

Map接口中的常用方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| **public V put(K key, V value)** | 普通 | **集合之中保存数据** |
| **public V get(Object key)** | 普通 | **根据key取得对应的value，key不存在返回null** |
| **public Set<Map.Entry<>K,V>> entrySet()** | 普通 | **将Map集合变为Set集合** |
| public Set<K> keySet() | 普通 | 取得全部的key |
| public Collection<V> values() | 普通 | 取得全部value |

###### Map常用子类

HashMap（90%），Hashtable（10%）

2.1.1 HashMap（90%）

**范例：**

|  |  |
| --- | --- |
| **public static void** main(String[] args) {  Map<Integer, String> map = **new HashMap**<Integer, String>();  map.put(1, **"张三"**);  map.put(2, **"张三"**);  map.put(3, **"李四"**);  map.put(3, **"王五"**);  map.put(**null**, **null**);  System.***out***.println(map); } | **public static void** main(String[] args) {  Map<Integer, String> map = **new Hashtable**<Integer, String>();  map.put(1, **"张三"**);  map.put(2, **"张三"**);  map.put(3, **"李四"**);  map.put(3, **"王五"**);  map.put(**null**, **null**);  System.***out***.println(map); } |
| **{null=null, 1=张三, 2=张三, 3=王五}** | **{3=王五, 2=张三, 1=张三}** |

注意点：

Map中的key不可以重复。键对象出现重复时，后来者的值会取代前者的值。值对象可以一样。

2.1.2Hashtable（10%）

HashMap与Hashtable比较与区别：

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | HashMap | Hashtable |
| 1 | 推出时间 | JDK1.2 | JDK1.0 |
| 2 | 性能 | 采用异步处理方式，性能较高 | 采用同步处理方式，性能较低 |
| 3 | 安全性 | 数据安全性差 | 数据安全性高 |
| 4 | 处理null | **键和值**都允许保存null | **键和值**都不允许保存null |

###### Map输出

每对键值对象保存到Map之中，是会被自动保存成为一个Map.Entry<K,V>对象，然后保存的。

Map.Entry常用方法：

|  |  |
| --- | --- |
| 方法名称 | 描述 |
| public K getKey() | 获取Map.Entry<K,V>对象的键 |
| public V getValue() | 获取Map.Entry<K,V>对象的值 |

**范例：**

|  |
| --- |
| **public static void** main(String[] args) {  Map<Integer,String> map = **new** HashMap<Integer,String>();  map.put(1,**"A"**);  map.put(2,**"B"**);  Set<Map.Entry<Integer,String>> set = map.entrySet();  Iterator<Map.Entry<Integer,String>> it = set.iterator();  **while**(it.hasNext()){  Map.Entry<Integer,String> mp = it.next();  System.***out***.println(mp.getKey() + **"->"** + mp.getValue());  } } |
| **1->A**  **2->B** |

一般情况，Map键为String类型，很少Integer，有时候用自定义的类对象作为键（非常少见），但是因为Map的键不能相同，插入值时会做比较，所以需要在类中覆写equals()和hashCode()。

##### 集合输出

**Iterator**（单向输出）hasNext(),next()

**ListIterator**（双向输出）hasNext(),next(),hasPrevious(),previous()

**Enumeration**(只在Vector中有)hasMoveElements(),nextElements()

**ForEach**

###### 3.1 Iterator接口（95%）：迭代输出

获取Iterator实例化对象：

Iterable接口有一个方法 public Iterator<E> iterator()；

Collection接口继承了Iterable接口，所以Collection及子孙类对象可直接使用此方法获得Iterator类对象。

Iterator接口中的常用方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public boolean hasNext() | 普通 | 是否有下一个数据 |
| public E next() | 普通 | 取下一个数据 |
| public void remove() | 普通 | 删数据 |

**范例：**

|  |
| --- |
| **public static void** main(String[] args) {  List<String> list = **new** ArrayList<String>();  list.add(**"one"**);  list.add(**"two"**);  Iterator<String> iter = list.iterator();  **while**(iter.hasNext()){  System.***out***.println(iter.next());  } } |

###### ListIterator接口（0.01%）：双向迭代输出

获取ListIterator实例化对象：

只能依靠List接口。List接口下有以下方法 public ListIterator<E> listIterator();

ListIterator是Iterator的子接口，对Iterator进行了扩展。

ListIterator接口中的扩展方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public boolean hasPrevious() | 普通 | 是否有上一个数据 |
| public E previous() | 普通 | 取上一个数据 |

注意，一定要使用过next()，才能使用previous()，不然下标在第一个，就没有前一个数据。

###### Enumeration接口（4.98%）：枚举输出

获取Enumeration实例化对象：

只能依靠Vector类。Vector类中有以下方法 public Enumeration<E> elements() ;

Enumeration接口只定义了2个方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public boolean hasMoreElements() | 普通 | 是否还有数据 |
| public E nextElements() | 普通 | 取下一个数据 |

###### 3.4 foreach（0.01%）不推荐使用

**范例：**基本使用

|  |
| --- |
| **public static void** main(String [] args) **throws** Exception{  List<String> list = **new** ArrayList<String>();  list.add(**"h"**);  list.add(**"e"**);  **for**(String str : list){  System.***out***.println(str);  } } |

常用实现HashMap,SortedMap的子类TreeMap

HashMap也用到了哈希码的算法，以便快速查找一个键，   
 TreeMap则是对键按序存放，因此有一些扩展的方法，比如firstKey(),lastKey()等，你还可以从TreeMap中指定一个范围以取得其子Map。

大多数情况下，从性能上来说ArrayList最好，但是当集合内的元素需要频繁插入、删除时LinkedList会有比较好的表现，但是它们三个性能都比不上数组，另外Vector是线程同步的。所以：   
如果能用数组的时候(元素类型固定，数组长度固定)，请尽量使用数组来代替List；   
如果没有频繁的删除插入操作，又不用考虑多线程问题，优先选择ArrayList；   
如果在多线程条件下使用，可以考虑Vector；   
如果需要频繁地删除插入，LinkedList就有了用武之地；   
如果你什么都不知道，用ArrayList没错。

LinkedList：

LinkedList不同于前面两种List，它不是基于数组的，所以不受数组性能的限制。

它每一个节点（Node）都包含两方面的内容：

1.节点本身的数据（data）；

2.下一个节点的信息（nextNode）。

所以当对LinkedList做添加，删除动作的时候就不用像基于数组的ArrayList一样，必须进行大量的数据移动。只要更改nextNode的相关信息就可以实现了，这是LinkedList的优势。

HashSet：

虽然Set同List都实现了Collection接口，但是他们的实现方式却大不一样。List基本上都是以Array为基础。但是Set则是在 HashMap的基础上来实现的，这个就是Set和List的根本区别。 HashSet的存储方式是把HashMap中的Key作为Set的对应存储项。看看 HashSet的add（Object obj）方法的实现就可以一目了然了。

public boolean add(Object obj) {

return map.put(obj, PRESENT) == null;

}

这个也是为什么在Set中不能像在List中一样有重复的项的根本原因，因为HashMap的key是不能有重复的。

LinkedHashSet：

HashSet的一个子类，一个链表。

TreeSet：

SortedSet的子类，它不同于HashSet的根本就是TreeSet是有序的。它是通过SortedMap来实现的。

Set总结：

Set实现的基础是Map（HashMap）

Set中的元素是不能重复的，如果使用add(Object obj)方法添加已经存在的对象，则会覆盖前面的对象

#### IO

##### File类

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public File(String pathname) | 构造 | 给定一个要操作文件的完整路径 |
| public File(File parent,String child) | 构造 | 给定要操作文件的父路径和子文件名称 |
| public boolean createNewFile() throws IOException | 普通 | 创建文件 |
| public boolean delete() | 普通 | 删除文件 |
| public boolean exists() | 普通 | 判断给定路径是否存在 |
| public File getParentFile() | 普通 | 获取指定路径的父路径 |
| public boolean mkdirs() | 普通 | 创建指定目录 |
| public String getName() | 普通 | 取得文件名称 |
| public boolean isDirectory() | 普通 | 判断给定路径是否是文件夹 |
| public boolean isFile() | 普通 | 判断给定路径是否是文件 |
| public boolean isHidden() | 普通 | 判断文件是否隐藏 |
| public long lastModified() | 普通 | 返回文件最后修改日期 |
| public long length() | 普通 | 返回文件大小，以字节为单位 |
| public boolean renameTo(File dest) | 普通 | 为文件重命名 |
| public File[] listFile() | 普通 | 将目录中所有文件以File对象数组的方式返回 |

**范例：**给定路径，父路径不存在则创建，子文件不存在则也创建，存在则删除

|  |
| --- |
| **public static void** main(String [] args) **throws** Exception{  File file = **new** File(**"D:"** + File.***separator*** + **"JavaIO"** + File.***separator*** + **"1.txt"**);  **if**(!file.getParentFile().exists()){  file.getParentFile().createNewFile();  }  **if**(file.exists()){  file.delete();  }**else**{  file.createNewFile();  } } |

注意点：

1. 分隔符在Windows上是"\"，Linux上是"/"，为了方便移植，java定义了常量  
    public static final String seperator

代表分隔符，常量不用大写表示的原因是File类在定义此规范之前就存在了

**范例：**列出指定目录下所有的文件

|  |
| --- |
| **public static void** main(String [] args) **throws** Exception{  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"**);  *print*(file); } **public static void** print(File file){  **if**(file.isDirectory()){  File [] fileList = file.listFiles();  **for**(**int** i = 0 ; i < fileList.**length** ; i ++){  **if**(fileList[i].isDirectory()){  *print*(fileList[i]);  }**else**{  System.***out***.println(fileList[i]);  }  }  } } |

**注意：**

1. 使用了递归调用

##### 字节输出流

字节：计算机中的存储单元，一个8位的二进制数，是一个具体的存储空间。

字符：抽象意义上的一个符号，如’1’,’中’，’a’,’$’,’¥’，’,’。

OutputStream抽象类，字节输出流

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public abstract void write(int b)  throws IOException | 普通 | 输出单个字节的数据 |
| public void write(byte [] b)  throws IOException | 普通 | 输出一组字节数据 |
| public void write(byte [] b, int off, int len)  throws IOException | 普通 | 输出部分字节数据 |

FileOutputStream,OutputStream的子类

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| publid FileOutputStream(File file)  throws FileNotFoundException | 构造 | 实例化FileOutputStream对象，主要用于新建数据 |
| public FileOutputStream(File file, boolean append)  throws FileNotFoundException | 构造 | 实例化FileOutputStream对象，主要用于追加数据 |

**范例：**追加文件

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  String str = **"hello world!\r\n"**;   OutputStream output = **new** FileOutputStream(file,**false**);  output.write(str.getBytes());  output.close(); } |

注意点：

(1)需要添加换行，可以在输出内容后加“\r\n”，

(2)必须关闭流（输入，输出都要记得关闭）

##### 字节输入流

InputStream抽象类，字节输入流

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public abstract int read() throws IOException | 普通 | 读取单个字节数据。每次执行会读取数据源的一个指定数据，如果发现已经到结尾返回-1。此处是抽象方法，但是FileInputStream会覆写 |
| public int read(byte[] b) throws IOException | 普通 | 读取多个字节数据。下个要读取的数据长度小于b的长度，返回数据长度，读完了返回-1 |
| public int read(byte[] b, int off, int len) throws IOException | 普通 | 读取指定多个字节数据 |

FileInutStream子类构造：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public FileInputStream(File file) throws FileNotFoundException | 构造 |  |
| public FileInputStream(String name) | 构造 |  |

**范例：简单**按照指定字节数组大小读取文本

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  InputStream input = **new** FileInputStream(file);  **byte** [] bc = **new byte**[1024];  **int** len = input.read(bc);  System.***out***.println(**new** String(bc, 0, len));  input.close(); } |

**范例：单个字节**循环读取文本**（开发中比较常用）**

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  InputStream input = **new** FileInputStream(file);  **byte** [] bc = **new byte**[1024];  **int** len = 0 ;  **int** tmp = 0 ;  **while**((tmp = input.read()) != -1){  bc[len] = (**byte**)tmp;  len++;  }  System.***out***.println(**new** String(bc, 0, len));  input.close(); } |

##### 字符输出流

Writer抽象类常用方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public void write(String) Throws IOExcetion | 普通 | 将字符串输出 |
| public void write(char[] cbuf) throws IOException | 普通 | 将字符数组输出 |
| public abstract void flush() throws IOException | 普通 | 强制清空缓存。字符输出流若不关闭，内容留在内存中，使用本方法可以强制把缓存内容输出 |
| public abstract void close() throws IOException | 普通 | 关闭输出流 |

**子类**FileWriter

**范例：**使用FileWriter子类，追加写入字符串

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  Writer writer = **new** FileWriter(file,**true**);  String str = **"hello"**;  writer.write(str);  writer.close(); } |

##### 字符输入流

Reader抽象类常用方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public int read() throws IOException | 普通 | 读取单个字符 |
| public int read(char[] cbuf) throws IOException | 普通 | 将内容读取到字符数组中，返回读入的长度 |
| public abstract void close() throws IOException | 普通 | 关闭输出流，子类FileReader实现此抽象方法 |

子类FileReader

**范例：**使用FileReader子类，读取文件

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  Reader read = **new** FileReader(file);  **char** [] data = **new char**[1024];  **int** len = read.read(data);  System.***out***.println(**new** String(data,0,len)); } |

字节流与字符流的对比：

字符流可以读中文，字节流会有乱码？？？

字符流对比字节流的好处就在于字符数据操作上，但只在Writer中体现，Reader没有直接操作字符串的方法。原因是输出大小肯定是程序所能承受之大小，但是输入大小不确定，可能超出程序能承受范围。

字节流针对于数据终端操作，如文件，而字符流是针对于缓存区(理解为内存)，所以字节流不关闭也能写到文件中，字符流需要关闭才能写到文件中，也可以通过flush()方法进行请执行的手工清空缓冲区。

处理各种数据都可以通过字节流完成，但是处理中文时使用字符流更好。

**范例：**实现文件拷贝

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file1 = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  File file2 = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"2.txt"**);  **byte** [] b = **new byte**[1024];  **int** tmp;  InputStream input = **new** FileInputStream(file1);  OutputStream output = **new** FileOutputStream(file2);  **while**((tmp = input.read(b)) != -1){  output.write(b,0,tmp);  }  input.close();  output.close(); } |

##### 转换流

有两种(都是字节流变成字符流)：

|  |  |  |  |
| --- | --- | --- | --- |
| **转换流操作类** | **父类** | **作用** | **构造** |
| OutputStreamWriter | Writer | 字节输出流变成字符输出流 | public OutputStreamWriter(OutputStream out) |
| InputStreamReader | Reader | 字节输入流变成字符输入流 | public InputStreamReader(InoutStream in) |

**范例1：**字节输出流变成字符输出流

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  OutputStream output = **new** FileOutputStream(file);  Writer writer = **new** OutputStreamWriter(output);  writer.write(**"hello"**);  *//output.close();* writer.close(); } |

注意：

有读写操作才需要close()方法吗，上例中output调用close()会报错????

**范例2：**字节输入流变成字符输入流

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  InputStream input = **new** FileInputStream(file);  Reader reader = **new** InputStreamReader(input);  **char** [] ch = **new char**[1024];  **int** tmp = 0 ;  **while**((tmp = reader.read(ch)) != -1){  System.***out***.print(**new** String(ch,0,tmp));  } } |

通过继承关系发现，所有字符数据的操作都是要进行转换的，真正传输的数据是不能有字符的，全部都是字节，字符只是在电脑中处理的结果。

##### 内存操作流

使用场景：必须发生IO操作，但是有不想有临时文件产生。

文件-》 程序 -》 内存

使用InputStream的子类FileInputStream 使用InputStream的子类ByteArrayInputStream

文件 《- 程序 《- 内存

使用OutputStream的子类FileOutputStream 使用OutputStream的子类ByteArrayOutputStream

**范例：**使用内存流完成字符串大小写转换操作

|  |
| --- |
|  |

##### 字符编码

GBK/GBK2312:

ISO 8859-1:

UNICODE:

UTF-8:

**范例：**查看系统环境属性，并输出乱码

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  System.*getProperties*().list(System.***out***); *//输出包含file.encoding=UTF-8* File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  OutputStream output = **new** FileOutputStream(file);  output.write(**"你好"**.getBytes(**"ISO8859-1"**));  output.close(); } |

##### 打印流

PrintStream和PrintWriter。相比较OutputStream和Writer的好处就是可以输出任意类型的数据。比如输出字符串Writer可以直接输出，OutputStream则需要将字符串转化字节数组；输出数字时需要将这些数据变成字符串，之后再变为字节数组输出。

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public PrintStream(File file) throws FileNotFoundException | 构造 | 接收File对象实例化PrintStream类 |
| public PrintStream(OutputStream out) | 构造 | 接收OutputStream对象实例化PrintStream类 |
| public PrintStream printf(String format, Object… args) | 普通 | 根据本地环境格式化输出，JDK1.5之后增加 |
| public void print(数据类型 b) | 普通 | 输出任意数据，此方法被重载多次 |
| pbublic void println(数据类型 b) | 普通 | 输出任意数据，此方法被重载多次 |

PrintWriter方法类似。

**范例：**使用printf()格式化输出

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"Desktop"** + File.***separator*** + **"1.txt"**);  OutputStream out = **new** FileOutputStream(file);  PrintStream tools = **new** PrintStream(out);  String name = **"张三"**;  **int** age = 20;  **double** score = 89.93894839;  tools.printf(**"姓名：%s, 年龄：%d, 成绩：%5.2f"**, name, age, score);  tools.close(); } |

注意：

String也可以格式化输出，但是输出的是字符串，进行数字的四舍五入还是需要BigDecimal完成。

##### System类-IO

System的3个IO常量：

public static final PrintStream err (PrintStream实例)

public static final PrintStream out (PrintStream实例)

public static final InputStream in （InputStream实例）

err和out的区别：

System.err输出不希望用户看到的错误，System.out输出希望用户看到的内容或错误。

**范例：**通过System.out为OutputStream对象实例化

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  OutputStream output = System.***out***;  output.write(**"hello"**.getBytes()); } |

注意点：

1. System.out是PrintStream实例化对象，PrintStream是OutputStream子类，所以可以利用System.out为OutputStream实例化；

2. 以上可以看出，OutputStream会根据实例化子类或对象不同，输出位置不同。

**范例：**由键盘输入数据，最多10个字节，末尾是中文可能会出现乱码

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  InputStream input = System.***in***;  **byte** [] b = **new byte**[10];  **int** len = input.read(b);  System.***out***.println(**new** String(b,0,len));  input.close(); } |

**范例：**由键盘输入任意多个字节，输入中文则全部是乱码

|  |
| --- |
| **public static void** main(String [] args) **throws** IOException {  InputStream input = System.***in***;  StringBuffer sb = **new** StringBuffer();  **int** tmp = 0 ;  **while**((tmp = input.read()) != -1){  **if**(tmp == **'\n'**){   **break**;  }  sb.append((**char**)tmp);  }  System.***out***.println(sb); } |

注意：

为什么输出用’\r\n’，输入用’\n’?????

##### 缓冲区操作

System.in是PrintStream类对象（常量），所以只能进行字节处理。遇到中文则容易出现乱码，这时候需要用到缓冲区。

两种类：

BufferedReader（最常使用，适合中文操作，且有readLine()方法，可以读取一行）

BufferedInputStream（不适合中文操作，且没有readLine()方法）

BufferedReader类

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public BufferedReader(Reader in) | 构造 | 接Reader对象实例化 |
| public String readLine() throws IOException | 普通 | 读取一行数据 |

**范例：**正确读取任意数据，包含中文

|  |
| --- |
| **public static void** main(String [] args){  InputStream input = System.***in*** ;  Reader reader = **new** InputStreamReader(input);  BufferedReader buf = **new** BufferedReader(reader);  **try** {  String str = buf.readLine(); *//readLine()方法可能抛IOException* System.***out***.println(str);  }**catch**(IOException e){  e.printStackTrace();  } } |

注意：

如上一点System.in所述，System.in是PrintStream的子类，PringStream是InputStream的子类，所以要使用BufferedReader类的readLine()方法，需要用InputStreamReader类转化。

不用close()?? 以上第6点同问。

**范例：**对输入数据验证是否为数字，是数字\*2输出，不是数字则提醒用户输入数字

|  |
| --- |
| **public static void** main(String [] args){  InputStream input = System.***in***;  Reader reader = **new** InputStreamReader(input);  BufferedReader buffer = **new** BufferedReader(reader);  **int** i = 0;  **boolean** flag = **true**; *//设定一个循环标记* String str = **""**;  **try** {  **while**(flag){  str = buffer.readLine();  **if**(str.matches(**"\\d+"**)){  System.***out***.println(**"结果："** + Integer.*parseInt*(str)\*2);  flag = **false**; *//输入为数字，则通过设定flag退出循环* }**else**{  System.***out***.println(**"请重新输入一个数字！"**);  }  }  }**catch**(IOException e){  e.printStackTrace();  } } |

##### Scanner

比起之前，更加方便的输入数据，并且最方便对输入数据验证。

Scanner类

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public Scanner(InputStream source) | 构造 | 从自定的字节输入流接收数据 |
| public boolean hasNext() | 普通 | 判断是否接收到内容 |
| public boolean hasNext(Pattern pattern) | 普通 | 判断接收的数据是否符合指定的正则标准 |
| public boolean hasNextXxx() | 普通 | 判断接收的内容是否为指定的数据类型 |
| public String next() | 普通 | 接收内容 |
| public String next(Pattern pattern) | 普通 | 接收内容，进行正则验证 |
| public int nextXxx() | 普通 | 接收指定的输入类型 |
| public Scaner useDelimiter(String pattern) | 普通 | 设置读取的分隔符 |

**范例：**使用Scanner普通输入

|  |
| --- |
|  |

**范例：**使用Scanner判断输入是否为double

|  |
| --- |
|  |

**范例：**验证输入的是否是日期类型（yyyy-MM-dd）

|  |
| --- |
|  |

**范例：**使用Scanner读取文件

|  |
| --- |
|  |

##### 总结

1. 程序输出数据使用PrintStream打印流，存在中文则用PrintWriter
2. 程序读取数据则Scanner最方便

##### 对象序列化和反序列化

ObjectOutputStream：对象序列化 程序->序列化对象

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public ObjectOutputStream(OutputStream out) throws IOException | 构造 | 接收父类对象实例化子类对象 |
| public final void writeObject(Object obj) throws IOException | 普通 |  |

**范例：**实现对象序列化

|  |
| --- |
| **class** Person **implements** java.io.Serializable{  *//必须实现Serializable标识接口（和Cloneable接口一样表示一种能力）* }  **public class** Client {  **public static void** main(String [] args) **throws** Exception{  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"1.txt"**);  Person per = **new** Person();  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(file));  oos.writeObject(per);  oos.close();  } } |

ObjectInputStream：对象反序列化 序列化对象->程序

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public ObjectInputStream(InputStream in) throws IOException | 构造 | 接收父类对象实例化子类对象 |
| public final void readObject(Object obj) throws IOException,ClassNotFoundException | 普通 |  |

**范例：**实现对象反序列化

|  |
| --- |
| **class** Person **implements** java.io.Serializable{  *//必须实现Serializable标识接口（和Cloneable接口一样表示一种能力）* }  **public class** Client {  **public static void** main(String [] args) **throws** Exception{  File file = **new** File(File.***separator*** + **"users"** + File.***separator*** + **"yzh"** + File.***separator*** + **"1.txt"**);  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(file));  Person per = (Person)ois.readObject(); *//接收对象* System.***out***.println(per);  ois.close();  } } |

#### 泛型

##### 基本使用

类中操作的 属性 或 方法的参数类型 不在定义时声明，而是在使用时动态设置。

**范例：**使用泛型

|  |
| --- |
| **class** Fan<T,P>{ //此时T,P两个属性还没确定  **private** T **m**;  **private** P **n**;  **public** Fan(T m, P n){  **this**.**m** = m;  **this**.**n** = n;  }; }  **public class** Test {  **public static void** main(String[] args){  Fan<String ,Integer> fan = **new** Fan<String, Integer>(**"a"**,1);//在使用时动态设置了T,P  } } |

泛型好处：

1. 类型安全。

2. 消除强制类型转换。

3. 潜在的性能收益。

**范例：**关于类型安全

|  |  |
| --- | --- |
| List list = **new** ArrayList(); list.add(**"a"**); list.add(1); | List<String> list = **new** ArrayList<String>(); list.add(**"a"**); list.add(1); *//此处报错* |

**泛型若省略类型，则默认是Object类型。**

重载时，泛型类型的改变无法重载，如

|  |
| --- |
| **public void** print(Message<String> msg){} **public void** print(Message<Integer> msg){} |
| Error:(60, 24) java: 名称冲突: print(Message<java.lang.Integer>)和print(Message<java.lang.String>)具有相同疑符 |

##### 通配符

实例化泛型类或者函数中形参是泛型类，就一定要定义泛型类类型，形参中可以用通配符替代。

泛型类做引用传递时，在函数形参中一旦确定了泛型类型，所有函数调用方的实参 就都 需要使用这种泛型类型，否则就会报错。而且，无法用Message<Object>接收所有泛型类型。

**范例：**

|  |
| --- |
| **public static void** main(String[] args){  List<Object> msg = **new** ArrayList<Integer>(); *//出错* } |
| **Error:(55, 28) java: 不兼容的类型: java.util.ArrayList<java.lang.Integer>无法转换为java.util.List<java.lang.Object>** |

所以只能用通配符。

**范例：**

|  |
| --- |
| **public static void** main(String[] args){  *print*(**new** Message<Integer>());  *print*(**new** Message<String>()); } **public static void** print(Message<?> msg){  *//类似于Message<?> msg = new Message<String>();*  System.***out***.println(msg); ***//因为泛型的类型还没确定，所以此处是无法修改泛型类型对象的属性的*** } |

##### 限定符

设置泛型上线：? extends 类,如? extends Number，表示?只能是Number或者是Number的子类，Integer等；

设置泛型下限：? super 类，如? super Number，表示?只能是Number或者是Number的父类。

|  |
| --- |
| **class** Message<T **extends** Number> { *//可以在类定义中限定* }  **public class** Test {  **public static void** main(String[] args){  *print*(**new** Message<Number>());  }  **public static void** print(Message<? **super** Number> msg){ *//也可以在方法中限定，****也可以同时加限定***System.***out***.println(msg);  } } |

##### 泛型方法

泛型也可以用在方法中，而且所在类并非必须为泛型类：**（注：泛型在返回值类型前面声明）**

|  |
| --- |
| **public** **class** Test {  **public** **static** <T> **void** print(T t){  *//T的类型在方法调用时定义*  }    **public** **static** **void** main(String[] args) {  String str = "hello";  *print*(str);  }  } |

##### 泛型接口

有两种泛型接口使用方式:

子类继续设置泛型

|  |
| --- |
| **interface** A<T>{  **public void** print(T a); } **class** B<T> **implements** A<T>{  **public void** print(T a){  System.***out***.println(**"a"**);  } } |

子类直接设置好具体的泛型类型

|  |
| --- |
| **interface** A<T>{  **public void** print(T a); } **class** B<String> **implements** A<String>{  **public void** print(String a){  System.***out***.println(**"a"**);  } } |

#### 枚举

之前需要使用public static final定义常量，现在可以用枚举定义常量集合。

枚举类中实际上每一个都是本枚举类的对象。

枚举不能继承与被继承

不能继承：任何枚举已经继承自java.lang.Enum，而Java是不支持多继承的；

不能被继承：采用enum声明后，该类会被编译器加上final声明（同String），故该类是无法继承的

枚举到底是多例设计模式

##### 基本使用

1. 定义颜色

|  |
| --- |
| **enum** Color{  ***RED***, ***GREEN***, ***BLUE*** }  **public class** Test {  **public static void** main(String[] args){  Color color = Color.***RED***;  System.***out***.println(color);  } } |

1. 利用switch进行判断：

|  |
| --- |
| **enum** Color{  ***RED***, ***GREEN***, ***BLUE*** }  **public class** Test {  **public static void** main(String[] args){  Color color = Color.***RED***;  **switch**(color){  **case *RED***:  System.***out***.println(**"红色"**);  **break**;  **case *GREEN***:  System.***out***.println(**"绿色"**);  **break**;  **case *BLUE***:  System.***out***.println(**"蓝色"**);  **break**;  **default**:  System.***out***.println(**"都不是"**);  **break**;  }  } } |

##### 方法

每个用enum定义的枚举类都继承了Enum类，Enum定义的方法：

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public final int ordinal() | 普通 | 取得枚举的序号 |
| public final String name() | 普通 | 取得枚举的名称 |
| toString() | 普通 | 返回枚举常量的名称，可重载 |
| compareTo(EnumTest.MON) | 普通 | 比较对象顺序，test在MON后，返回1，在 |
| getDeclaringClass().getName() | 普通 | 返回枚举类型相对应的class对象 |

枚举类可以使用values()方法取得枚举类中的全部对象，并使用foreach进行输出

|  |
| --- |
| **enum** Color{  ***RED***, ***GREEN***, ***BLUE ;*** *//每一个都是本枚举类的实例化对象*}  **public class** Test {  **public static void** main(String[] args){  **for**(Color color : Color.*values*()){  System.***out***.println(color.name());  }  } } |

##### 枚举其他定义

枚举中也可以定义属性，方法和构造方法。

|  |
| --- |
| **enum** Color{  ***RED***(**"红色"**), ***GREEN***(**"绿色"**) ; *//必须在枚举实例最后一项后面添加分号*  **private** String **title** ;  **private** Color(String title){  **this**.**title** = title;  }  **public** String toString(){  **return this**.**title**;  } }  **public class** Test {  **public static void** main(String[] args){  System.***out***.println(Color.***RED***);  } } |

注意点：

1. 枚举类中每一个定义的对象必须卸载枚举类首行，
2. 构造方法只能是private或default的，
3. 一旦定义了有参构造，在枚举类中声明每一个枚举类对象时，都要明确调用此构造，否则编译错误，

##### 枚举实现接口

|  |
| --- |
| **interface** A{  **public void** fun(); }  **enum** Color **implements** A{  ***RED***(**"红色"**), ***GREEN***(**"绿色"**) ;  **private** String **title** ;  **private** Color(String title){  **this**.**title** = title;  }  **public** String toString(){  **return this**.**title**;  }  **public void** fun(){}; *//实现接口中的方法* } |

##### 枚举中定义抽象方法

枚举类中有抽象方法，那么枚举类中的每一个枚举类对象都必须要实现抽象方法。

|  |
| --- |
| **enum** Color{  ***RED***{  **public void** fun(){}; *//在每一个本枚举类实例化对象中实现本类抽象方法* },  ***GREEN***{  **public void** fun(){};  };  **public abstract void** fun(); *//定义抽象方法* } |

#### Annotation

##### @Override

覆写。添加之后，若覆写有误，会在编译时直接提示用户。

##### @Deprecated

方法过期。表示紧随其后的一个方法过期了（但是删除此方法之前的代码就会出错），不建议使用。

##### @SuppressWarnings

压制警告信息。

#### 日期类

##### java.util.Date

表示特定的瞬间，精确到毫秒。很多的方法已经过时，不推荐使用，下面仅列出没有过时的：

|  |  |  |
| --- | --- | --- |
| 方法 | 类型 | 说明 |
| public Date() | 构造 | 实例化Date类对象，在底层调用System.currentTimeMillis()作为日期参数, 精确到毫秒 |
| public Date(long time) | 构造 | 将long型变成Date类对象，time是从1970年1月1日 00:00:00起的毫秒数 |
| public long getTime() | 普通 | 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此Date对象表示的毫秒数 |
| public void setTime(long time) | 普通 | 设置此 Date 对象，time是从1970年1月1日 00:00:00起的毫秒数 |
| public boolean before(Date date) | 普通 | Date对象之间的比较。调用此方法的Date对象在date之前返回true,否则返回false |
| public boolean after(Date date) | 普通 | Date对象之间的比较。调用此方法的Date对象在date之后返回true,否则返回false |
| public int compareTo(Date date) | 普通 | 调用此方法的Date对象和date相等时候返回0;之前则返回负数;之后则返回正数 |
| public String toString() | 普通 | 将 Date 对象转换为以下形式的String：  dow mon dd hh:mm:ss zzz yyyy，其中zzz 是时区。  示例：Thu Aug 16 14:20:36 CST 2018 |

##### java.text.DateFormat（抽象类）

实际开发使用较少。

##### java.text.SimpleDateFormat（格式化日期处理类）

|  |  |
| --- | --- |
| 继承关系 | |
| java.lang.Object  -java.text.Format  -java.text.DateFormat  -java.text.SimpleDateFormat | public class SimpleDateFormat extends DateFormat |

|  |  |  |
| --- | --- | --- |
| **主要方法** | **类型** | **说明** |
| public SimpleDateFormat(String pattern) | 构造 | 实例化SimpleDateFormat对象。  pattern可选为：年(yyyy),月(MM),日(dd),时(HH),分(mm),秒(ss),毫秒(SSS)。如new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS ")  或new SimpleDateFormat("yy年MM月dd日 hh时mm分ss秒") |
| public final String format(Date date) | 普通 | Date转为String |
| public Date parse(String str) **throws ParseException** | 普通 | String转为Date |

**范例：**将Date转为一定格式的日期String

|  |
| --- |
| **public static void** main(String[] args) {  Date date = **new** Date();  SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**);  String str = sdf.format(date);  System.***out***.println(str); } |

**范例：**将一定格式的日期String转为Date

|  |
| --- |
| **public static void** main(String[] args) **throws Exception**{  String str = **"2018-03-22 10:41:07.014"**;  SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**);  Date date = sdf.parse(str);  System.***out***.println(date); } |

##### java.util.Calendar

|  |  |
| --- | --- |
| 继承关系 | |
| java.lang.Object  -java.util.Calendar | public abstract class Calendar extends Object implements Serializable, Cloneable, Comparable<Calendar> |

用来设置和获取日期数据的特定部分，如小时，日，或者分钟，或在日期的这些部分加上或者减去值

**范例：**Calendar对象常用方法示例

|  |
| --- |
| **public static void** main(String[] args) {  *//获取抽象类Calendar类对象，默认使用当前时间格式化*  Calendar calendar = Calendar.*getInstance*();  System.***out***.println(calendar); *//输出calendar对象信息* System.***out***.println(calendar.get(Calendar.***YEAR***)); *//年* System.***out***.println(calendar.get(Calendar.***MONTH***)); *//月，第一个月为0，第二个月为1...* System.***out***.println(calendar.get(Calendar.***DATE***)); *//日* System.***out***.println(calendar.get(Calendar.***DAY\_OF\_MONTH***)); *//日，同Calendar.DATE* System.***out***.println(calendar.get(Calendar.***HOUR***)); *//时，12小时制* System.***out***.println(calendar.get(Calendar.***HOUR\_OF\_DAY***)); *//时，24小时制* System.***out***.println(calendar.get(Calendar.***MINUTE***)); *//分* System.***out***.println(calendar.get(Calendar.***SECOND***)); *//秒* System.***out***.println(calendar.get(Calendar.***DAY\_OF\_WEEK***)); *//星期，周日为1，周一为2...   //获取calendar中，当前常量所指定域的最小值，如当前月份最小值就是1-1=0*  System.***out***.println(calendar.getMinimum(Calendar.***MONTH***));  *//获取calendar中，当前常量所指定域的最大值，如当前月份最大值就是12-1=11* System.***out***.println(calendar.getMaximum(Calendar.***MONTH***));  calendar.set(2019,6-1,1); *//设置Calendar对象* calendar.add(Calendar.***MONTH***, 3); *//Calendar对象月份加3* calendar.add(Calendar.***MONTH***, -3); *//Calendar对象月份减3    //其他after(),before(),compare()用法与Date的一样* } |

时间操作：

https://blog.csdn.net/yu\_kang/article/details/78317707

jdk8:

https://www.cnblogs.com/comeboo/p/5378922.html

#### 常用类库

##### Runtime类

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public static Runtime getRuntime() | 普通 | 取得Runtime实例化对象。Runtime类是**单例模式**,单例模式一定会有一个static方法取得本类实例。 |
| public long maxMemory() | 普通 | JVM最大可用内存量 |
| public long totalMemory() | 普通 | JVM可用内存量 |
| public long freeMemory() | 普通 | JVM空间内存量 |
| public void gc() | 普通 | 手动垃圾回收 |

##### System类

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| public static void arrayCopy(Object 源数组,int 源数组开始点,Object 目标数组名称,int 目标数组开始点,int 目标数组长度) | 普通 | 数组拷贝。Object接收数组引用。复制大量数据时，建议使用此方法 |
| public static long currentTimeMillis() | 普通 | 当前时间（ms单位） |
| public static void gc() | 普通 | 手动垃圾回收，和Runtime中gc()等效 |

##### Random类

**范例：**产生10个0-100的随机数

|  |
| --- |
| **public static void** main(String[] args){  Random rd = **new** Random();  **for**(**int** i = 0 ; i < 10 ; i++){  System.***out***.println(rd.nextInt(101)); *//不大于101的正整数* } } |

##### Math类

定义了所有与数学有关的基本公式。此类中所有方法都是static的。

|  |  |  |
| --- | --- | --- |
| **方法** | **类型** | **说明** |
| static **long** round(**double** a) | 普通 | 四舍五入，原理是+0.5再向下取整，注：-16.1向下取整是-17 |
| static **int** round(**float** a) | 普通 | 四舍五入 |

##### Array类

继承关系

java.lang.Object

java.lang.reflect.Array

原型声明

public final class Array extends Object

Array类的方法全部都是静态方法，而且构造方法被private修饰（是不是想到了Math类 :-)），它们的作用就是提供静态的函数库

数组与List之间的相互转化

|  |  |  |
| --- | --- | --- |
| **常用方法** | **类型** | **说明** |
| private Array() | 构造 | 这意味着Array类不能被实例化 |
| public BigDecimal(int val) | 普通 |  |
| public BigDecimal(String val) | 普通 |  |
| public static Object get(Object array, int index) | 普通 |  |
| public static boolean getBoolean(Object array, int index) | 普通 |  |
| public static byte getByte(Object array, int index) | 普通 |  |
| public static char getChar(Object array, int index) | 普通 |  |
| public static double getDouble(Object array, int index) | 普通 |  |
| public static float getFloat(Object array, int index) | 普通 |  |
| public static int getInt(Object array, int index) | 普通 |  |
| public static long getLong(Object array, in index) | 普通 |  |
| public static short getShort(Object array, int index) | 普通 |  |
| public static Object newInstance(Class<?> componentType) | 普通 |  |
| public static Object newInstance(Class<?> componentType, int length) | 普通 |  |
| public static void set(Object array, int index, Object value) | 普通 |  |
| public static void setBoolean(Object array, int index, boolean z) | 普通 |  |
| public static void setByte(Object array, int index, byte b) | 普通 |  |
| public static void setChar(Object array, int index, char c) | 普通 |  |
| public static void setDouble(Object array, int index, double d) | 普通 |  |
| public static void setFloat(Object array, int index, flaot f) | 普通 |  |
| public static void setInt(Object array, int index, int i) | 普通 |  |
| public static void setLong(Object array, int index, long l) | 普通 |  |
| public static void setShort(Object array, int index, short s) | 普通 |  |

##### Arrays

继承关系

java.lang.Object

java.util.Arrays

1

2

原型声明

public class Arrays extends Object

1

构造方法

private Arrays()

1

和Array类一样，Arrays类同样不能被实例化

方法概览

/\*\* Returns a fixed-size list backed by the specified array. \*/

static <T> List<T> asList(T...a)

/\*\* Searchs a range of the specified array of ints for the specified value using the binary search algorithm. \*/

static int binarySearch(int[] a, int fromIndex, int toIndex, int toIndex, int key)

/\*\* Copies the specified range of the specified array into a new aarray. \*/

static int[] copyOf(int[] original, int newLength, int from, int to)

/\*\* Returns true if the two specified arrays of ints are equal to one another. \*/

static boolean equals(int[] a, int[] b)

/\*\* Assigns the specified int value to each element of the specified range of the specified array of ints. \*/

static void fill(int[] a, int fromIndex, int toIndex, int val)

/\*\* Sorts the specified range of the array into ascending order. \*/

static void sort(int[] a, int fromIndex, int toIndex)

/\*\* Returns a string representation of the contents of the soecified array. \*/

static String toString(int[] a)

上面列出的是一些常用的方法，其中有fromIndex..toIndex的，可以去掉（API方法已经重载，不用担心）

还有，使用二分查找Arrays.binarySearch(..)对数组搜索时，又一个大前提是：数组是已经排序好的（有序数组才能用二分查找）

以及，Arrays.sort()方法的几种常见用法要烂熟于心：

1）忽略大小写：sort(a, String.CASE\_INSENSITIVE\_ORDER)

2）反向排序：sort(a, Collections.reverseOrder())

另外，当Arrays.equals(..)、Arrays.toString(..)方法对多维数组无效时，考虑使用Arrays.deepEquals(..)、Arrays.deepToString(..)方法

##### BigDecimal

float和double只能用来进行科学计算或工程计算。在大多数的商业计算中，一般用java.math.BigDecimal类来进行精确计算。原因参照<https://www.cnblogs.com/jayworld/p/10107335.html>。

|  |  |  |
| --- | --- | --- |
| **常用方法** | **类型** | **说明** |
| public BigDecimal(double val) | 构造 | 不建议使用！原因可见System.out.println(new BigDecimal(2.3)) |
| public BigDecimal(int val) | 构造 |  |
| public BigDecimal(String val) | 构造 | 常用，其他类型如double，可使用toString()方法转化成String |
| public static BigDecimal valueOf(double val) | 普通 | double直接转化为BigDecimal |
| public static BigDecimal valueOf(long val) | 普通 | long直接转化为BigDecimal |
| public BigDecimal add(BigDecimal value) | 普通 | 加 |
| public BigDecimal subtract(BigDecimal value) | 普通 | 减 |
| public BigDecimal multiply(BigDecimal value) | 普通 | 乘 |
| public BigDecimal divide(BigDecimal value) | 普通 | 除。  如果除不尽会报错java.lang.ArithmeticException: … |
| public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode) | 普通 | 除。  divisor：除数，scale：小数点后保留位数，roundingMode：舍入模式，四舍五入采用 **ROUND\_HALF\_UP**  ROUND\_CEILING //向正无穷方向舍入  ROUND\_DOWN //向零方向舍入  ROUND\_FLOOR //向负无穷方向舍入  ROUND\_HALF\_DOWN //向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，向下舍入, 例如1.55 保留一位小数结果为1.5  ROUND\_HALF\_EVEN //向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，如果保留位数是奇数，使用ROUND\_HALF\_UP，如果是偶数，使用ROUND\_HALF\_DOWN  **ROUND\_HALF\_UP** //向（距离）最近的一边舍入，除非两边（的距离）是相等,如果是这样，向上舍入, 1.55保留一位小数结果为1.6  ROUND\_UNNECESSARY //计算结果是精确的，不需要舍入模式  ROUND\_UP //向远离0的方向舍入 |
| public BigDecimal setScale(int newScale, RoundingMode roundingMode) | 普通 | 求约 |

注意：

基本类型中，只有double和long，可以使用valueOf()方法转化成BigDecimal类型

BigDecimal都是不可变的（immutable）的，在进行每一步运算时，都会产生一个新的对象

**范例：**

|  |
| --- |
| **public static void** main(String [] args) {  BigDecimal bd1 = **new** BigDecimal(**"10"**);  BigDecimal bd2 = **new** BigDecimal(**"3"**);  *//加减乘除* BigDecimal addResult = bd1.add(bd2); BigDecimal subtractResult = bd1.subtract(bd2); BigDecimal multiplyResult = bd1.multiply(bd2); BigDecimal divideResult = bd1.divide(bd2); *//除不尽时会报错java.lang.ArithmeticException: …*  *//不整除时的”约等于”操作，可以避免报错*  BigDecimal bd3 = bd1.divide(bd2,2,RoundingMode.***HALF\_UP***); *//bd3 = 3.33*  *//保留小数位数和方式* BigDecimal bd4 = **new** BigDecimal(**"0.1234"**);  BigDecimal bd5 = bd4.setScale(3,RoundingMode.***HALF\_UP***); *////保留3位小数，且四舍五入，bd5 = 0.123* } |

##### Comparable接口

JAVA提供了两种比较器接口来进行对象排序：java.lang.Comparable和java.util.Comparator。

Comparable直接在类定义的时候就写好。Comparator在类已经定义完之后使用。

7.1 Comparable

**定义：**

|  |
| --- |
| **public interface** Comparable<T> {**public int** compareTo(T o); } |

String类实现了Comparable接口，所以可以直接进行字符串大小比较。

**范例：**使用比较器Comparable

|  |
| --- |
| **class** Person **implements** Comparable<Person>{  **private** String **name**;  **private int age**;  **public** Person(String name, **int** age){  **this**.**name** = name;  **this**.**age** = age;  }  @Override  **public int** compareTo(Person per){  **if**(**this**.**age** > per.**age**){  **return** 1;  }**else if**(**this**.**age** == per.**age**){  **return** 0;  }**else**{  **return** -1;  }  } }  **public class** Test {  **public static void** main(String[] args) **throws** Exception{  Person p1 = **new** Person(**"zs"**,20);  Person p2 = **new** Person(**"ls"**,30);  System.***out***.println(p1.compareTo(p2));  } } |

##### Comparator接口

**定义（部分）：**

|  |
| --- |
| **public interface** Comparator<T> {**int** compare(T o1, T o2);**boolean** equals(Object obj);  } |

**范例：**比较一个已存在的类，只需要添加以下一个类

|  |
| --- |
| **class** PersonCompartor **implements** Comparator<Person>{  @Override  **public int** compare(Person p1, Person p2){  **if**(p1.getAge() > p2.getAge()){  **return** 1;  }**else if**(p1.getAge() == p2.getAge()){  **return** 0;  }**else**{  **return** -1;  }  } } |

#### JSON

常用四种解析方法：JSON官方，GSON ，FastJSON ，jackson

1. JSON官方
2. GSON
3. FastJSON

<https://www.cnblogs.com/jayworld/p/10096486.html>

关键点：

|  |  |
| --- | --- |
| JSON String -> JSONObject | JSONObject jsonObject = JSON.parseObject(str); |
| JSONObject -> JSON String | String str = jsonObject.toJSONString(); 或者 String str = jsonObject.toString(); //JSONObject类已经覆写了toString()方法 |
| JSON String -> JSONArray | JSONArray jsonArray = JSON.parseArray(str) //注意：数组类型的JSON字符串不能用parseObject()方法转化成JSONObject |
| JSONArray -> JSON String | String str = JSONArray.toJSONString(); 或者 String str = JSONArray.toString(); // JSONArray类已经覆写了toString()方法 |
| JSON String -> javaBean | ClassName className = JSON.parseObject(str , new TypeReference<ClassName>(){}); |
| javaBean -> JSON String | String str = JSON.toJSONString(javaBean); |
| JSONObject -> javaBean | String str = JSONObject.toJSONString; //先转换为JSONString ClassName className = JSON.parseObject(str , new TypeReference<ClassName>(){}); //再JSONString -> javaBean |
| javaBean -> JSONObject | JSONObject jso = (JSONObject)JSON.toJSON(javaBean); |

**范例：**String和JSONObject，JSONArray，javaBean之间的转化

|  |
| --- |
| **public class** Test {  **public static void** main(String [] args) {  *//json字符串-简单对象型* String JSON\_OBJ\_STR = **"{\"studentName\":\"lily\",\"studentAge\":12}"**;  *//json字符串-数组类型* String JSON\_ARRAY\_STR = **"[{\"studentName\":\"lily\",\"studentAge\":12},{\"studentName\":\"lucy\",\"studentAge\":15}]"**;  *//复杂格式json字符串* String COMPLEX\_JSON\_STR = **"{\"teacherName\":\"crystall\",\"teacherAge\":27,\"course\":{\"courseName\":\"english\",\"code\":1270},\"students\":[{\"studentName\":\"lily\",\"studentAge\":12},{\"studentName\":\"lucy\",\"studentAge\":15}]}"**;   *//JSON字符串-简单对象型与JSONObject之间的转换* JSONObject jsonSimple = JSON.*parseObject*(JSON\_OBJ\_STR); *// JSON是JSONObject的父类* System.***out***.println(jsonSimple.getString(**"studentName"**));  System.***out***.println(jsonSimple.getIntValue(**"studentAge"**));  System.***out***.println(jsonSimple.getInteger(**"studentAge"**));   *//json字符串-数组类型与JSONArray之间的转换* JSONArray jsonArray = JSON.*parseArray*(JSON\_ARRAY\_STR);  *//遍历方式1* **for**(**int** i = 0 ; i < jsonArray.size() ; i++){ *//注意：集合长度用size()方法返回长度，数组和字符串长度用什么方法？* System.***out***.println(jsonArray.getJSONObject(i));  }  *//遍历方式2* **for**(Object obj : jsonArray){ *//此处不能用JSONObject obj : jsonArray* JSONObject jsb = (JSONObject)obj;  System.***out***.println(jsb.getString(**“studentName”**));  }   *//复杂json格式字符串与JSONObject之间的转换* JSONObject jsonComplex = JSON.*parseObject*(COMPLEX\_JSON\_STR);  String teacherName = jsonComplex.getString(**“teacherName”**);  Integer teacherAge = jsonComplex.getInteger(**“teacherAge”**);  JSONObject course = jsonComplex.getJSONObject(**“course”**);  JSONArray students = jsonComplex.getJSONArray(**“students”**);   Student student = JSON.*parseObject*(JSON\_OBJ\_STR,Student.**class**);   ArrayList<Student> students = JSON.*parseObject*(JSON\_ARRAY\_STR,**new** TypeReference<ArrayList<Student>>() {});  **for**(Student student : students){  }   Teacher teacher = JSON.*parseObject*(COMPLEX\_JSON\_STR,**new** TypeReference<Teacher>() {});  *//调用 TypeReference的类（如上行中Teacher类）需要有默认构造函数* } }  **class** Student {  *//…* } **class** Course {  *//…* } **class** Teacher {  **private** String **teacherName**;  **private** Integer **teacherAge**;  **private** Course **course**;  **private** List<Student> **students**;  *//…* } |

另外，FastJSON还可以直接给JSONObject赋值

|  |
| --- |
| JSONObject jso = **new** JSONObject(); jso.fluentPut(**"a"**,1).fluentPut(**"b"**,2); *//设置*  jso.fluentClear(); *//清空*  Map<String,Integer> map = **new** HashMap<String,Integer>(); map.put(**"a"**,1); map.put(**"b"**,1); jso.fluentPutAll(map); *//全部设置*  jso.fluentRemove(**"a"**); *//移除某个键值对* |

1. jackson

#### 正则

1. 常用正则符号：

|  |  |
| --- | --- |
| **字符匹配单个字符** | |
| a | 表示匹配字母a |
| \\ | 表示转移字符“\“ |
| \t | 表示转移字符”\t“ |
| \n | 表示转移字符“\n” |
| **一组字符** | |
| [a,b,c] | 表示可能是字母a，可能是b，可能是c |
| [^a,b,c] | 表示不是字母a，不是字母b，不是字母c |
| [a-zA-Z] | 表示全部字母中任意一个 |
| [0-9] | 表示任一数字 |
| **边界匹配** | |
| ^ | 表示一组正则的开始 |
| $ | 表示一组正则的结束 |
| **简写表达式** | |
| . | 表示任一字符 |
| \d | 表示任一数字，等价于[0-9] |
| \D | 表示任一非数字，等价于[^0-9] |
| \w | 表示任一字母，数字，下划线，等价于[0-9a-zA-Z\_] |
| \W | 等价于[^0-9a-zA-Z\_] |
| \s | 表示空格，如\n，\t等 |
| \S | 表示非空格 |
| **数量表示** | |
| 正则表达式? | 表示正则出现0或1次 |
| 正则表达式\* | 表示正则出现0，1或多次 |
| 正则表达式+ | 表示正则出现1次或多次 |
| 正则表达式{n} | 表示正则出现n次 |
| 正则表达式{n,} | 表示正则出现n次以上 |
| 正则表达式{n,m} | 表示正则出现n~m次 |
| **逻辑表示与或非** | |
| 正则表达式A正则表达式B | 表达式A后紧跟着B |
| 正则表达式A|正则表达式B | 表示出现表达式A或者表达式B |
| （正则表达式） | 将多个子表达式合成一个表示，作为一组出现 |

java.util.regex包中两个类Pattern类和Matcher类

Pattern类（定义并且编译正则的匹配模板），其中存在的方法：

|  |  |  |
| --- | --- | --- |
| **方法名称** | **类型** | **描述** |
| public String[] split(charSequence input) | 普通 | 字符串全拆分 |
| public String[] split(charSequence input, int limit) | 普通 | 字符串部分拆分 |

Matcher类（匹配应用），其中存在的方法：

|  |  |  |
| --- | --- | --- |
| **方法名称** | **类型** | **描述** |
| public boolean matches() | 普通 | 字符串匹配 |
| public String replaceAll(String replacement) | 普通 | 字符串全替换 |
| public String replaceFirst(String replacement) | 普通 | 字符串替换首个 |

String类对正则的支持，主要方法：

|  |  |  |
| --- | --- | --- |
| **方法名称** | **类型** | **描述** |
| public boolean matches(String regex) | 普通 | 与指定正则匹配 |
| public String replaceAll(String regex, String replacement) | 普通 | 替换满足正则条件的内容 |
| public String replaceFirst(String regex, Sring replaement) | 普通 | 替换满足正则条件的首个内容 |
| public String[] split(String regex) | 普通 | 按照指定正则全拆分 |
| public String[] split(String regex, int limit) | 普通 | 按照指定正则拆分成指定个数 |

可以看出String的部分方法和Pattern/Matcher中的方法一致。

**范例：**

将字符串中的字母消除

|  |
| --- |
| String result = string.replaceAll(**"[a-zA-Z]?"**,**""**); |

字符串按照数字拆分

|  |
| --- |
| String[] result = string.split(**"[0-9]"**); |

要求字符串格式aaaaab,在b之前可以有任意个a

|  |
| --- |
| string.matches(**"a\*b"**) |

验证一个字符串是否为整形数据，如果是，将其变成int型数据

|  |
| --- |
|  |

验证字符串是否为小数，如果是，将其变成double型数据，而后乘以5 注意转换问题

|  |
| --- |
|  |

输入一个字符串，若是按照格式“年-月-日 时：分：秒”，则转换成Date型数据

|  |
| --- |
|  |

一个用户名只能有字母数字下划线组成，长度6-15位

|  |
| --- |
|  |

验证电话号：51278734，01051238888，010-51344356，（010）34234777，（010）-37847838

|  |
| --- |
|  |

验证Email地址，地址用户名以字母数字下划线组成，不能以数字和.开头，域名只能是.com .cn .net

|  |
| --- |
|  |

判断字符串“姓名：年龄：成绩|姓名：年龄：成绩|。。。。。。”

并对上题做比较（Arrays.sort()，比较器）

|  |
| --- |
|  |

#### 异常

##### Error和Exception

继承关系图：

|  |  |
| --- | --- |
| java.lang.Object  -java.lang.Throwable  -java.lang.Error | java.lang.Object  -java.lang.Throwable  -java.lang.Exception |

* 1. Error

Error是Java运行环境的内部错误或者硬件问题，如虚拟机错误，栈溢出，内存溢出等，这种错误与程序本身无关，程序本身无法捕获和处理。是由JVM抛出的。常见Error：NotClassDeFoundError（类未定义错误），OutofMemoryError(内存溢出错误)，StackOverFlowError(栈溢出错误)。

1.2 Exception

Exception是程序问题。程序可以进行处理。分为Checked Exception和Unchecked Exception。

Checked Exception：检查型异常（非运行时异常）

直接继承自Exception。java认为检查型异常都可以被处理，所以必须使用try…catch进行显式捕捉与处理，否则编译不通过。如IOException，SqlException等。

Unchecked Exception：非检查性异常（运行时异常）

继承自Exception的子类RuntimeException。运行之前无法预知，运行之后才会出现，不需要手动catch,jam自动处理。如错误类型转换，数组越界，空指针等，具体如NumberFormatException(例，parseInt()方法会抛)，ClassCastException,NullPointerException,ArithmeticException,ArrayIndexOfBoundException等。

NoClassDefFoundError 和 ClassNotFoundException 有什么区别？

##### Java异常处理流程

1. 程序中产生异常，JVM自动根据异常类型实例化一个异常类对象，
2. 若有异常处理，会根据try内产生的异常类对象，与每一个catch进行匹配，有匹配成功的则根据catch内容进行处理，若是没有任何一个catch匹配，则交给JVM做默认处理（进行异常信息的输出，而后中断程序执行），
3. 若没有异常处理，则交给JVM做默认处理，
4. 无论是否有异常，Finally内代码都会执行。

##### 异常捕获与处理的基本格式示例

|  |
| --- |
| **try**{  **int** i = 1/0; }**catch**(ArithmeticException x){  x.printStackTrace(); }**catch**(Exception e){  e.printStackTrace(); //printStackTrace()方法打印出来的异常信息是最完整的 }**finally** {  System.***out***.println(**"i"**); } |

注意点：

* + - * 1. try不能同时没有catch和finally
        2. 下个catch捕获的Exception类型必须比上一个范围广
        3. 有了catch，finally并非强制性添加
        4. finally里面处理的变量必须是try catch模块之外声明的变量，不能是try里面声明的变量，为了保证finally里面能正常执行

##### throws

throws主要是用在方法定义上，表示此方法中不做异常处理，交给调用方处理。主方法main()中使用throws，表示交给JVM处理。例：**public void** function**()** **throws** Exception{}。注意位置。

##### throw

##### 手动抛出异常对象.

|  |
| --- |
| **try** {  **throw new** Exception(**"手动抛出异常对象"**); } **catch** (Exception e) {  e.printStackTrace(); } |

##### 自定义异常

1. 检查性异常类，需要继承 Exception 类，
2. 运行时异常类，需要继承 RuntimeException类。

**范例：**

|  |
| --- |
| **class MyException extends Exception{**  **public MyException(String msg){**  **super(msg);**  **}**  **}**  **public class TestDemo{**  **public static void main(String [] args){**  **throw new MyException(“自定义异常类”);**  **}**  **}** |

##### 常见异常

<http://www.cnblogs.com/jayworld/p/8495174.html>

#### 设计模式

波波微课设计模式：

依赖注入设计模式

单例模式

构造者模式

其中创建模式

适配器模式

桥接器设计模式

组合模式

装饰模式

门面模式

代理模式

六种结构模式

职责链模式

命令模式

解释器模式

迭代器模式

策略模式

观察者模式

模板方法模式

##### 工厂

**说明：**

新建一个工厂类，类中有函数可以根据传入的参数，新建相应的类并return给调用方。

**示例：**

|  |
| --- |
| **class** Apple {  **public** Apple(){  System.***out***.println(**"Eat one apple"**);  } } **class** Banana {  **public** Banana(){  System.***out***.println(**"Eat one banana"**);  } } **class** Factory {  **public static** Object factory(String cla){  **if**(**"apple"**.equals(cla)){  **return new** Apple();  }**else if** (**"banana"**.equals(cla)){  **return new** Banana();  }**else**{  **return null**;  }  } } **public class** Test {  **public static void** main(String [] args){  Object obj = Factory.*factory*(**"apple"**);  } } |

##### 单例

单例模式可能面试题：

1. 提前（eager）和延迟（lazy）初始化

2. 静态初始化模块

3. 多线程安全

4. 双重检查锁定

5. jvm/jit指令优化重排

6. 反射（reflection）

7. 序列化（serialization）

8. 克隆（clone）

单例1.0 提前（eager）初始化单例

|  |
| --- |
| **class** EagerSingleton{  //使用final防止被多次创建  **private static final** EagerSingleton ***INSTANCE*** = **new** EagerSingleton();   *//私有构造函数，防止被再次新建对象* **private** EagerSingleton(){}   **public** EagerSingleton getInstance(){  **return *INSTANCE***;  } } |

单例2.0 延迟（lazy）初始化单例

|  |
| --- |
| **class** LazySingleton{  **private static** LazySingleton *INSTANCE* ;   *//私有构造函数，防止被再次新建对象* **private** LazySingleton(){}   **public static** LazySingleton getInstance(){  //使用if判断防止被多次创建  **if**(*INSTANCE* == **null**){  *INSTANCE* = **new** LazySingleton();  }  **return** *INSTANCE*;  } } |

单例2.1 多线程安全单例

|  |
| --- |
| **class** Singleton{  **private static** Singleton *INSTANCE* ;   *//私有构造函数，防止被再次新建对象* **private** Singleton(){}   **public static synchronized** Singleton getInstance(){  **if**(*INSTANCE* == **null**){  *INSTANCE* = **new** Singleton();  }  **return** *INSTANCE*;  } } |

单例2.2 双重检查锁定单例

|  |
| --- |
| **class** Singleton{  **private static volalite** Singleton *INSTANCE* ;  //volalite可以保证不会再老版本jdk中得到半生不熟的对象   *//私有构造函数，防止被再次新建对象* **private** Singleton(){}   **public static** Singleton getInstance(){  **if**(*INSTANCE* == **null**){  **synchronized**(Singleton.**class**){ *//多线程锁定需要再了解！！！* **if**(*INSTANCE* == **null**){  *INSTANCE* = **new** Singleton();  }  }  }  **return** *INSTANCE*;  } } |

单例3.0 比尔 普夫单例

比较完善的单例方法，因为java内机制规定：

a. 内部类SingletonHolder只有在getInstance()方法第一次调用的时候才会被加载（实现了lazy），

b. 一个类的加载过程是线程安全的，即线程互斥的。

|  |
| --- |
| **class** Singleton{  **private static** Singleton *INSTANCE* ;   *//私有构造函数，防止被再次新建对象* **private** Singleton(){}   **private static class** Handle{  **private static final** Singleton ***INSTANCE*** = **new** Singleton();  }   **public static** Singleton getInstance(){  **return** Handle.***INSTANCE***;  } } |

单例4.0 枚举单例

|  |
| --- |
| **enum** Singleton{  ***INSTANCE***;   *//添加单例方法* **public void** method(){   } }  **public class** Client {  **public static void** main(String [] args) **throws** Exception{  Singleton.***INSTANCE***.method();  } } |

如何破坏单例：

反射破坏单例，其他

解决办法

单例应用案例：

Core Java

java.lang.Runtime

java.awt.Desktop

Spring容器 • Singleton Scope(per container)

##### 多例

**说明：**

单例和多例的核心：构造方法私有化

**示例：**

|  |
| --- |
| **class** Sex {  **private static final** Sex ***MALE*** = **new** Sex(**"男"**) ;  **private static final** Sex ***FEMALE*** = **new** Sex(**"女"**) ;  **private** String **title** ;  **private** Sex (String str){  **this**.**title** = str ;  }  **public static** Sex getInstance (String str){  **switch** (str){  **case "male"**:  **return *MALE*** ;  **case "female"**:  **return *FEMALE*** ;  **default**:  **return null** ;  }  } }  **public class** Client {  **public static void** main(String [] args) {  Sex sex = Sex.*getInstance*(**"male"**) ;  } } |

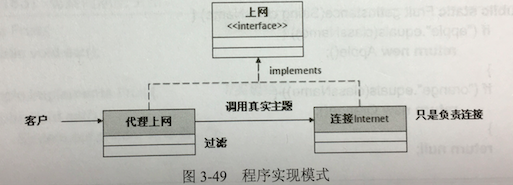
##### 代理

代理设计模式第二次讲

Java新特性00:25:00

单例模式01:07:00

范例：上网

****

|  |
| --- |
| **interface** Network {  **public void** browser() ; }  **class** Real **implements** Network {  **public void** browser(){} }  **class** Proxy **implements** Network {  **private** Network **network** ;  **public** Proxy(Network network){  **this**.**network** = network ;  }  **public void** check(){  System.***out***.println(**"检查网站合法性"**) ;  }  **public void** browser(){  **this**.check() ;  **this**.**network**.browser() ;  } }  **public class** Client {  **public static void** main(String [] args) {  Real real = **new** Real() ;  Proxy proxy = **new** Proxy(real) ;  proxy.browser() ;  } } |

**说明：**

**示例：**

|  |
| --- |
| **interface** Star {  **void** singStar();  **void** danceStar();  **void** filmStar();  **void** getMoney(); } **class** SingStar **implements** Star{  **public void** singStar(){  System.***out***.println(**"明星唱歌！"**);  };  **public void** danceStar(){};  **public void** filmStar(){};  **public void** getMoney(){}; } **class** StarProxy **implements** Star{  **private** Star **star**;  **public** StarProxy(Star star){  **this**.**star** = star;  }  **public void** singStar(){   };  **public void** danceStar(){};  **public void** filmStar(){};  **public void** getMoney(){  **star**.getMoney();  }; } **public class** Test {  **public static void** main(String [] args){   } } |

##### 模板

**说明：**

**示例：**

|  |
| --- |
| **abstract class** Action {   **public static final int *EAT*** = 1 ;  **public static final int *SLEEP*** = 2 ;  **public void** order (**int** flag){  **switch** (flag){  **case *EAT*** :  **this**.eat() ;  **break**;  **case *SLEEP*** :  **this**.sleep() ;  **case *EAT*** + ***SLEEP***:  **this**.eat() ;  **this**.sleep() ;  **break**;  }  }  **public abstract void** eat() ;  **public abstract void** sleep() ; }  **class** Pig **extends** Action{  **public void** eat(){  System.***out***.println(**"吃"**);  }  **public void** sleep(){  System.***out***.println(**"睡"**) ;  } }  **public class** Client {  **public static void** main(String [] args) {  Action a = **new** Pig() ;  a.order(Action.***EAT*** + Action.***SLEEP***);  } } |

##### 其他

#### 反射机制

一般情况先知道类，再用类实例化对象。反就是通过对象找到类。

先了解class类：

取得class对象的3种方式

方式一：使用Object的getClass()方法

|  |
| --- |
| A a = **new** A(); Class<?> cls = a.getClass(); cls.getName(); |

方式二：使用 类.class 取得

|  |
| --- |
| Class<?> cls = A.**class**; cls.getName(); |

方式三：使用Class类的static方法forName()，根据类名字获得类对象。（主要使用方式）

public static Class<?> forName(String className) throws ClassNotFoundException

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception{  String className = **"Person"**;  Class<?> cls = Class.*forName*(className); } |

获取之后，就可以通过反射实例化对象了

public T newInstance() throws InstantiationException, IllegalAccessException

**范例：**通过反射实例化无参类对象

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception{  String className = **"Person"**;  Class<?> cls = Class.*forName*(className);  Person per = (Person)cls.newInstance(); } |

之前的工厂模式：

|  |
| --- |
| **interface** Fruit{  **void** eat(); } **class** Apple **implements** Fruit{  **public void** eat(){  System.***out***.println(**"eat apple!"**);  } } **class** Factory{  **public static** Fruit getInstance(String str){  **if**(**"apple"**.equals(str)){  **return new** Apple();  }  **return null**;  } }  **public class** HelloWorld {  **public static void** main(String [] args){  Fruit fruit = Factory.*getInstance*(**"apple"**);  fruit.eat();  } } |

需要插入新的类时，必须修改工厂类，添加if条件。但是使用反射机制之后不需要修改工厂类：

|  |
| --- |
| **interface** Fruit{  **void** eat(); } **class** Apple **implements** Fruit{  **public void** eat(){  System.***out***.println(**"eat apple!"**);  } } **class** Factory{  **public static** Fruit getInstance(String className) **throws** Exception{  Class<?> cls = Class.*forName*(className);  Fruit fruit = (Fruit) cls.newInstance();  **return** fruit;  } }  **public class** HelloWorld {  **public static void** main(String [] args) **throws** Exception{  Fruit fruit = Factory.*getInstance*(**"Apple"**);  fruit.eat();  } } |

反射的深入应用：获取类构造，类方法，类属性。

p352

#### GC

垃圾就是没有任何栈内存指向的堆内存空间。

JAVA中GC有两种方式，一种是JVM不定期自动执行GC操作，还有一种是用户手动调用gc():

Runtime类的方法：public void gc()

System类的方法：public static void gc()，其实是间接调用了Runtime类的gc()方法。

#### 网络编程

#### 知识点

##### 1. [length,length(),size()区别](http://blog.csdn.net/sk880609/article/details/7524006)

length属性是针对数组的,比如list1.length

length()方法是针对字符串String的,比如string.length()

size()方法是针对泛型集合说的,如果想看这个泛型有多少个元素,就调用此方法来查看

##### 2. equals和 == 区别：

比较基本数据类型时，使用==，毫无疑问，equals只用于对象之间比较；

比较对象时，==和equals都是比较内存地址，地址相同就返回true；

但是有些类如String、Integer、Date等，覆写了Object父类的equals()方法，其中加入了对引用指向堆内容的比较(引用地址在栈中，被指向的对象在堆中)，若堆中内容一样，则也返回true。

##### 3. new

每new一次，就调用一次类的构造函数，新建一个类的对象。但是如果类里面只有private的构造函数，就不能new一个对象，除非有重载的public构造函数。

##### 4. Path和CLASSPATH区别

Path：指定系统可执行的程序路径。

CLASSPATH：java文件filename.java写完之后，运行需要两步：1）使用javac.exe编译：javac filename.java，在当前路径下生成一个编译好的文件filename.class； 2）使用java.exe加载编译好的filename.class（不带文件类型）：java filename。CLASSPATH就是指定被java.exe加载的 .class文件的查找路径。默认是当前路径（即SET CLASSPATH = .），可以用SET命令设置为别的路径。

##### get和post的区别

1. 作用

get用来获取、查询服务器上的信息；post是向服务器提交数据的一种请求，一般用来更新资源信息。

1. 提交后服务器端处理方式

get方式提交，服务器端使用request.QueryString获取变量的值；post方式提交，服务器端使用request.Form获取数据

1. 机制

get把参数数据队列加到URL中，如http://www.xxx.com?sessonid=db23434&name=hongten&age=20；post把表单内容和字段放在HTML HEAD中一起传送到指定URL。

1. 安全性

get安全性低，post安全性高。

1. 大小

特定浏览器对URL长度有限制，所以get参数有长度限制；post如果没有服务器处理能力限制，理论可以无限大。

##### 传参

传递常量，只在局部改变，出代码块依然为之前值；

传递引用，局部改变被传递引用的指向，出了代码块，被传递引用指向没改变，

局部改变实参引用指向的内存地址中的内容，内存中值就被改变，出了代码块，实参指向的内存地址中的内容就被改变了。

引用传递 实例化对象实参可被改变

|  |
| --- |
| **public static void** main(String[] args){  **int** i = 1;  System.***out***.println( i + 1 + **"结束"**); *//输出 2结束* System.***out***.println( **"结束"** + i + 1); *//输出 结束11* System.***out***.println( **"开始"** + i + 1 + **"结束"**); *//输出 开始11结束* System.***out***.println( **"结束"** + (i + 1)); *//输出 结束2* System.***out***.println( **"开始"** + (i + 1) + **"结束"**); *//输出 开始2结束* } |

##### 内存分配

|  |  |
| --- | --- |
| 栈内存空间 | 1.保存基本数据类型的值；  2.保存类的实例，即指向堆区对象的引用（指针）。 |
| 堆内存空间 | 保存每个对象的具体属性内容（包括常量，JVM为每个已加载的类型维护一个常量池，包含了这个类中用到的常量的一个有序集合。池中的数据和数组一样通过索引访问。）（不包括方法） |
| 全局数据区 | 存放static定义的静态成员。 |
| 全局代码区 | 保存所有的方法定义 |
| 寄存器 | JVM内部虚拟存储器，程序不可控制。 |

数据段

存放static定义的静态成员

堆

栈

……

内存

对象

x = 10;

i = 0;

常量池

代码段

硬盘

##### 算法

冒泡排序法：最多排n(n-1)/2次。第一轮n-1次，第二轮n-2次……，一共1+2+3+…+n-1=n(n-1)/2次。

二叉树：

##### java保留几位小数的做法

##### 数据转换

1. char与String互转

char转String：用String的构造函数或valueOf()方法

String str = new String(char[] value);

String str = new String(chat[] value,int offset,int count)；

String str = String.valueOf(char c);

String str = String.valueOf(char[] c);

String str = String.valueOf(char[] c, int offset, int count);

String转char：

char c = str.charAt(int index);

char[] c = str.toCharArray();

1. byte与String互转

byte转String：用String的构造函数

String str = new String(byte[] bytes);

String str = new String(byte[] bytes, int start, int length);

String转byte：

byte[] b = str.getBytes();

byte[] b = str.getBytes(String charsetName)；（必须捕捉throws UnsupportEncodingException）

1. int,long,float,double,boolean与String互转方法一样。

基本数据类型转成String只有一个String的valueOf()方法，

String转成基本数据类型可以用valueOf()和parseInt()。

int -> String

方法一 String str = String.valueOf(int i);（使用String的静态方法）

方法二 String str = i+””;(产生两个String，不推荐)

String -> int

方法一 int i = Integer.valueOf(String str).intValue();(先转为Integer对象，再转为int)

方法二 int i = Integer.valueOf(String str);(先转为Integer对象，再自动拆箱为int)

方法三 int i = Integer.parseInt(String str);(直接转为int。 注：Character没有此方法。)

Integer转String：

String str = integer.toString();（基本封装类都覆写了toString()方法）

String转Integer：

Integer integer = Integer.valueOf(String s);

1. 基本类型之间转换

由低转到高自动转换；由高转到低需要强制类型转换，且**可能导致溢出或者精度损失。**

低🡪高：**byte,short,char—> int —> long—> float —> double**

范例：由低转到高

char c1='a';//定义一个char类型

int i1 = c1;//char自动转换为int

范例：由高转到低，会导致精度损失

(int)1.9f == 1;

（int）-1.9 == -1;

注意点：浮点型到整数的转换时通过舍弃小数得到，而不是四舍五入。

##### for循环括号中间用；分隔

##### if else的使用

if后括号中，比较相同使用“==”

在返回不为空的函数中，if else的使用：

|  |  |  |
| --- | --- | --- |
| **错误：** | **正确：** | **正确：** |
| **public** String forTest (**int** i){  **if**(i == 1){  **return "yes"**;  }**else if**(i == 0){  **return "no"**;  } } | **public** String forTest(**int** i){  **if**(i == 1){  **return "yes"**;  }**else if**(i == 0){  **return "no"**;  }**else**{  **return "default"**;  } } | **public** String forTest(**int** i){  **if**(i == 1){  **return "yes"**;  }**else if**(i == 0){  **return "no"**;  }  **return "default"**; } |

##### isInstance和instanceof的区别

instanceof ：

实例 instanceof 类 返回boolean类型本类或子类对象才返回true ？？？

isInstance: 自身类.class.isInstance(自身实例或子类实例)  返回true

**范例：**

|  |
| --- |
|  |

instanceof: 自身实例或子类实例 instanceof 自身类   返回true

**范例：**

|  |
| --- |
| **if** (a **instanceof** A){  System.***out***.println(**"true"**); } |

以下情况返回true

isInstance和instanceOf的区别：

isInstance: 自身类.class.isInstance(自身实例或子类实例)  返回true

例：

|  |
| --- |
| String s = new String(“abc”);  System.out.println(String.class.isInstance(s)); //如果s是调用isInstance方法的类或接口（此例中为String）的实例，返回true |

instanceof: 自身实例或子类实例 instanceof 自身类   返回true

例：

|  |
| --- |
| String s = new String(“abc”);  if (s instanceof String){  System.out.println(“true”);  } |

##### lambda表达式

// 1. 不需要参数,返回值为 5

() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值

x -> 2 \* x

// 3. 接受2个参数(数字),并返回他们的差值

(x, y) -> x – y

// 4. 接收2个int型整数,返回他们的和

(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)

(String s) -> System.out.print(s)

##### 装箱和拆箱

**范例：**装拆箱易出错点

|  |
| --- |
| **public void** fun(){  Object obj = 10.1 ;  **double** d = (Double)obj ; *//此处不向下转型会出错，obj要转为double，要先向下转型到Double类型，再拆箱！* } |

#### 基本语法

##### 算术注意点

1. 相除

int i = 5/3;

**i = 1; int不保留小数。**

1. ++a与a++的区别

前缀自增自减法(++a,--a): 先进行自增或者自减运算，再进行表达式运算

后缀自增自减法(a++,a--): 先进行表达式运算，再进行自增或者自减运算

思考：

|  |
| --- |
| **public static void** main(String[] args) {  **int** n = 3;  **while**(n-- != 0){  System.***out***.println(n);  } } |

##### if else

|  |
| --- |
| **public static** String print(**int** i){  **if**(i == 0){  **return "false"**;  }**else if**(i == 1){  **return "true"**;   }**else**{ *//函数返回为空时，可以没有else；不为空时，没有else则需要加“return 返回值类型”*  **return "null"**;  } } |

##### while

while(){

}

do{

}while();

##### for

1. for( ; ; ){

} //所有已学语言for循环括号中都是 ;

1. 增强for

for( : ){

}

例，遍历数组：

public static void main(String[] args) {

double[] myList = {1.9, 2.9, 3.4, 3.5};

// 打印所有数组元素

for (**double element: myList**) {

**//只需数组名，不加括号**

System.out.println(element);

}

}

##### 条件运算符(三目)（？：）

例：

public class Test {

public static void main(String args[]){

int a , b;

a = 10;

b = (a == 1) ? 20: 30;

System.out.println( "Value of b is : " + b );

}

}

以上实例编译运行结果如下：

Value of b is : 30

##### switch

|  |
| --- |
| **public static void** main(String[] args){  **int** i = 0 ;  **switch**(i){ **case** 0: System.***out***.println(**"false"**);  **break**; **case** 1:  System.***out***.println(**"true"**);  **break**;  **default**: System.***out***.println(**"null"**);  **break**;  } } |

注意点：

1. switch被判断数据类型可以是：四种整形类型byte,short,char,int,字符串String和枚举类型enum，不支持基本类型包装类，但是可以自动拆箱成基本类型，
2. case后面只能是常量（可以是运算表达式），但不能是变量，已被赋值的变量也不行，
3. 若是case都没加break，则从匹配的case开始，执行后续每一个case里面的内容；若匹配case后没break，则执行匹配及后续每一个case里面的内容，
4. 若是有匹配case，则default里面内容不执行。

##### 源文件声明规则

1. 如果源文件在某package中，package语句放在源文件首，
2. 一个源文件只能有一个public类，但可以有多个非public类，
3. 源文件名称应和public类名一致，例Example.java，
4. 如果源文件包含import语句，那么应该放在package语句和类定义之间。
5. 在同一源文件中，不能给不同的类不同的包声明。

##### 命令行参数

public class CommandLine {

public static void main(String args[]){

}

}

在命令行中输入：java CommandLine hello world, 则arg[0] = hello, arg[1] = world。

##### 可变参数

可变参数中，定义的虽然不是数组，但是会按照数组方式进行操作的。

范例：

|  |
| --- |
| public void fun(int ... data){  for(int j = 0; j <= data.length; j ++){  System.out.println(data[j]);  }  } |

##### 命名问题

1. Java标识符（类名，变量名，方法名）由字母、数字、下划线、$组成，应以 字母/ ”$” /”\_” 开头
2. 类名一般单词首字母大写，例MyFirstJavaClass
3. 方法名一般第一个单词首字母小写，其他单词首字母大写，例myFunction.
4. 变量命名和方法名一样，区别在于有没有括号，
5. 大小写敏感

##### 注释

单行：

//

多行：

/\*……

\*……

\*/

##### 数据类型

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | 字节 | 位数 | 最小值 | 最大值 | 例 |
| 基本数据类型(  只包含具体的数据，不牵扯内存的关联) | 数值型 | 整型 | byte | 1 | 8 | -128（-2^7） | 127（2^7-1） | byte a = 100，byte b = -50 |
| short | 2 | 16 |  |  | short s = 1000，short r = -20000 |
| int | 4 | 32 |  |  | int a = 100000, int b = -200000 |
| long | 8 | 64 |  |  | long a = 100000L，Long b = -200000L |
| 浮点型 | float | 4 | 32 |  |  | float f1 = 234.5f |
| double | 8 | 64 |  |  | double d1 = 123.4 |
| 布尔型 | | boolean |  | 1 |  |  | boolean one = true |
| 字符型 | | char | 2 | 16 |  |  | char letter = ‘A’ |
| 引用数据类型 |  | | 类，数组，接口 |  |  |  |  | 引用类型指向一个对象  Site site = new Site("Runoob") |

习惯用法：

1. 整数永恒使用int，小数永恒使用double

2. 数据传输和字符转码过程使用byte

3. 日期或者文件大小用long

4. 逻辑关系用boolean

开发中，小数永恒使用double, 如果要使用float数据类型需要添加f/F

例：

float f = 10f ;

Java bug:

3 \* 10.1 //30.2999999997

int转为long：

方式1：前面加”(long)”

方式2：后面加”L”

范围大的数据类型转为小的，不强制转换会报错。但用强制转换会损失精度。

类中的非引用变量会初始化为默认初始值，引用变量初始化为null（boolean变量默认会被初始化为false）。

##### 规则输出printf

%d 十进制整数

%.2f保留两位小数

%s 字符串

##### 变量类型

局部变量，类变量（静态变量），成员变量（非静态变量）。

**类变量和实例变量的区别在于：类变量是所有对象共有，其中一个对象将它值改变，其他对象得到的就是改变后的结果；而实例变量则属对象私有，某一个对象将其值改变，不影响其他对象。**

|  |  |
| --- | --- |
| 局部变量 | 1. 声明在方法，构造方法或者语句块中 2. 局部变量没有默认值，所以局部变量被声明后，必须经过初始化，才可以使用 3. 局部本身就是一个访问范围限定，不许加任何访问修饰符 4. 在栈上分配 |
| 成员变量（  实例变量  ） | 1. 声明在一个类中，但在方法、构造方法和语句块之外 2. 变量的值可以在声明时指定，也可以在构造方法中指定，不可以在一条语句中声明，下一条语句中赋值。但可以不赋值，取默认值 3. 具有默认值。**数值型变量默认值是0**，布尔型默认值是false，引用类型默认值是null。 4. 成员变量可以声明在使用前或者使用后 5. 访问修饰符可以修饰成员变量，一般情况下应该把成员变量设为私有 6. 在堆上分配 |
| 类变量  （  静态变量  Static  ） | 1. 声明在一个类中，但在方法、构造方法和语句块之外 2. 静态变量除了被声明为常量外很少使用。常量是指声明为public/private，final和static类型的变量。**常量初始化后不可改变。** 3. 具有默认值。**数值型变量默认值是0**，布尔型默认值是false，引用类型默认值是null。 4. 类变量被声明为public static final类型时，**类变量名称必须使用大写字母**。如果静态变量不是public和final类型，其命名方式与实例变量以及局部变量的命名方式一致。 5. 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。**一个对象改变了类变量，在其他类中这个类变量的值也改变了。** 6. 储存在静态存储区。 |

##### 可变参数

public class VarargsDemo {

public static void main(String args[]) {

// 调用可变参数的方法

printMax(34, 3, 3, 2, 56.5);

printMax(new double[]{1, 2, 3});

}

public static void printMax( double... numbers) {……}}

##### 数组

声明数组变量：首选dataType[] arrayRefVar, 也可dataType arrayRefVar[]

初始化方法：

1） 静态初始化：

int[] myArray = {1,2,3,4,5}; //这种方法不可以先声明后赋值，其他两种可以

int[] myArray = new int[]{1,2,3}; // []内必须为空，{}内可空可不空（为空时，数组长度为0，且不能扩展，无意义）

2） 动态初始化：

int [] myArray = new int[5]; // []内必须有长度

**范例：**静态初始化方式作为实参

|  |
| --- |
| **public static void** main(String [] args) {  *function*(**new int**[]{1,2,3}); *//编译正常  function*({1,2,3}); *//编译错误* } **public static void** function(**int** [] array){} |

**范例：**数组作为函数返回值

|  |
| --- |
| **public int**[] function(){  **int** [] array = {1,2,3};  **return** array; *//直接用数组名* } |

数组两个操作方法：System.out.arraycopy(),java.util.Array.sort()

数组没有覆写toString(),输出必须用下标循环输出。

1. 多维数组

方法1，可变数组

int[][] a = new int[2][];

a[0] = new int[2];

a[1] = new int[3];

此法错误：~~int[][] a = new int[][2]~~，第一维不能省略

方法2

int[][] a = new int[2][3];

带初始化：

int[][] a = {{11,12,13},{21,22,23}};

**2维数组array[][]:**

**array.length返回行数，**

**array[i]返回第i行的列数。**

1. array 类

练习：

数组反转，数组排序（用已有方法或自己写排序）

对象数组内存图

##### jar命令

-c 创建一个jar包

-v 生成详细打包过程，并输出到标准设备

-f 指定jar包名

打包成jar包

1. 使用javac编译

2. 把含有编译好的class文件的文件夹打包

jar -cvf my.jar dirname

3. 打包完自动建立

##### 递归

所有循环都能变成递归。训练循环变成递归。

##### 代码模型

模型1:

|  |
| --- |
|  |

模型2：表结构，一个部门多个雇员，一个雇员属于一个部门，一个雇员有一个领导 P162

|  |
| --- |
|  |

模型3:

|  |
| --- |
|  |

模型4:

|  |
| --- |
|  |

##### 链表

|  |
| --- |
| **class** Node {  **private** String **data**;  **private** Node **nextNode**;  **public void** setData(String data){  **this**.**data** = data;  }  **public void** setNextNode(Node nextNode){  **this**.**nextNode** = nextNode;  }  **public** String getData(){  **return this**.**data**;  }  **public** Node getNextNode(){  **return this**.**nextNode**;  } } **class** Link {  **private** String **data**;  **private** Node **firstNode**;  **private** Node **flagNode**;  **public void** add(String data){  Node newNode = **new** Node();  newNode.setData(data);  **if**(**firstNode** == **null**){  **this**.**firstNode** = newNode;  }**else**{  **flagNode** = **firstNode**;  **while**(**flagNode**.getNextNode() != **null**){  **flagNode** = **flagNode**.getNextNode();  }  **firstNode**.setNextNode(newNode);  }  } } **public class** HelloWorld {  **public static void** main(String [] args) {  Link link = **new** Link();  link.add(**"A"**);  link.add(**"B"**);  link.add(**"C"**);   } } |

##### 代码块

类中代码块（构造块）

|  |  |
| --- | --- |
| **class** Demo {  **int i** = 10 ;  {  i = 100 ;   } } | **class** Demo {  {  **int** i = 100;  }  **int i** = 10 ; } |
| **正常** | **正常** |

类中方法中代码块

|  |  |
| --- | --- |
| **class** Demo {  **public void** fun(){  {  **int** i = 100;  }  **int** i = 10 ;  } } | **class** Demo {  **public void** fun(){  **int** i = 10 ;  {  **int** i = 100;  }  } } |
| **正常** | **报错，i变量已经定义过了** |

主方法中代码块（普通代码块）

|  |  |
| --- | --- |
| **public class** Client {  **public static void** main(String [] args) {  {  **int** i = 100 ;  }  **int** i = 100 ;  } } | **public class** Client {  **public static void** main(String [] args) {  **int** i = 10 ;  {  **int** i = 100 ;  }  } } |
| **正常** | **报错，i变量已经定义过了** |

非主类静态块

|  |
| --- |
| **class** Demo {  **static** {  **int** i = 10 ;  } } |

主类中静态块

|  |
| --- |
| **public class** Client {  **public static void** main(String [] args) {  }  **static** {  } } |

代码块执行顺序：主类静态代码块-非主类静态代码块-非主类普通代码块-构造函数-非主类方法中代码块

|  |
| --- |
| **class** Demo {  **public void** fun(){  {  System.***out***.println(**"非主类方法中代码块"**) ;  }  }  **public** Demo(){  System.***out***.println(**"构造函数"**) ;  }  {  System.***out***.println(**"非主类普通代码块"**) ;  }   **static** {  System.***out***.println(**"非主类静态代码块"**) ;  } }  **public class** Client {  **public static void** main(String [] args) {  **new** Demo().fun() ;  }  **static** {  System.***out***.println(**"主类静态代码块"**) ;  } } |
| **输出：**  主类静态代码块  非主类静态代码块  非主类普通代码块  构造函数  非主类方法中代码块 |

##### java执行顺序

当程序执行时，需要生成某个类的对象，Java执行引擎会先检查是否加载了这个类，如果没有加载，则先执行类的加载再生成对象，如果已经加载，则直接生成对象。

在类的加载过程中，类的static成员变量会被初始化，另外，如果类中有static语句块，则会执行static语句块。static成员变量和static语句块的执行顺序同代码中的顺序一致。记住，在Java中，类是按需加载，只有当需要用到这个类的时候，才会加载这个类，并且只会加载一次。

在生成对象的过程中，会先初始化对象的成员变量，然后再执行构造器。也就是说类中的变量会在任何方法（包括构造器）调用之前得到初始化，即使变量散部于方法定义之间。

范例：说明new 类时，先初始化类成员变量，再调用构造。

|  |
| --- |
| **class** A {  **public** A(){  System.***out***.println(**"1"**);  }   B **b** = **new** B(); }  **class** B {  **public** B(){  System.***out***.println(**"2"**);  } }  **public class** Client {  **public static void** main(String [] args) {  A a = **new** A();  } } |

接下来需要看的顺序：

数组

array

泛型

枚举

spring http