1. 认识对象：类）
2. 程序：为实现特定目标或解决特定问题而用计算机语言编写的命令序列的集合；由序列组成，告诉计算机如何完成一个具体的任务。分为系统程序和应用程序两大类。
3. 程序应该包括：对数据的描述（数据的类型和组织形式）和对操作的描述（操作步骤，算法），Java语言编写的程序使用的是编译与解释相结合的方式来运行的。
4. 语言：高级语言：C, C++, Java, C#, Basic, Pascal

脚本语言：JavaScript, Python, Ruby, PHP, Asp.Net, Perl

1. 面向对象设计思想：万物皆对象，对象是细粒度的，通过消息（方法）相互作用；程序由类组成。
2. 面向对象的思维：明确问题域（分析这个问题里有哪些类和对象）、分析这些类和对象应该具有哪些属性和方法、分析类和类之间具体有什么关系。
3. 面向对象编程三个特性：封装，继承，多态

面向对象编程的核心思想就是将数据和数据的操作封装在一起；多态，一种是方法的重载，另一种是指同一个操作被不同对象调用时可能产生不同的行为。

1. public修饰的类的名称必须与Java文件同名
2. 认识对象：类）
3. 对象
4. 对象是我们认识世界的基本单元，万物皆对象
5. 对象的特征：对象的标示（名称）唯一；对象的状态（属性、数据区）；对象的行为（方法、功能）

属性：（对象的特征）对象具有的各种特征，在类中表示为变量，每个对象的每个属性都有特定值，属性名称由类的所有实例共享；

方法：（对象的动作）对象执行的操作，指定操作对象数据的方式。

1. 对象提供的服务是由对象的方法实现的，发送消息实际上也就是调用对象的方法。
2. 类是一种抽象的复合数据类型。类是总称，对象是个体，也叫实例。类和对象是抽象和具体的关系，类是创建对象的模板，对象是类的具体实例。
3. 对象提供的服务是由对象的方法来实现的，发送消息就是调用一个对象的方法
4. 创建一个类就是创建一个新的数据类型，实例化一个类，就得到一个对象。
5. 类的真正意义就是在描述事物，类的构成有两部分：成员变量和成员方法。

成员变量可以是基本类型或数组，也可以是类的对象。 （属性的体现）

成员方法用于处理该类的数据，是用户与对象间或对象之间的交互接口。（功能的体现）

1. 面向对象的核心：

1个工具：抽象（摒弃细节，提取共性）；

2个概念：对象（客观存在的实体），类（具有相同性质的对象的抽象体）；

3个特性：封装（encapsulation），继承（inheritance），多态（polymorphism）；

4个步骤：分析，设计，实现，测试

1. 面向对象设计思想：万物皆对象；程序由类组成；数据与操作行为放在一起作为对象；对象之间通过消息（方法）相互作用；重点在于类的设计，而不是对象的设计。
2. new关键字可创建具体实例，对象是通过使用new关键字调用类的构造方法创建出来的。
3. 类之间的关系：关联（实例之间有对应关系），依赖（调用，类A访问类B提供的服务），聚集（整体与部分，类A的对象由类B的对象组合而成），泛化（A继承B），实现（A实现了B接口）
4. 关键字：

量：true, false, null, this, super

运算：new, instanceof(双目运算符，用于判断一个对象是否是特定类或其子类的实例)

类型：boolean, byte, char, short, int, long, float, double, void, class, interface(接口), enum(枚举)

控制：if, else, switch, case, default, for, do, while, continue, break, return

修饰：private, protected, public, abstract(抽象类), static, final, synchronized(保证最多只有1个线程执行), strictfp(精确浮点), native(说明方法是原生函数，即由C/C++实现的), transient(使属性不被序列化), volatile(说明变量值随时随着线程而修改)

声明：package, import, extend, implements

异常：try, catch, finally, throws, throw

调试：assert(创建断言assertion)

保留：const(不允许改变), goto(无条件转移)

1. 标识符：用于识别不同实体的由若干符号组成的一个名字，用于类、方法、变量的名字。
2. 格式

开头的符号：字母、下划线、$

其余的符号：字母、下划线、$、数字

1. 标识符

项目名：要开发的系统名字

文件名：系统中包含的文件，一般由public修饰的类名确定

类名：每个单词的首字母大写，其余小写

对象名：又称实例，属于变量

变量名：用于存储数据，分为成员变量和局部变量，成员变量第一个单词全部小写，后面的与类名风格一致，局部变量一般小写

方法名：参数属于局部变量，没有局部方法

1. 十进制：1-9开头；八进制：0开头；十六进制：0x开头
2. 数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 基本数据类型 | 数值类型 | 整数：byte, short, int, long |
| 浮点数：float, double |
| 字符类型 | char |
| 布尔类型 | boolean |
| 复合数据类型 | 类 类型 |  |
| 数组 |  |
| 接口类型 | interface |

1. 数据各式各样，要先根据数据的需求（即类型）为它申请一块合适的空间

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型名称 | | 关键字 | 占用位数 | 取值范围 |
| 整数 | 字节型 | byte | 8 | -27 ~ 27-1 |
| 短整型 | short | 16 | -215 ~ 215-1 |
| 整型 | int | 32 | -231 ~ 231-1 |
| 长整型 | long | 64 | -263 ~ 263-1 |
| 小数 | 单精度浮点型 | float | 32 | 1.4e-45f ~ 3.4028235e38f |
| 双精度浮点型 | double | 64 | 4.9e-324d ~ 1.7976931348623157e308d |
| 字符型 | | char | 16 | 0-65535或u0000-\UFFFF |
| 布尔型 | | boolean | 8 | true和false |

1. Java程序的基本结构）
2. 基本Java程序三大件：

包的声明：指定类所在的位置；类的导入：用到其它位置中的类；类的定义：核心部分

1. 一个Java源文件中允许编写多个类，但最多有一个类用public修饰。
2. 关键字class表示类定义的开始，类修饰符分为访问控制符和类型说明符。
3. 类的访问控制符有两个，一个是public，即公共类，另一个是默认，即没有访问控制符。

① 公共类，能够被其它所有的类访问和引用。

② 一个Java源程序中只能有一个public类，且一般含有main方法。

③ 不用public定义的类，只能被同一个包中定义的类访问和引用。

④ 在一个Java程序中可以定义多个这样的类。

1. 类的类型说明符主要有两个 final，abstract

成员变量的修饰符主要有四种，public、private、protected、默认

成员方法的修饰符为public、private、protected、static、final等

1. 成员变量的定义格式： [修饰符] 变量的数据类型 变量名[=初始值]

成员方法的定义格式：

[修饰符] 返回值类型 方法名 ([形参说明]) [throws 例外名1, 例外名2]

{

局部变量声明;

执行语句组;

}

1. 方法一般有一个返回值表示执行结果，无返回值用void表示，返回值类型可以是所有类型，有返回值的用return语句将值返回给调用者。
2. 创建对象：
3. 必须使用new创建一个对象；
4. 使用属性和方法：对象名.成员变量 对象名.方法名
5. 同一个类的每个对象有不同的成员变量的存储空间；
6. 同一个类的每个对象共享该类的方法。
7. Java中所有对象都有默认的toString()方法，若创建类时没有定义toString()则输出对象时会输出对象的哈希码值（对象的内存地址）
8. 声明并不为对象分配内存空间，只分配引用空间；对象的引用类似指针，是32位地址空间，而对象所在实际内存地址是不可操作的。
9. 在同一个类中，方法可以直接访问成员变量

在不同的类中，方法要先创建要访问类的对象，再用对象访问类中定义的成员。

1. 在类的方法定义中使用this关键字代表使用该方法的对象的引用，this可看做一个变量，其值是当前对象的引用；this有时可用于处理方法中成员变量与参数重名的问题。
2. 方法的重载：指一个类中可以定义 名字相同而参数不同 的多个方法（包括参数个数和类型），调用时会根据不同的参数表选择对应的方法。
3. 构造方法(constructor)：一类特殊的成员方法，用new来对新创建的对象进行初始化，与类同名；没有返回值；每个类至少有一个构造方法。
4. 构造方法有不带参数（默认）和带参数两种，构造方法可以重载且通常都重载。

如果类没有显式定义构造方法，系统将自动提供默认的构造方法；如果程序员定义了构造函数，系统就不会创建默认的无参构造函数。

默认构造方法没有参数，没有方法体，如：Animal(){}，所以不写构造方法就能用new Xxx()创建类的实例。

带参数的构造方法，可以显式地为实例变量赋予初始值。

1. 格式：类名 对象名 = new 构造函数（实际参数）

new关键字的作用：为对象分配内存空间；引起对象构造方法的调用；为对象返回一个引用。

1. 默认构造方法创建对象时，成员变量会被赋予数据类型的隐含初值。

|  |  |  |  |
| --- | --- | --- | --- |
| 变量类型 | 默认值 | 变量类型 | 默认值 |
| byte | 0 | short | 0 |
| int | 0 | long | 0L |
| float | 0.0f | double | 0.0d |
| char | ‘\u0000’ | boolean | false |
| 引用类型 | null |  |  |

1. 如果一个类没有明显的父类，Object类就是它的父类。
2. 自动初始化只用于成员变量，方法的局部变量不能被自动初始化，必须赋值才能使用。
3. 局部变量：定义在方法中或{}语句中，存储在栈内存的方法中，随方法运行而出现，随方法弹栈而消失，没有默认初始化值。

成员变量：定义在类中，存储在堆内存的对象中，随对象出现或消失，有默认初始化值

1. Java虚拟机后台线程负责内存的回收。

垃圾强制回收机制：Java系统提供了方法“System.gc()”和“Runtime.gc()”强制立即回收垃圾（但系统并不保证会立即进行垃圾回收）。Java虚拟机在回收对象存储单元前先调用该对象的finalize方法，如果该对象没有定义finalize()方法，则先调用该对象默认的finalize方法。

1. 判断一个存储单元是否是垃圾的依据：该存储单元所对应的对象是否仍被程序所用。

判断一个对象是否仍为程序所用的依据：是否有引用指向该对象。

1. 用new语句创建类的对象时，Java虚拟机会从最上层的父类开始，依次执行各个父类以及当前类的构造方法，若子类构造方法没有通过super语句调用父类的构造方法，那么Java虚拟机会自动先调用父类的默认构造方法。
2. 对象是程序处理数据的最主要载体，数据以实例变量的形式存放在对象中，每个对象生命周期的开始阶段都被分配内存并对其实例变量初始化。

对象构造顺序：new对象时，从最上层的父类开始，依次执行构造方法。

1. 匿名对象：没有名字的对象，如：new Student(); 匿名对象用完就是垃圾，且其没有被栈内存中的变量指向，所以会被回收。匿名对象常作为实参传给一个函数调用。

new Student().show(); 匿名对象方法调用。

1. 对象由两部分组成：对象的实体和对象的引用
2. 静态属性和静态方法）
3. 程序运行时的内存占用：代码区：存放程序的代码；数据区：存放全局数据和静态数据；堆区：存放动态申请的数据；栈区：存放局部数据和参数
4. 用static修饰的属性（变量）称为静态属性，又叫类变量；用static修饰的方法称为静态方法，又叫类方法（静态方法里，无this）；用static修饰的初始化语句块，称为静态初始化语句块。
5. static修饰的变量或方法独立于该类的任何对象，他们被类的实例对象共享，类被加载时就会加载static修饰的部分，之后即便没有创建对象也可以访问static修饰的变量或方法。静态变量在内存中只有一份，只分配一次内存空间；静态方法内部不能用this

静态可以访问静态，但不能访问非静态，非静态可以访问静态。

static方法要想调用non-static的域或方法，必须通过对象名引用

1. 即使没有创建该类的具体对象，类中的static类成员也存在，此时只能通过类名调用。

静态变量和方法：由static修饰，由该类产生的所有实例共享；类变量保存在全局数据区；调用方式：类名.静态变量，类名.静态方法 或 对象名.实例变量，对象名.实例方法

实例变量和方法：每个实例拥有一份独立的拷贝，每个实例对象的数据是独立且唯一的；实例变量保存在堆(heap)中；调用方式：对象名.实例变量，对象名.实例方法

1. 静态代码块
2. 静态代码块在非静态代码块之前执行；静态代码块只在第一次new执行，之后不再执行，而非静态代码块每new一次就执行一次。

（首次new）静态代码块—>非静态代码块—>默认构造方法—>普通方法中的代码块

（之后）非静态代码块—>默认构造方法—>普通方法中的代码块

1. 一个类中允许定义多个静态代码块，执行的顺序根据定义的顺序进行
2. 静态代码块只能定义在类里面，独立于任何方法，不能定义在方法里面。
3. 静态代码块里面的变量都是局部变量，只在本块内有效。
4. 静态代码块在类被加载时自动执行
5. 静态代码块只能访问类的静态成员，而不允许访问实例成员。
6. 封装）
7. 封装：隐藏属性、方法或实现细节的过程，仅对外公开接口，可以降低各个系统之间的耦合，是面向对象方法的核心思想之一，有两个含义：
8. 把对象的属性和行为看成一个密不可分的整体，封装在一个独立单位（对象）中；
9. 尽可能隐藏对象的内部细节
10. 属性和方法的访问控制修饰符

|  |  |  |
| --- | --- | --- |
| private | 私有级别 | 同类 |
| default | 默认级别 | 同类，同包 |
| protected | 受保护级别 | 同类，同包，子类 |
| public | 公开级别 | 同类，同包，子类，整体 |

1. +表示public -表示private #表示protected
2. 类的访问级别

public：可以被所有类访问 默认(default)：同包访问

1. 属性通常用private，提供public的getter()和setter()方法，使得外界可以访问属性。
2. 构造方法可以用private修饰
3. 单例模式： 一个类在内存中只有一个实例（对象）存在，该类一般没有属性，无法继承，所以无法扩展，无法更改它的实现。其类中有私有的构造函数和公共的静态方法
4. 两种模式：

饿汉式：全局的单例实例在类装载时构建

懒汉式：全局的单例实例在第一次被使用时构建（即单例的延迟加载方式）

1. 适用场合：

①只需要使用一个单独的资源并需要共享其状态信息时；②需要频繁创建和销毁的对象；③创建对象耗时或耗资源过多，但又常用的对象；④工具类对象；⑤频繁访问数据库或文件的对象。

1. 单例类：仅有一个实例的类。系统中具有惟一性的组件可作为单例类，单例类的实例会占用较多内存，或者实例初始化过程较长，因此随意创建单例类的实例会影响系统的性能。
2. 继承）
3. 继承：一种“is-a”的关系；是在保留原有类的数据成员和成员方法的基础上派生出新类，新类可以有变异；新类自动具有原有类的所有数据成员和成员方法。

单继承：子类只能有一个直接父类。

多继承：子类可以有多个直接父类。

Java不支持类的多继承，但支持接口的多继承和一个类实现多个接口。继承用extends，实现用implements。

1. 基类==超类==父类==superclass：通用的方法放在基类中

派生类==子类==subclass：通过增加数据成员或成员方法将基类变为某种更有用的类型

基类是对若干个派生类的抽象；派生类是基类的具体化。

1. 子类拥有父类的所有属性和方法，但父类的私有属性和方法，子类无法直接访问到。子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。子类可以用自己的方式实现父类的方法（方法覆盖）。子类不能继承父类的构造方法。
2. 子类不能继承父类的构造函数；
3. 无参子类构造方法：子类可以通过super()显示调用父类无参的构造函数，也可以隐式调用
4. 有参子类构造方法：初始化父类的成员变量；初始化子类的成员变量；必须显示调用父类有参构造函数

用this调用本类构造函数，或用super调用父类构造函数，都必须是该方法体中的第一条可以执行语句，否则会产生语法错误。

1. 创建子类对象时，子类按层次结构从上到下调用所有父类的构造函数。
2. 如果父类没有不带参数的构造方法，则在子类的构造方法中必须明确说明调用父类的某个带参数的构造方法，通过super，且这条语句还必须出现在构造方法的第一句。
3. 子类创建对象时，子类的构造方法先调用父类的某个构造方法，完成父类部分的创建；然后再调用子类自己的构造方法，完成子类部分的创建。
4. 如果子类的构造方法没有明显指明使用父类的哪个构造方法，就调用父类的不带参数的构造方法。
5. 子类只能操作继承的那部分成员变量。子类可以通过继承的方法来操作子类未继承的变量和方法。
6. 如果子类调用父类的构造方法，则通过super()调用来实现。

如果子类调用父类的同名方法，则通过super.方法名()来实现。

this变量代表对象本身，super代表父类；用this调用本类构造函数和用super调用父类构造函数，都必须是该方法体中的第一条可以执行语句。

this用于：①引用自身对象的成员变量 this.age; ②引用自身对象的成员方法 this.diaplay(); ③调用自身的构造方法 this (“Jack”, Male, 10);

super用于：①引用父类中的隐藏成员变量 super.age; ②访问父类中被覆盖的方法 super.diaplay(); ③访问父类的构造方法 super (“Jack”, Male, 10);

1. 向上转型：将子类转换成父类，只能调用父类中定义的属性和方法，对于子类中的新属性与方法，必须强制转换成子类类型才能调用
2. 变量隐藏：继承中，如果子类的成员变量与父类的成员变量同名，称子类隐藏（override）了父类的成员变量。Fu fu = new Zi(); 此时fu.num是父类中的num值。子类若要引用父类的同名变量，可用super做前缀加圆点操作符，即super.num。
3. 方法覆盖：子类重写父类的方法，它们的名称、参数及返回类型完全相同。Fu fu = new Zi(); 此时fu.show()调用的是子类中的show()方法。
4. 继承提高了程序的复用性；通过向上映射支持多态，提高了程序的可扩展性；增强了一致性。继承破坏了封装性（所以父类中能写成final的尽量写成final）。继承是强耦合关系，父类变，子类必须变。
5. 多态\_1）
6. Java中的引用变量有两个类型：编译时的类型与运行时的类型。编译时的类型由声明该变量时使用的类型决定，运行时的类型由实际赋给该变量的对象决定。如果编译时的类型与运行时的类型不一致就会出现所谓的多态。

class B extends A A a=new B(); 编译时a的类型是A，运行时其类型为B，这就是编译和运行时类型不一致。

1. 多态，子类重写（覆盖）父类方法，运行时使用父类的类型，子类的对象；即用父类的变量去引用不同的子类，调用相同的方法会有不同的结果。实现多态的三个要素：继承；方法重写；用父类类型指向子类对象Fu a = new Zi()。
2. 多态性：指在程序中用相同的名称来表示不同的含义，分为静多态与动多态。
3. 静态多态：编译时多态，也称为静绑定、静态联编；在编译时决定调用哪个方法。包括变量的隐藏、方法的重载，一般指方法重载；只要构成了方法重载，就形成了静态多态的条件；静态多态与是否发生继承没有必然联系。
4. 动态多态：运行时多态，也称动绑定、动态联编；在编译时不能确定所要调用的方法，在运行时才能确定调用哪个方法。覆盖方法、抽象方法和接口采用动多态实现。继承是实现动态多态的首要前提。Java形成动态多态必备条件：①必须有继承；②继承中必须有方法覆盖；③必须由父类的引用指向子类的实例，并且通过父类的引用调用被覆盖的方法；
5. 方法重载 (overloading)：在同一个类中定义多个同名的方法，但这些方法的参数列表不同，即参数的个数或类型或顺序不同。构造方法也可以重载。返回类型不能用来区分重载的方法。
6. 方法覆盖：也称重写。子类对父类的方法重写时，重写的方法与原方法的名称、参数、返回类型完全相同，其访问控制满足：

①若父类方法定义时有异常抛出，子类的重写方法不能有更多的异常抛出；

②子类方法不能缩小父类方法的访问权限：a) default方法可以被重写为 default、protected和public的； b) protected方法可以被重写为 protected和public的； c) public方法只能重写为 public的；

③静态方法不能被重写为非静态方法。

1. Fu a = new Zi(); a.book的值是父类中的book值，a.test()是子类覆盖父类的方法，若子类没有重写此方法，则调用父类方法，若父类没有此方法而子类有，则编译错误。即调用父类的属性、子类的覆盖方法。
2. 覆盖与重载的区别：覆盖（重写）是指子类重写父类的方法，函数名称和参数相同，实现体不同；重载是指同一个类中有相同的函数名称，但参数不同。
3. 继承是子类使用父类的方法，而多态是父类使用子类的方法。
4. Java多态机制是基于“方法绑定(binding)”。方法绑定分为
5. 前期绑定：早绑定，绑定动作发生于程序执行前
6. 后期绑定：晚绑定，动态绑定，执行期绑定，绑定动作在执行期根据对象类型而进行

面向过程的语言都是前期绑定。

Java的方法中，除了final，static，private和构造方法是前期绑定外，其他的方法全是后期绑定。所有private方法都被隐式的指定为final的。

1. 抽象类
2. 抽象类：不能实例化，但可以创建引用。当类实现了一个接口但没有实现接口的所有方法时；或当类继承于抽象父类但没有具体实现抽象父类中的抽象方法，该类必须为抽象类。
3. 抽象方法：没有方法体public abstract void print(); ，用abstract修饰，必须在抽象类中。抽象方法不能被private、final、static修饰，必须被子类覆盖。
4. 一个类里有抽象方法，则该类必须为抽象类；一个抽象类中可以没有抽象方法。
5. 抽象类与具体类

抽象类：用于划分具体类，不能实例化，定义了未提供实现的抽象方法，为自己的部分方法提供实现；

具体类：用于表示真实对象，可以实例化，不定义抽象方法，为所有的方法提供实现。

1. 多态\_2）
2. 接口：

①类对外提供的所有服务：类的所有能被其他程序访问的方法构成类的接口；

②interface定义的接口类型：用于明确描述系统对外提供的服务，能够更加清晰地把系统的实现细节与接口分离。

1. 接口：关键字interface，用来实现不相关的类之间多重继承功能，是一种特殊的抽象类，只包含常量和方法的定义。
2. 接口中所有方法都是抽象的，只有声明（方法头和参数列表），没有方法体。
3. 接口中变量默认public static final，方法默认public abstract；
4. 接口中没有构造方法；
5. 一个类可以实现多个接口；
6. 接口是一种更高级的封装，通过接口可以了解对象的交互界面、实现不相关类的相同行为；
7. 接口可以继承，而且可以多重继承。
8. 接口的声明与实现

声明接口：

[public][interface] 接口名称 [extends 父接口名列表]

{

[public][static][final] 数据类型 变量名 = 常量名; //静态常量

[public][abstract][native]返回值类型 方法名(参数列表); //抽象方法

}

实现接口

[修饰符] class 类名 [extends 父类名] [implements 接口A, 接口B, …]

{

类的成员变量和成员方法;

为接口A中的所有方法编写方法体，实现接口A;

为接口B中的所有方法编写方法体，实现接口B; …

}

若没有实现接口中的所有方法，其实现类仍需定义为抽象类。一个类实现某接口的抽象方法时，必须用完全相同的方法头，并用public修饰符。

1. 接口回调：把实现某一接口的类创建的对象引用赋给该接口声明的接口变量，该接口变量就可以调用被类实现的接口中的方法。
2. 接口变量 = 实现该接口的类所创建的对象; 接口变量.接口方法([参数列表]);
3. 接口做参数：如果一个方法的参数是接口类型，就可以将任何实现该接口的类的实例的引用传递给接口参数，那么接口参数就可以回调类实现的接口方法。
4. 抽象类和接口：

共同点：都不能实例化，都可包含抽象方法及具体实现的方法，都需要其他类具体实现，都位于系统的抽象层，需要其他类来进一步提供实现细节。

区别：

1. 接口中的成员变量和方法只能是public类型的；抽象类中的成员变量和方法可以是各种访问级别，抽象类比接口包含了更多的实现细节。
2. 接口中的成员变量只能是public、static和final类型；抽象类中可以定义各种类型的实例变量和静态变量。
3. 一个类只能继承一个直接的父类，这个父类可能是抽象类；一个类可以实现多个接口。
4. 在语义上，接口表示更高层次的抽象，声明系统对外提供的服务；而抽象类则是各种具体类型的抽象。
5. 接口是一种能力，规定了类“做什么”，不关心“怎样做”。
6. native关键字：Java要实现对底层的控制，就需要一些其他语言的帮助；native用于声明一个方法是由机器相关语言（如C/C++）实现的而非java类库。
7. 复合与继承）
8. 继承：子对象继承父对象的数据和操作，通过继承可以支持多态；“is”的关系。
9. 复合：也称包容、包含。指把一个对象作为另一个对象的一部分，从而达到复用的目的；“has”的关系。
10. 设置访问权限：
11. 一般属性设置为private，方法设置为public；
12. 特殊的属性和方法用protect与default；
13. 总会在一个类中提供一个或多个统一的访问接口，如public方法
14. 完善类的设计\_1）
15. 多态(overview)的实现条件：继承、重写、向上转型。即有父子类关系或类实现接口关系，父类引用变量调用方法时调用子类重写后的方法。 Fu fu = new Zi();

成员变量：多态调用子父类中同名的成员变量时，编译和运行都看等号左边。即等号左边的类中有该变量才能编译通过，否则报错；运行时用的也是左边（父）类中的变量值。

成员方法：多态调用子父类中同名的成员方法时，编译看左边，运行看右边。即等号左边的类中有该方法才能编译通过，否则报错，运行时调用右边（子）类中重写的成员方法，若没重写，则调用父类的对应成员方法。

1. 父子类对象之间的转换：自动转换（向上映射/转型）、强制类型转换（向下映射/转型）
2. 向上转型：子类对象赋值给父类引用，Fu a = new Zi(); 中a就是上转型对象；多态本身就是向上转型的过程，当不想使用子类特有功能时，就需要使用向上转型；上转型对象可以访问子类继承或隐藏的成员变量、调用子类继承或重写的方法。其成员变量是父类的，方法是子类重写的；上转型对象不能操作和调用子类新增的成员变量和方法。
3. 向下转型：已经向上转型的子类对象使用强制类型转换，将父类引用转为子类引用，此时其成员变量和方法都是子类的。Fu a = new Zi(); Zi zi = (Zi) a; 当要使用子类特有功能时，就需要使用向下转型；若正常直接创建父类对象，无法向下转型，会产生运行时异常（ClassCastException转换异常），所以转换前必须做类型判断 if( !a instanceof Dog){…}
4. instanceof操作符：用于判断一个引用类型所引用的对象是否是某类的一个实例。是Java独有的双目运算符。左边是引用类型，右边是类名或接口名。

例：Fish fish = new Fish(); “fish instanceof XXX”表达式为true要求XXX是Fish类或Fish类的直接间接父类或Fish类实现的接口。

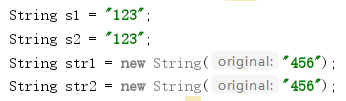
1. 所有的Java类都直接或间接继承java.lang.Object类，Object类是所有Java类的祖先，其中定义了所有的Java对象都具有的方法。

- equals(Object obj)：比较两个对象是否相等。当两个引用变量指向同一对象时，返回true

- toString()：返回当前对象的字符串表示。

equels() 与 == ：Object类下的equals()方法与”==”没有区别，都是判断两个对象是否为同一引用，只有当比较的两个对象是同一个对象时返回true。

有些类重写了equels()方法，此时二者的结果会不同，此时的equels()方法，包括java.io.File、java.util.Date、java.lang.String、包装类（如java.lang.Integer和java.lang.Double等）。

s1==s2: 字面量直接进入常量池，所以指向的是同一个内存地址；

s1==”1”+”23”：拼接的部分都是字面量，编译时会被优化，所以仍然是同一个地址；

“字符串” == “字符”+”串” 是错误的：引号包含文本，创建的String对象用“+”连接，产生的新对象会被加入字符串池而不是常量池中，所以不是同一个地址；

str1==str2 是错误的：因为两个都是new出来的新对象，而String库中的equels()被重写了，所以不再指向同一个地址；

“Hello”==”Hel”+new String(“lo”) 是错误的：所有包含new方式新建对象（包括null）表达式产生的新对象都不会加入字符串池中。

1. 内部类
2. 成员内部类
3. 实例内部类

Outer.InnerTool tool=new Outer().new InnerTool(); 等价于

Outer outer=new Outer(); Outer.InnerTool tool =outer.new InnerTool() ;

创建实例内部类的实例时，外部类的实例必须已存在。（如上代码）

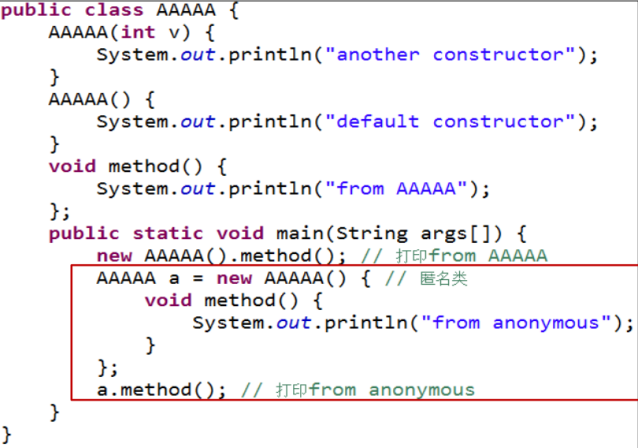
在内部类中，可以直接访问外部类的所有成员变量和成员方法。

实例内部类的实例自动持有外部类的实例的引用，

1. 静态内部类

静态内部类的实例不会自动持有外部类的特定实例的引用，在创建内部类的实例时，不必创建外部类的实例。Outer.Inner inner = new Outer.Inner(); 其中Inner为静态内部类

其他类可通过完整的类名直接访问静态内部类的静态成员。

1. 局部内部类：在一个方法内部定义的类，只能在当前方法中使用，可以访问外部类的所有成员变量和方法，以及所在方法中用final修饰的变量或参数（最终变量或参数）
2. 匿名类：没有名字的类，将类和类的方法定义在一个表达式范围里，本身没有构造方法，但是会调用外部类的构造方法。

匿名内部类将内部类的定义与生成实例的语句合在一起，并省去了类名和关键字class、extends、implements等

右图输出：

default constructor

from AAAAA

default constructor

from anonymous

1. 内部类的用途
2. 封装类型：内部类是一种封装类型的有效手段，正常类只能有public和default访问级别，成员内部类可以有public，protected，default，private四个访问级别。
3. 直接访问外部类的成员：内部类可以访问外部类各种访问级别的变量方法。
4. 回调外部类的方法
5. 回调：指一个类实际上实现了某种功能，但没有直接提供相应的接口，客户类可以通过这个类的内部类的接口来获得该功能。而这个内部类本身并没有提供真正的实现，只是调用外部类的实现。
6. 内部类编译会生成.class文件，命名如下：

成员内部类：外部类名$内部类名.class

局部内部类：外部类名$数字和内部类名.class

匿名类：外部类名$数字.class

|  |  |  |  |
| --- | --- | --- | --- |
| 比较方面 | 实例内部类 | 静态内部类 | 局部内部类 |
| 主要特征 | 内部类的实例引用特定的外部类的实例 | 内部类的实例不与外部类的任何实例关联 | 可见范围是所在的方法 |
| 可用的修饰符 | 访问控制修饰符  abstract，final | 访问控制修饰符  static，abstract，final | abstract，final |
| 可以访问外部类的哪些成员 | 可以直接访问外部类是所有成员 | 只能直接访问外部类的静态成员 | 可以直接访问外部类的所有成员，以及所在方法的或实际上的最终变量和参数 |
| 拥有成员的类型 | 实例成员 | 静态成员和实例成员 | 实例成员 |
| 外部类如何访问内部类的成员 | 必须通过内部类的实例访问 | 静态成员可以通过内部类的完整类名访问 | 必须通过内部类的实例访问 |

1. 最终类：final修饰，不再派生子类

最终方法：final修饰，不能被覆盖，也不能被重载，方法的定义永远不能更改。

final类中的方法可以不是final方法（实际上都是隐式final方法），final方法可以不在final

类中。abstract和final不能同时修饰一个类。

使用最终方法和最终类可以关闭动态绑定。

1. 常量：程序运行过程中，值不变的量，用final修饰，
2. final可以修饰简单数据类型，也可以修饰（包括用户自定义）复合数据类型；
3. final常量可以在声明时赋初值，也可以在构造函数中赋初值；
4. 常量可以是局部常量或类常量，类常量一般用static修饰。
5. 声明格式：final 数据类型 常量名 = 值
6. 常量名大写，多个单词用下划线连接。
7. 简单数据类型常量的值不能改变；复合数据类型常量指引用不能改变，但具体值可以改变。
8. 完善类的设计\_2）
9. 包：package创建，是类和接口的集合，用来管理名称空间。

包定义语句必须是Java源文件中的第一条可执行语句。

系统默认会为每个.java源文件创建无名包，无名包不能被其他包引用。

包的引用用import。默认Java类智能访问到Java.lang的类和接口。

1. Java常用类库：

java.lang ：包含Java语言的核心类库 java.util ：提供各种实用工具类

java.io ：标准输入输出类 java.net ：实现网络功能

java.awt ：图形界面中常用类和接口 java.applet ：提供对通用Applet支持的类和接口

java.security ：支持Java程序安全性的类和接口

1. 模块：从逻辑上将系统分解为更细微的部分，可以是函数、类、功能块等；
2. 耦合：主要描述模块之间的关系；

低耦合：模块之间直接依赖程度低，模块独立性强。若模块A与B只是通过数据交互则为弱耦合，若模块A直接操作模块B中数据，则为强耦合；

1. 内聚：主要描述模块内部

高内聚：模块内部的元素，关联性越强，内聚越高，模块单一性越强。一个模块尽可能独立完成某个功能。

1. 接口设计原则：单一职责原则（一个类只负责一个功能领域中的相应职责），开闭原则（对扩展开放，对修改关闭），里氏代换原则（所有引用父类处都能透明使用其子类对象），依赖倒转原则（针对接口编程而不是实现编程），接口隔离原则（用多个专门的接口），迪米特法则（软件实体少与其他实体相互作用）
2. 外观模式：为系统中多个子系统提供一致的对外调用, 对客户端隐藏子系统细节, 降低其与子系统的耦合。

桥接模式：JDBC中，把面向厂商的接口和面向使用者的API拆分隔离。

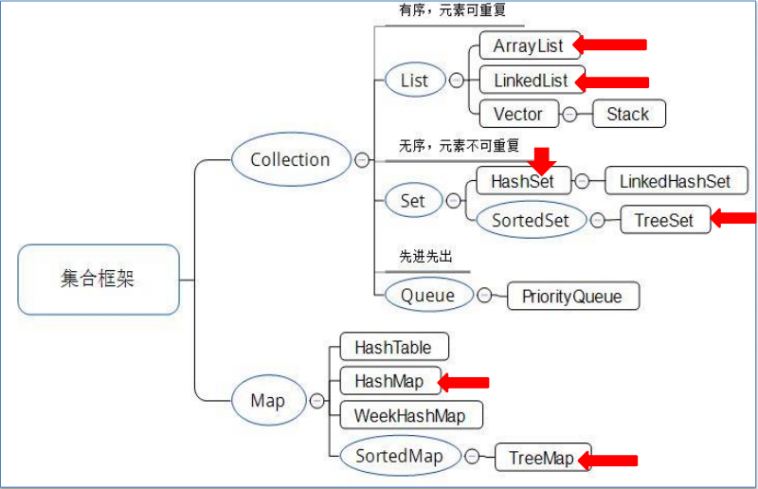
适配器模式：引入第三方库(hibernate, log4j)，不直接在代码中继承或使用实体类，而是抽出上层统一接口，增加实现类，对外暴露接口。

1. 代码复用原则：面向接口的编程，优先使用对象组合而不是类继承。

继承被称为白箱复用，对象组合称为黑箱复用；

继承之间的类的依赖关系远远强于组合之间的依赖关系

1. 集合与映射）
2. Java集合框架：包括接口， 具体类，算法



集合类：在java.util包里，提供了一些基本数据结构的支持。只容纳Object类的实例对象，不容纳基本类型的数据；使用集合成员前必须重新造型。

集合框架：为表示和操作集合而规定的一种统一、标准的体系结构。包含对外的接口、接口的实现、对集合运算的算法。

1. 接口：即表示集合的抽象数据类型。接口提供了让我们对集合中所表示的内容进行单独操作的可能。
2. 实现：也就是集合框架中接口的具体实现。实际它们就是那些可复用的数据结构。
3. 算法：在一个实现了某个集合框架中的接口的对象上，完成某种有用的计算的方法，例如查找、排序等
4. 集合框架中的接口
5. Collection：集合层次中的根接口，List、Set、Queue接口的父接口，存储一组不唯一、无序的对象，接口里的方法可以操作Set集合与List，JDK不提供其直接实现类；
6. Set：存储一组唯一、无序的对象，不能包含重复元素，元素必须定义equels()方法，SortedSet是升序排列的Set，实现Set接口的类有HashSet，TreeSet；
7. List：有序集合，存储一组不唯一但有序（插入顺序）的对象，可以包含重复元素，提供了按索引访问的方式，实现List接口的类有ArrayList和LinkedList。

① ArrayList：顺序存储，实现了长度可变的数组，遍历和随机访问的效率较高；

② LinkedList：链表存储，插入删除元素时效率较高。

1. Map：存储一组键值对象，包含key-value对，不包含重复key，专门处理键值映射数据的存储，可以根据键实现对值的操作。SortedMap是升序排列key的Map；最常用的实现类是HashMap。
2. ArrayList接口常用方法

|  |  |
| --- | --- |
| 方法名 | 说明 |
| boolean add(Object o) | 在列表的末尾顺序添加元素，起始索引从0开始 |
| void add(int index, Object o) | 在指定的索引位置添加元素，索引位置介于0与列表中农元素个数之间 |
| int size() | 返回列表中的元素个数 |
| object get(int index) | 返回指定索引位置的元素，取出的元素是Object类型，使用前需进行强制类型转换 |
| boolean contains(Object o) | 判断列表中是否存在指定元素 |
| boolean remove(Object o) | 从列表中删除元素 |
| object remove(int index) | 从列表中删除指定位置元素，起始索引从0开始 |

1. LinkedList接口特殊方法

|  |  |
| --- | --- |
| 方法名 | 说明 |
| void addFirst(Object o) | 在列表的首部添加元素 |
| void addLast(Object o) | 在列表的末尾添加元素 |
| object getFirst() | 返回列表中的第一个元素 |
| object getLast() | 返回列表中的最后一个元素 |
| object removeFirst() | 删除并返回列表中的第一个元素 |
| object removeLast() | 删除并返回列表中的最后一个元素 |

1. Map接口特殊方法

|  |  |
| --- | --- |
| 方法名 | 说明 |
| Object put(Object key, Object val) | 以“键-值”对的方式进行存储 |
| Object get(Object key) | 根据键返回相关联的值，如果该键不存在，返回null |
| Object remove(Object key) | 删除由指定键映射的键值对 |
| int size() | 返回元素个数 |
| Set keySet() | 返回键的集合 |
| Collection values() | 返回值的集合 |
| Boolean containsKey(Object key) | 如果存在由指定的键映射的键值对，返回true |

1. 集合类对比
2. Vector与ArrayList

同：实现原理相同，功能相同

异：Vector线程安全；ArrayList线程非安全，重速度。

1. Hashtable与HashMap

同：实现原理相同，功能相同

异：Hashtable继承Dictionary类，线程安全，不允许null值；HashMap实现Map接口，线程非安全，允许null值。

1. Iterator
2. Collection接口的iterate()方法返回一个Iterator，然后通过Iterator接口的两个方法可实现遍历：

- boolean hasNext() ：判断是否存在另一个可访问的元素

- Object next() ：返回要访问的下一个元素

1. Iterator接口主要用于遍历Collection集合中的元素，Iterator对象也称迭代器
2. Iterator接口向应用程序提供了遍历Collection集合元素的统一编程接口，隐藏了各种 Collection实现类的底层细节
3. Iterator仅用于遍历集合，其本身并不提供承装对象的能力。如果要创建Iterator对象，必须有一个被迭代的集合。
4. Iterator可以用于所有集合，不能添加对象只能向后遍历。
5. ListIterator：在List、ArrayList、LinkedList、Vector时使用的迭代器，含有方法add(E e), hasNext(), hasPrevious(), next(), nextIndex(), previous(), previousIndex(), remove(), set(E e)。可以添加对象，前后双向遍历，定位当前索引的位置，修改对象。
6. Java集合就像一种容器，可以把多个对象的引用放入容器中。Java集合类可用于存储数量不等的多个对象、保存具有一个映射关系的关联数组。可分为Set，List，Map三种体系

Set：不区分元素的顺序，不允许出现重复元素；

List：区分元素的顺序，允许包含重复元素；

Map：健-值(Key-Value)，映射中保存成对的健-值信息，不能包含重复的健，每个健最多只能映射一个值。

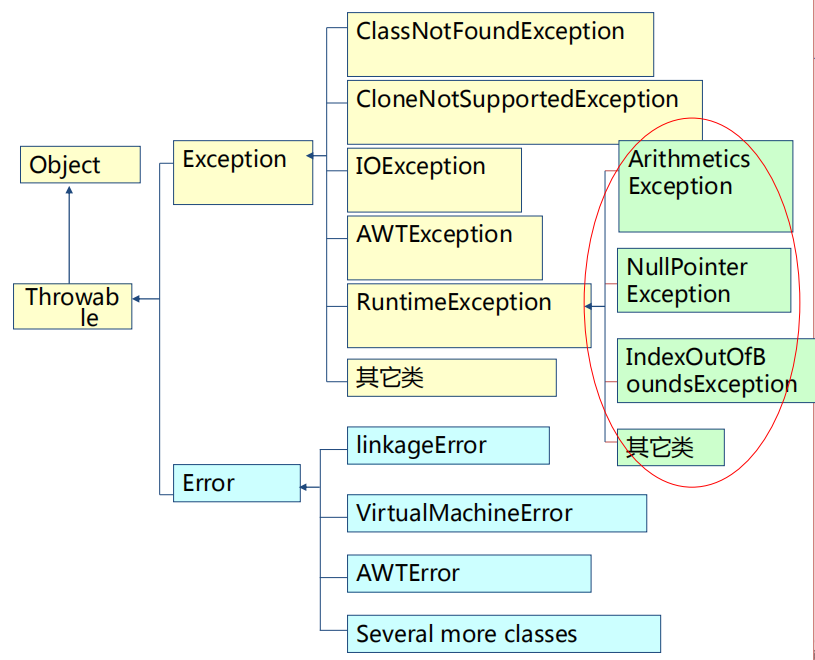
1. 泛型集合：Java5增加，可以记住容器中对象的数据类型。

.

1. 异常处理）
2. 异常：一个可以正确运行的程序在运行中可能发生的错误。编译时的报错和运行结果不正确的逻辑错误都不是异常。具有偶然性、可预见性、严重性。

输入密码错误不当做程序异常来处理

1. 异常类的结构

Throwable：Object的直接子类，所有异常类的父类，java.lang包中；

Exception类：继承自Throwable类；

Error：由Java虚拟机生成并抛出，Java程序不做处理。

Runtime Exception：系统检测，用户程序可不做处理；

非Runtime Exception：Java程序必须捕获或声明所有非运行时异常

throw：用户自己产生异常

常见异常：

ArithmeticException 算数异常

ArrayIndexOutOfBandsException 数组越界

ArrayStoreException 数组类型与元素类型不符

IOException IO流异常，如读写数据时文件不存在

FileNotFoundException 文件未找到

NullPointerException 空指针

MalformedURLException 格式错误的URL异常

NumberFormatException 数字格式异常

OutOfMemoryException 内存不足

1. 异常（按编译时是否受检来分）

非受检异常(unchecked exception)：包括RuntimeException及其子类和Error及其子类，只能在运行时被检测到，编译时不能被检测到。程序无需处理，系统处理即可

受检异常(checked exception)：除了RuntimeException及其子类之外的Exception，编译时即可检测到，必须用声明异常或try-catch处理。

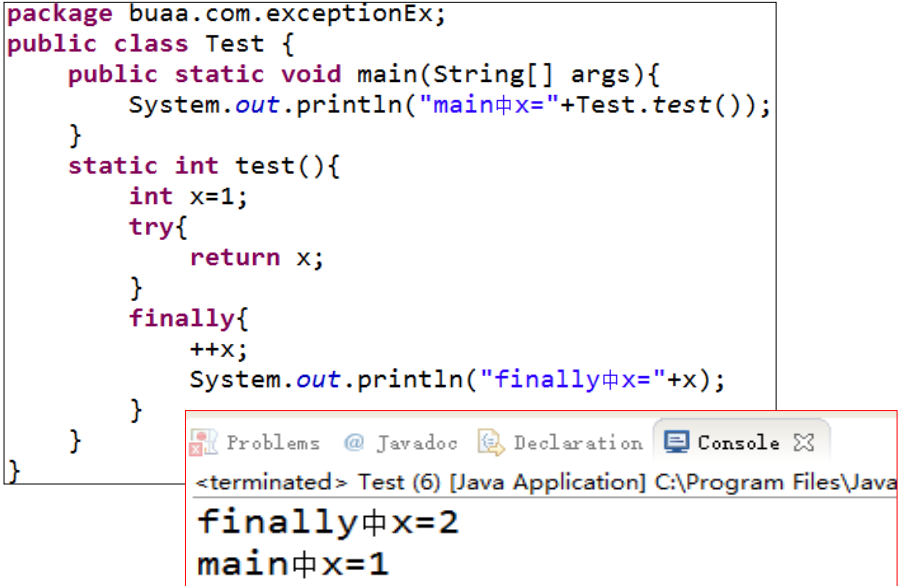
1. 异常处理机制：抓抛模型，若程序执行过程中出现异常，会自动生成一个异常类对象交给运行时系统，称throw；异常会一级一级抛给方法的调用者直到处理，称catch。如果一个异常回到main()方法且main()也不处理，则程序终止运行。
2. 处理可能抛出的异常：
3. 在当前方法中用try-catch捕获并处理异常；
4. 在方法声明处用throws声明抛出异常。

程序中需要处理的异常，一般用try-catch-finally语句捕获并处理；对于程序中不需要处理的异常，可用throws语句在方法中抛出。

1. throws在方法声明处声明抛出特定异常，throw在方法内部抛出具体的异常
2. try-catch-finally语句：
3. try {接受监视的程序块} catch (Exception e) {抛出异常时执行的代码} finally {无条件执行语句}
4. 程序流程从引起异常的代码转到最近的try语句的catch子句。
5. 一个try可以有多个catch语句，程序会执行第一个匹配的catch语句。
6. 若一个异常类及其子类都出现在catch中，子类放在前面。
7. finally语句为无条件执行语句，一般用于释放资源。若没有catch子句，则在执行了try语句块后立即执行
8. 如果try{}里有一个return语句，那么紧跟在这个try后的finally {}里的代码会在return前执行，但finally中的值不影响try中的值。
9. 声明异常：一个方法不处理异常，而是由调用它的方法来处理异常。

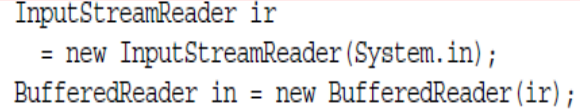
格式：<访问权限修饰符> <返回值类型> <方法名> (参数列表) throws 异常列表{}

1. throws语句：用来表明一个方法可能抛出的各种异常，并说明该方法会抛出但不捕获异常。
2. 重写方法不能抛出比被重写方法范围更大的异常类型。
3. 自定义异常：自己定义的异常，
4. 语句形式：<class> <自定义异常名> extends <Exception>{}
5. throw抛出异常主要用于自定义异常 <throw> <异常对象>，程序会在throw语句处停止，转向try-catch寻找异常处理方法。
6. throws语句在方法声明处声明抛出特定异常；throw语句在方法中抛出具体的异常；try-catch处理异常，条件满足时在catch语句中抛出新的自定义异常。
7. 不要用一个catch块处理所有异常，也不要把大量代码放在一个try语句中进行监测。
8. 若try{}里有return语句，则紧跟在这个try后的finally {}代码会在return前执行。



1. 文件与数据流）
2. IO流：字节流以字节为单位，处理单元为1字节，用InputStream和OutputStream；

字符流以字符为单位，处理单元为2字节，用Reader和Writer。

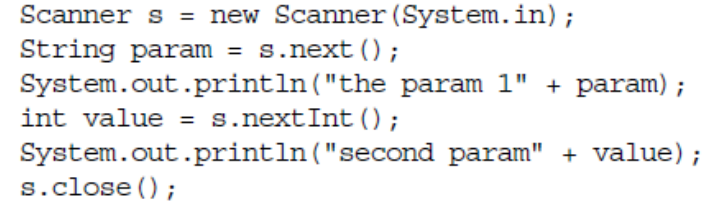
InputStreamReader类是从字节流到字符流的桥接器，读取字节并解码为字符，一般在BufferedReader中包装。

1. 命令行输入java 类名 参数1 参数2 …，此时args[]从0开始依次存储参数。

例：java Test arg1 arg2 “another arg” args[0] is arg1，args[1] is arg2，args[2] is another arg

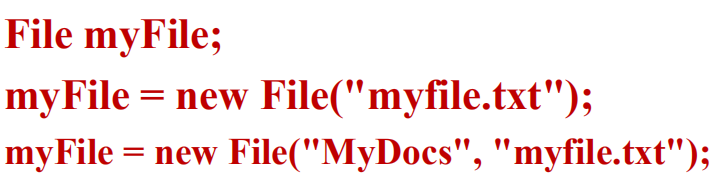
1. 常见格式代码

|  |  |
| --- | --- |
| 代码 | 描述 |
| %s | 将参数格式化为字符串，通常通过对对象调用ToString方法 |
| %d %o %x | 将整数格式化为十进制、八进制或十六进制值 |
| %f %g | 格式化浮点数。%g是使用科学记数法 |
| %n | 将换行符格式化为字符串或流 |
| %% | 将%字符格式化为字符串或流 |

1. 扫描器Scanner
2. 文件File类：代表与平台无关的文件和目录，能新建、删除、重命名文件和目录，以及对路径进行操作，但不能访问文件内容，如需访问，需要用I/O流。
3. File类中的方法

|  |  |
| --- | --- |
| 方法名 | 用途 |
| boolean exists() | 文件是否存在 |
| boolean isFile() | 是否是文件 |
| boolean isDirectory() | 是否是目录 |
| boolean canRead() | 文件是否可读 |
| boolean canWrite() | 文件是否可写 |
| boolean isAbsolute() | 该抽象路径名是否是绝对的 |
| boolean mkdir() | 创建目录 |
| boolean mkdirs() | 创建目录并包括所有必需但不存在的父目录 |
| File createNewFile() | 如果文件不存在则创建 |
| boolean renameTo (File newName) | 重命名文件 |
| boolean delete() | 删除文件或目录 |
| String getName() | 获取文件名 |
| String getPath() | 获得抽象路径名（字符串形式） |
| String getParent() | 获得父目录名，不存在时返回null |
| File getParentFile() | 获得父目录对象，不存在时返回null |
| String getAbsolutePath() | 获得绝对路径（字符串形式） |
| File getAbsoluteFile() | 获得绝对路径（File对象） |
| long lastModify() | 返回文件的最后修改时间 |
| long length() | 返回文件长度 |
| String[] list() | 返回目录下的文件和子目录，包含隐藏文件，对文件操作会返回null |
| String[] list(FilenameFilter filter) | 有文件过滤器的list |
| listFiles() | 有文件过滤器的list |
| boolean setLastModified(long time) | 设置最后修改时间 |
| boolean setReadOnly() | 设置只读属性 |

1. 在对文件进行I/O操作之前，必须先获得这个文件的基本信息
2. 类File一般不涉及文件内部具体内容，只是从整体上对文件或路径进行操作
3. File类的四个构造方法
4. – File(File parent, String child);
5. – File(String pathname);
6. – File(String parent, String child);
7. – File(URI uri);



1. 数据流：一组有顺序的、有起点和终点的字节集合，输入输出流都是数据流，import java.io.\* 。流是程序和I/O设备之间的数据传输通道，是数据传输的抽象表达，与具体设备无关。通常“流”指字节流，“读写器”指字符流。

输入数据流：只读,不写，将数据从文件、标准输入或其他外部设备加载到内存。

– 字节流：是抽象类InputStream的子类；

– 字符流：是抽象类Reader的子类；

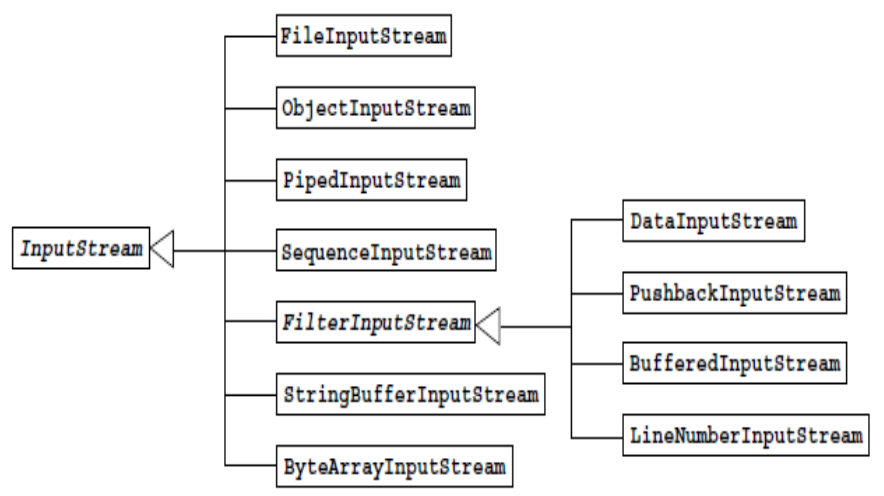
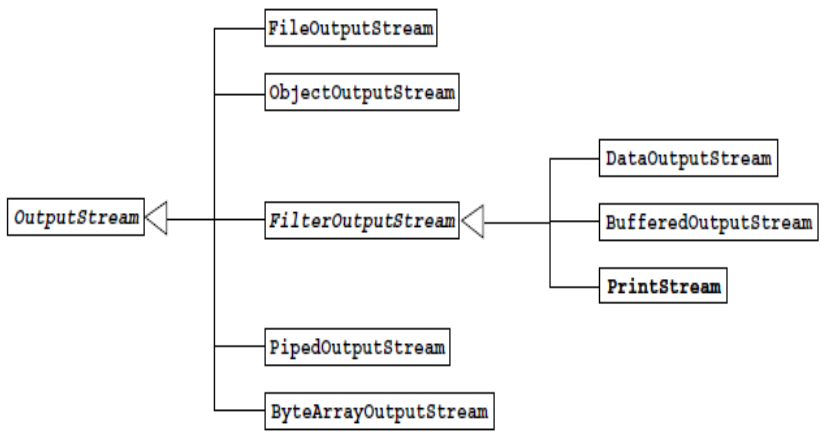
输出数据流：只写,不读，将内存中的数据保存到文件中或传输给输出设备。

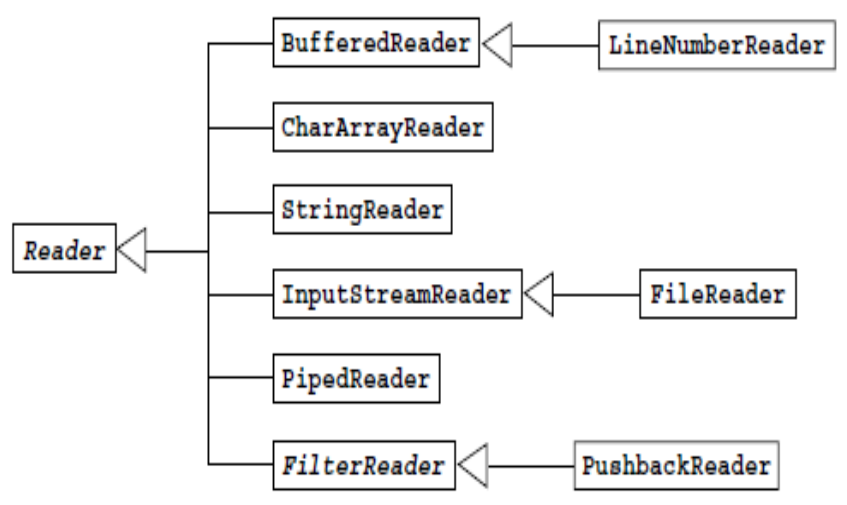
– 字节流：是抽象类OutputStream的子类；

– 字符流：是抽象类Writer的子类；

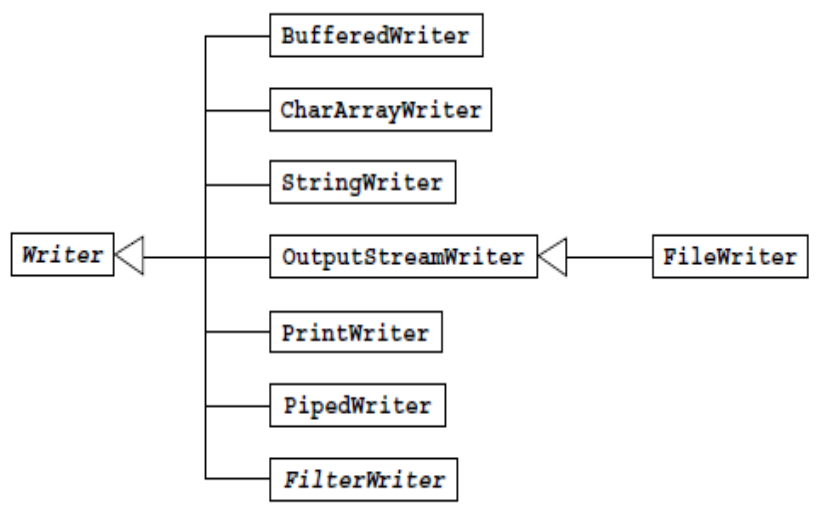
1. 流可以被认为是从源或到接收器的数据流，从源头到接收器。源流启动数据流，即输入流；接收流终止数据流，即输出流。源和汇都是节点流，节点流的类型是文件、内存和线程或进程之间的管道。

Java支持两种流：字符和字节。字符的输入输出由读写器处理。字节的输入输出由输入流和输出流处理：术语流指字节流，术语读写器指字符流。

1. InputStream类层次结构
2. FileInputStream：从本地文件中读出数据。
3. PipedIntputStream：用于管道输入/输出时从管道中读取数据
4. FilterInputStream：其子类PushbackInputStream，BufferedInputStream读取数据时进行缓冲，提高效率，增加特殊功能。
5. DataInputStream：实现了java.io中的DataInput接口，读取数据的同时，可以对数据进行格式处理。能用来读取java中的基本数据类型。
6. ByteArrayInputStream：包含一个内存缓冲区，用于从内存中读取数据。
7. SequencedInputStream：把两个以上InputStream输入流对象转换为单个inputStream输入流对象。
8. ObjectInputStream：用来读取对象
9. AudioInputStream：Audio的输入输出
10. OutputStream类层次结构
11. FileOutputStream：用于向本地文件中写入数据。
12. PipedOutputStream：用于管道输入/输出时，把数据向管道输出
13. DataOutputStream：提供了对java的基本数据类型的支持
14. PrintStream：提供了向屏幕输出有格式数据的很多方法
15. Reader类层次结构



1. Writer类层次结构



Reader类和Writer类中的大部分方法与InputStream类和OutputStream类中的对应方法名称相同，只是读取或写入的数据是字符、字符数组和字符串等

输入输出流中多个方法都可能抛出IOException异常，应放在try…catch块中捕获和处理异常。

|  |  |  |  |
| --- | --- | --- | --- |
|  | InputStream | Reader | 说明 |
| 基本方法 | int read() | int read() | 抽象方法，读取并返回一个字节/字符 |
| int read(byte[] buffer) | int read(char[] cbuf) | 读取多个字节/字符放入数组中并返回实际读取字节/字符数量，通常读取数量为数组长度 |
| int read(byte[], int offset, int length) | int read(char[] cbuf, int offset, int length) | 读取length个字节/字符，放到以下标offset开始的数组中，返回实际读取字节/字符数量 |
| 其他方法 | void close() | void close() | 关闭流，释放内存资源 |
| long skip(long n) | long skip(long n) | 跳过n个字节/字符不读，返回实际跳过的字节/字符数量 |
| boolean markSupported() | boolean markSupported() | 当前流是否支持读指针的记录功能 |
| void reset() | void reset() | 把读指针重新指向用mark方法所记录的位置 |
| void mark(int readlimit) | void mark(int readAheadLimit) | 记录当前指针所在位置，参数表示再读多少字节/字符后指针失效 |
| int available() |  | 返回尚未读取的字节数量 |
|  | boolean ready() | 判断是否准备好被读取 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | OutputStream | Writer | 说明 |
| 基本方法 | void write(int c) | void write(int c) | 写入一个字节/字符数据，为c的低16位 |
| void write(byte[] buffer) | void write(char[] buffer) | 将一个字节/字符数组中的数据写入输出流 |
| void write(byte[], int offset, int length) | void write(char[], int offset, int length) | 将一个字节/字符数组中的从offset位置开始的length个数据写入输出流 |
|  | void write(String string) | 将一个字符串中的字符写入输出流 |
|  | void write(String string, int offset, int length) | 将一个字符串从offset开始的length个字符写入输出流 |
| 其他方法 | void close() | void close() | 关闭流，释放内存资源 |
| void flush() | void flush() | 将输出流中缓存的数据全部写出到目的地 |

1. InputStream/OutputStream：
2. InputStream是抽象类，不能通过“new InputStream()”构造实例，可以通过构造InputStream子类的实例来获得InputStream类型的实例。

三种read()：int read()；int read(byte b[])；int read（byte[], int off, int len)

available() 与read()混用，读之前可以知道有多少字符；

mark() 与reset()混用，可重复读取输入流所指定的字节数据

1. OutputStream是抽象类，不能通过“new OutputStream()”构造实例，可以通过构造OutputStream子类的实例来获得OutputStream类型的实例，声明了输出流的基本操作
2. FileInputStream / FileOutputStream：
3. FileInputStream：InputStream的子类，用于从本地文件中读取数据，有三个构造方法，不支持缓冲和做标记
4. FileOutputStream：OutputStream的子类，有五个构造方法，常用FileOutputStream(String name);和FileOutputStream(String name, boolean append);

对文件内容进行操作的基本步骤

1. 创建该文件对应的实例。如FileInputStream f = new FileInputStream(“...”)；
2. 对该文件进行读写（I/O）操作
3. 最后关闭文件，释放所占用的资源
4. 过滤流类：为输入/输出流提供通用接口，将一个流连接到另一个流的尾部，得到满足程序要求的很长的输入和输出过滤器。为防止多线程同时对一个I/O流进行操作，过滤流类实现了对象的同步机制，某一时刻只能一个线程访问它。
5. FileterInputStream：过滤器输入流，抽象类，是其它带过滤器的输入流的超类，重写了InputStream的所有方法。

构造方法：FileterInputStream(InputStream in);

成员变量：protected InputStream in，在构造方法中会对其赋值this.in = in;

1. FilterOutputStream：过滤器输出流，抽象类，是其它带过滤器的输出流的超类，重写了OutputStream的所有方法。过滤器输出流位于一个OutputStream之上

构造方法：FileterOutputStream(OutputStream in);

成员变量：protected OutputStream out，在构造方法中会对其赋值this.out = out;

1. 带缓存的输入流和输出流：BufferedInputStream和BufferedOutputStream，创建实例时会在内存开辟一个512字节的缓存，用于存放数据流中的数据。
2. 这两个类用来连接别的I/O流，借助字节数组缓存，在读取或存储数据时将一个较大数据块读入内存或将内存中较大数据块一次性写入指定文件，以提高读/写数据的效率。
3. flush()：通过BufferedOutputStream写数据时，数据不直接写入数据流，而是先写入缓冲区，当缓冲区满时，数据才会写入数据流。缓冲区未满时，可用flush()将缓冲区的数据强制全部写入输出流。
4. 第一次读取数据按块读入缓冲区，之后如果有数据读取操作则直接访问缓冲区。缓冲区除了可以提高性能外，还提供了标记读取位置（mark()）和重置标记（reset()）的支持。
5. 最优的缓冲区大小通常根据操作系统、内存空间、机器的其他配置情况来确定。通常缓冲区大小应为内存页或磁盘块等的整数倍，以避免页或块的浪费。
6. 主要方法
7. BufferInputStream：

|  |  |
| --- | --- |
| BufferedInputStream(InputStream in);  BufferedInputStream(InputStream in, int size); | 构造方法 |
| int available() | 返回可以从这个InputStream读到的字节数 |
| int read() | 读下一个字节 |
| int read(byte[] b, int offset, int length) | 从offset开始读最多length个字节到b[]里去 |
| void close() | 关闭这个输入流 |
| void mark(int limit) | 在当前位置做一个标记 |
| boolean markSupported() | 测试是否支持mark()和reset()方法 |
| void reset() | 返回到最后一个mark()做的标记处 |
| long skip(long n) | 跳过n个字节 |

1. BufferOutputStream

|  |  |
| --- | --- |
| BufferedOutputStream(OutputStream out);  BufferedOutputStream(OutputStream out, int size); | 构造方法 |
| void flush() | flush这个BufferedOutputStream |
| void write(int b) | 将b的低8位写到输出流 |
| void write(byte[] b, int offset, int length) | 从offset开始的b[]最多写length个字节到输出流 |

1. DataInputStream与DataOutputStream：读取于存储基本数据类型的数据。且数据流的存储格式采用统一的形式，与具体的计算机无关，构造方法以输入/出流的实例引用为参数。
2. PushbackInputStream：回压输入流，提供了几个unread()方法，可以把读过的几个字节数据退回到输入流中，也可以回压别的字节数据到输入流中。
3. 构造函数：

– public PushbackInputStream(InputStream in);

– public PushbackInputStream(InputStream in, int size);

1. Unread方法：

– public void unread(int b);

– public void unread(byte[] b);

– public void unread(byte[]b,int off, int len);

1. PrintStream：打印流，System.out是指向PrintStream实例的一个引用。提供的输出方法分为write，print，println三类，System.out就是指向PrintStream实例的一个引用。成员方法一般不会抛出异常。具有自动强制输出(flush)功能，即输出回车换行时，缓存中的数据会全部自动写入指定的文件或在标准输出窗口中显示。
2. 对象流：ObjectInputStream和ObjectOutputStream类，其属性本身就包含各种类型变量或其他类对象，用来创建对象输入/输出流，读写文件内容。使用对象流读写对象时要保证对象时串行化的。
3. 创建对象流：

– Public ObjectInputStream(InputStream in);

– Public ObjectOutputStream(OutputStream in);

1. 读写对象流：

– Public final void writeObject(Object obj) throws IOException

– Public final void readObject() throws OptionalDataException, ClassNotFoundException, IOException

1. 串行化(Serializable)：指对象通过把自己转化为一系列字节，记录状态数据，以便再次利用的过程。 是java.io包中定义的一个接口。对不希望串行化的对象可用transient修饰。一个类要具有可串行化的特性就必须实现接口java.io.Serializable。对于可以串行化的对象可以用java.io.ObjectOutputStream输出，用java.io.ObjectInputStream读入。
2. 管道流：管道输入流PipeInputStream和管道输出流PipeOutputStream成对出现，同时使用并相互连接，形成一个数据通信管道。管道流是InputStream和OutputStream类的子类，能作为别的输入/输出流的参数，即连接到别的流，从而增加其输入/输出的功能。管道数据流主要用于线程间的通信。
3. 创建管道流
4. PipedInputStream pis = new PipedInputStream();

PipedOutputStream pos = new PipedOutputStream(pis);

1. PipedOutputStream pos = new PipedOutputStream();

PipedInputStream pis = new PipedInputStream(pos);

1. PipedInputStream pis = new PipedInputStream();

PipedOutputStream pos = new PipedOutputStream();

pis.connect(pos);

1. 管道输入流中增加了一个方法，用来接收一个字节数据b，如果管道中没有数据，该方法将阻塞。– Protected void receive(int b) throws IOException
2. 被连接的管道流必须没有与别的管道流连接，否则会抛出IOException异常，管道输出流输出数据到管道，管道输入流在管道中读取数据。
3. Reader和Writer类是抽象类，处理字符流，读写的数据是字符、字符数组和字符串。FileReader和FileWriter分别是Reader和Writer的子类，PrintWriter是Writer的子类。
4. 方法对应

|  |  |
| --- | --- |
| 字节流类 | 字符流类 |
| InputStream | Reader |
| OutputStream | Writer |
| FileInputStream | FileReader |
| FileOutputStream | FileWriter |
| ByteArrayInputStream | CharArrayReader |
| ByteArrayOutputStream | CharArrayWriter |
| StringBufferedInputStream（已不用） | StringReader |
|  | StringWriter |
| BufferedInputStream | BufferedReader |
| BufferedOutputStream | BufferedWriter |
| PipedInputStream | PipedReader |
| PipedOutputStream | PipedWriter |
| PrintStream | PringWriter |

1. Reader和Writer：抽象类，对应读写器的一些基本操作，以字符为单位访问文件，读写器处理的是字符流流。
2. 读写器构造出来的一般是文本文件，往往需要更大的存储空间。
3. 构造方法：Reader(); Reader(Object lock); Writer(); Writer(Object lock);
4. InputStreamReader / OutputStreamWriter
5. InputStreamReader继承自Reader类，通过其read()方法从字节输入流中读取一个或多个字节数据转换为字符数据，不是缓冲流；OutputStreamWriter继承自Writer类，将字符输出流转为字节流输出。InputStreamReader类和OutputStreamWriter类都可以接一个缓冲流来提高效率。
6. InputStreamReader方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| InputStreamReader(InputStream in); | 构造方法 |
| InputStreamReader(InputStream in, Charset cs); |
| InputStreamReader(InputStream in, CharsetDecoder dc); |
| InputStreamReader(InputStream in, String charset); |
| void close(); | 关闭输入流，释放系统资源 |
| String getEncoding(); | 返回系统的encoding名 |
| int read(); | 读一个字符 |
| int read(char[] cbuf, int offset, int length); | 从cbuf[]的offset开始读入length个字符 |
| boolean ready(); | 检查输入流是否已经准备好了 |

1. OututStreamReader方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| OutputStreamReader(OutputStream out); | 构造方法 |
| OutputStreamReader(OutputStream out, Charset cs); |
| OutputStreamReader(OutputStream out, CharsetEncoder enc); |
| OutputStreamReader(OutputStream out, String charset); |
| void close(); | 关闭输出流，释放系统资源 |
| void flush(); | 清空输出流 |
| void write(int c); | 写字符c |
| void write(char[] cbuf, int offset, int length); | 从cbuf[offset]开始读入length个字符 |
| void write(String s, int offset, int length); | 写一个子串 |
| String getEncoding(); | 返回系统的encoding名 |

1. InputStreamReader实现了字节流到字符流的转变，常与BufferedReader合用：

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

1. InputStreamWriter实现了字符流到字节流的转变，常与BufferedWriter合用：

Writer out = new BufferedWriter(new OutputStreamWriter(System.out));

1. 如果字符流不是来自本地，可能会编码不一样，直接读取会读出不正确字符

– ir=new InputStreamReader(is,“utf-8”);

1. FileReader / FileWriter
2. FileReader：是Reader的子类，用于从字符文件读取字符
3. FileWriter：是Writer的子类，用于向字符文件输出字符数据
4. BufferedReader / BufferedWriter
5. BufferedReader

|  |  |
| --- | --- |
| 方法 | 说明 |
| BufferedReader(Reader in); | 构造方法 |
| BufferedReader(Reader in, int size); |
| void close(); | 关闭输入流，释放系统资源 |
| int read(); | 读一个字符 |
| int read(char[] cbuf, int offset, int length); | 从cbuf[]的offset开始读入length个字符 |
| String readLine(); | 读一行文本 |
| void mark(int limit); | 在当前位置做一个标记 |
| boolean markSupported(); | 测试是否支持mark()和reset()方法 |
| void reset(); | 返回到最后一个mark()做的标记处 |
| boolean ready(); | 判断输入流是否准备好了 |
| long skip(long n); | 跳过n个字节 |

1. BufferedWriter

|  |  |
| --- | --- |
| 方法 | 说明 |
| BufferedWriter(Writer out); | 构造方法 |
| BufferedWriter(Writer out, int size); |
| void close(); | 关闭输入流，释放系统资源 |
| void flush(); | 清空数据流 |
| void write(int c); | 写字符c |
| void write(char[] cbuf, int offset, int length); | 从cbuf[offset]开始读入length个字符 |
| void write(String s, int offset, int length); | 写一个子串 |
| void newLine(); | 换行 |

1. CharArrayReader / CharArrayWriter：将输入与输出面向一个char型的数组，都有成员变量protected char[] buf和protected int count（这个buf就是数据缓冲区）
2. CharArrayReader：将数据读入缓冲区buf
3. CharArrayWriter：将数据写入缓冲区
4. PrintWriter：Writer的子类，
5. PrintWriter不会抛出异常，可通过PrinterWriter的成员方法：public boolean checkError(); 检查是否有错误发生。
6. PrinterWriter实现了PrinterStream中所有的print成员方法，所以PrintWriter也可以直接输出各种各样的数据，而不必转换成字符串类型或其它类型。
7. PrintWriter的自动强制输出(flush)功能与PrintStream的响应功能有所不同。当用PrintWriter时，只有调用了其成员方法println()才可能自动强制输出。
8. LineNumberReader类：继承自BufferedReader类，也是缓冲字符输入流，但支持更多的行数处理功能。
9. 小结
10. Java中有数据传输的地方都用到I/O流（通常是文件、网络、内存、标准输入输出等）
11. File, File(Input/Output)Stream, RandomAccessFile是处理本地文件的类。
12. Data(Input/Output)Stream是一个过滤流的子类，可读写各种基本数据，在文件和网络中经常使用，如：readByte, writeBoolean等。
13. Buffered(Input/Output)Stream在数据送到目的之前先缓存，达到一定数量时再送到目的地，以提高程序的运行效率。
14. Piped(Input/Output)Stream适合于一个处理的输出作为另一个处理的输入的情况。
15. 多线程程序设计）
16. 概念

程序(program)：一段静态代码，对数据描述与操作的代码的集合，应用程序执行的蓝本。

进程(process)：程序的一次执行过程，是系统运行程序的基本单位。程序是静态的，进程是动态的。系统运行一个程序即是一个进程从创建、运行到消亡的过程。

多任务(multi task)：在一个系统中可以同时运行多个程序，即有多个独立运行的任务，每个任务对应一个进程。

线程：比进程更小的执行单位，一个进程中可以包含多个线程。

1. 一个进程在执行过程中，为同时完成多项操作，可产生多个线程，形成多条执行线索。
2. 每个线程都有自身产生、存在和消亡的过程。
3. 一个进程中的所有线程都在该进程的虚拟地址空间中，使用该进程的全局变量和系统资源。
4. 操作系统给每个线程分配不同的CPU时间片，在某一时刻，CPU只执行一个时间片内的线程，多个线程在CPU内轮流执行。
5. 多进程
6. 多进程环境中每个进程既包括其所要执行的指令，也包括执行指令所需的系统资源, 如CPU、内存空间、I/O端口等，不同进程所占用的系统资源相对独立；
7. 多线程环境中每个线程都隶属于某一进程，由进程触发执行，属于同一进程的所有线程共享该进程的系统资源；
8. 一个进程就是一个应用程序，有自己的入口和出口；线程本身没有入口和出口，其自身也不能独立运行，它隶属于某个进程，由进程触发执行，完成其任务后自动终止，或由进程使之强制终止。
9. 多线程
10. 多线程程序设计：单个程序包含并发执行的多个线程。多线程程序执行时，该程序对应的进程中有多个控制流在同时执行，即具有并发执行的多个线程；如PV操作，银行问题，网络聊天，Web Server接受客户端的请求问题。
11. 线程之间共享相同的内存单元，所以线程之间的切换速度和通信远比进程之间快。
12. 多个线程轮流抢占CPU资源而运行时，微观上，一个时间只能有一个作业被执行，在宏观上可使多个作业被同时执行，即等同于多台计算机同时工作，使系统资源特别是CPU的利用率得到提高，从而可以提高整个程序的执行效率。
13. 每个线程都有一个独立的程序计数器和方法调用栈（method invocation stack）

– 程序计数器：也称为PC寄存器，当线程执行一个方法时，程序计数器指向方法区中下一条要执行的字节码指令。

– 方法调用栈：简称方法栈，用来跟踪线程运行中一系列的方法调用过程，栈中的元素称为栈桢。每当线程调用一个方法，就会向方法栈压入一个新桢，桢用来存储方法的参数、局部变量和运算过程中的临时数据。

1. 线程调度：通常是抢占式而不是分时间片式，即线程先抢到CPU资源则先运行。
2. 一个线程获得执行权后，将持续运行下去，直到运行结束或阻塞，或者有另一个高优先级线程就绪（这种情况称为低优先级线程被高优先级线程所抢占）。
3. 所有被阻塞的线程按次序排列，组成一个阻塞队列。例如:

–因为需要等待一个较慢的外部设备，例如磁盘或用户。

–让处于运行状态的线程调用Thread.sleep()方法。

–让处于运行状态的线程调用另一个线程的join()方法。

1. 所有就绪但没有运行的线程，根据其优先级排入一个就绪队列。当CPU空闲时，如果就绪队列不空，就绪队列中第一个具有最高优先级的线程将运行；当一个线程被抢占而停止运行时，它的运行态被改变并放到就绪队列的队尾；一个被阻塞（可能因为睡眠或等待I/O设备）的线程就绪后通常也放到就绪队列的队尾。
2. 多线程运行时，各个线程通过竞争CPU时间而获得运行机会，各线程得到和占用CPU的时间是不可预测的，一个运行时线程在什么地方被暂停是不确定的。
3. 优先级
4. 线程的调度按其优先级高低顺序执行，同样优先级的线程遵循“先到先执行”原则
5. 线程创建时，继承父线程的优先级
6. 线程优先级：范围 1~10（级），数值越大，级别越高
7. Tread类定义的3个常数

MIN\_PRIORITY 最低(小)优先级（值为1）

MAX\_PRIORITY 最高(大)优先级（值为10）

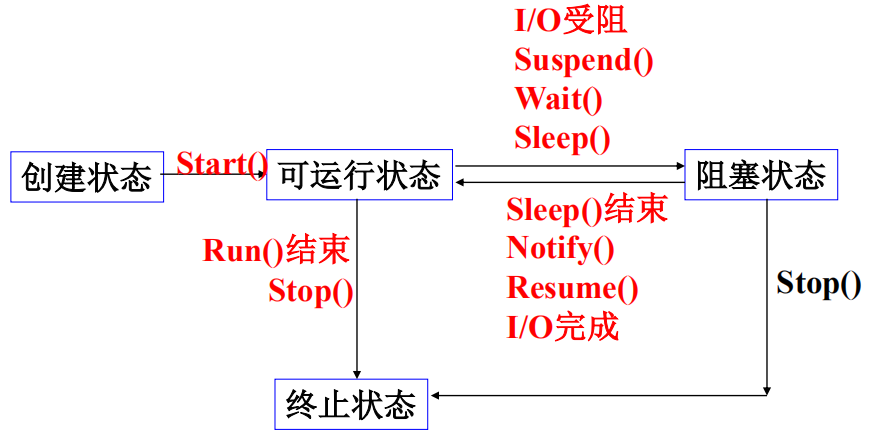
NORM\_PRIORITY 默认优先级（值为5）

1. 常用方法

getPriority()：获得线程的优先级

setPriority()：设置线程的优先级

1. 主线程
2. 每当用java命令启动一个Java虚拟机进程（Application应用程序），Java虚拟机就会创建一个主线程，该线程从程序入口main()方法开始执行。
3. 在主线程中创建Thread类或其子类对象时，就创建了一个线程对象。Programmer可以控制线程的启动、挂起与终止。
4. 线程的状态：（多线程运行时）新建、就绪runnable、运行running、阻塞blocked、终止
5. 新建：一个Thread类或其子类对象被创建，此时它已经有了相应的内存空间和其他资源。如 Thread myThread = new Thread();
6. 就绪：调用start()方法启动处于新建状态的线程后，将进入线程队列等待CPU服务，此时它已经具备了运行的条件，轮到它享用CPU资源时，就可以脱离创建它的主线程，开始自己的生命周期。
7. 运行：就绪态的线程被调度并获得处理器资源时，便进入运行状态。每个Thread类及其子类的对象都有一个run()方法，定义了这个线程的操作和功能。当线程对象被调用执行时，它将自动调用本对象的run()方法，从第一句开始顺序执行。
8. 阻塞：一个正在执行的线程暂停自己的执行而进入的状态。引起线程由运行状态进入阻塞状态的可能情况及使其返回就绪态的方法：
9. –该线程正在等待I/O操作的完成——等待该I/O操作完成
10. –调用了该线程的sleep()方法——等待该休眠结束后自动脱离阻塞态
11. –调用了wait()方法——调用notify()或notifyAll()方法
12. –让处于运行状态的线程调用另一个线程的join()方法
13. 终止：
14. –自然终止：线程完成了自己的全部工作
15. –强制终止：在线程执行完之前，调用stop()或destroy()方法终止线程
16. 状态转换：



1. Java中的多线程是建立在Thread类，Runnable接口的基础上的，通常有两种办法让我们来创建一个新的线程：

–创建一个Thread类,或者一个Thread子类的对象

–创建一个实现Runnable接口的类的对象

1. Thread类
2. 构造方法（8个）

Thread(); Thread(Runnable target); Thread(Runnable target, String name);

Thread(ThreadGroup group, Runnable target); Thread(ThreadGroup group, String name);

Thread(String name); Thread(ThreadGroup group, Runnable target, String name);

Thread(ThreadGroup group, Runnable target, String name, long stackSize);

说明：name为线程名称；target为要运行的线程对象；group为线程组名称；stackSize为堆栈的大小

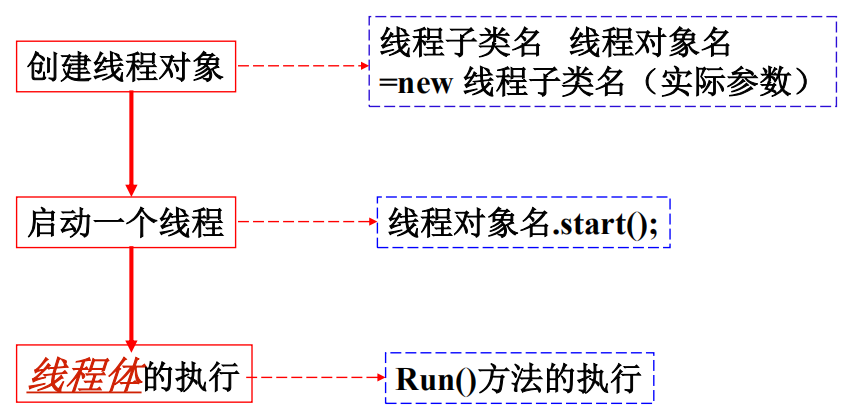
1. 常用方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| currentTread() | 返回当前运行的Thread对象 |
| setName() | 设置线程名称 |
| getName() | 返回线程名称 |
| setPriority() | 设置线程优先级 |
| getPriority() | 返回线程优先级 |
| start() | 启动一个线程 |
| run() | 线程体，由start()调用，当run()方法返回时，当前线程结束 |
| stop() | 使调用它的线程立即停止执行 |
| isAlive() | 如果线程已被启动而未被终止则返回true，若是新创建的或已被终止则返回false |
| sleep(int n) | 使线程睡眠n毫秒，之后再次运行 |
| yield() | 将CPU控制权主动交到下一个可运行线程 |
| join() | 引起线程等待，直至join所调用的线程结束 |
| suspend() | 使线程挂起，暂停运行 |
| resume() | 恢复挂起的线程，使其处于可运行状态 |
| wait() | 使线程等待 |
| notify() | 随机唤醒该对象等待池中的一个线程 |
| notifyAll() | 唤醒该对象等待池中的所有线程 |

1. sleep()与yield()
2. sleep()：静态实例方法，使线程转入阻塞态，具有更好的可移植性；
3. yield()：静态实例方法，使线程转入就绪态，主要用于测试阶段，认为提高程序的并发性。
4. wait()与sleep()

同：都能使线程等待而停止运行

异：sleep()不会释放对象的锁，而wait()方法进入等待时会释放对象的锁，使得其它线程能对这些枷锁的对象进行操作。

1. join()方法：使当前正在运行的线程暂停下来，等待指定的时间后或等待调用该方法的线程结束后，再恢复运行。
2. 构造方法：
3. public final void join() throws InterruptedException;
4. public final void join(long millis)throws InterruptedException;
5. public final void join(long millis,int nanos)throws InterruptedException;
6. 方法join()将引起现行线程等待，直至方法join()所调用的线程结束。
7. 创建线程
8. 用线程类（Thread的子类）创建线程

线程体：public void run()方法，是整个线程的核心，线程所要完成任务的代码都定义在线程体中，不同功能的线程之间的区别就在于它们的线程体不同。

1. 用Runnable接口创建线程
2. 从根本上讲，任何实现线程功能的类都必须实现Runnable接口
3. Thread(Runnable target); Thread(Runnable target, String name);
4. Runnable接口中只定义了一个方法run()，即方法体。
5. 适用情况：Java只允许单继承，如果一个类已经继承了Thread就不能再继承其他类；在除了run()之外不打算再重写Thread类的其他方法的情况
6. 主线程与其他线程共存，可以用join()方法保证主线程最后结束。（见OOP课件多线程程序设计-P54-P59）
7. 终止线程
8. 当线程执行完run()方法，将自动终止运行
9. 实际编程中，定义一个标志变量，通过程序改变标志变量的值，从而控制线程从run()方法中退出。
10. 创建用户多线程的步骤
11. 方式一
12. 创建一个Thread类的子类；
13. 在子类中将希望该线程做的工作写到run()里；
14. 生成该子类的一个对象；
15. 调用该对象的start()方法。
16. 方式二
17. 创建一个实现Runnable接口的类；
18. 在该类中将希望该线程做的工作写到run()里面；
19. 生成该类的一个对象；
20. 用上述对象去生成Thread类的一个对象；
21. 调用Thread类的对象的start()方法；
22. 方式三
23. 创建一个实现Runnable接口的类；
24. 在该类中将希望该线程做的工作写到run()里；
25. 写一个start()方法，在里面创建Thread，调用start()；
26. 生成Runnable类的一个对象；
27. 调用该对象的start()方法。
28. 同步synchronized
29. 同步：一种保证共享资源完整性的手段，在代表原子操作的程序代码前加上synchronized标记，这样的代码成为同步代码块。
30. synchronized关键字可以用在成员方法上、静态方法上、语句块上。
31. synchronized的用法：
32. public synchronized void write();

– 由synchronized关键字锁定的是一个对象,而不是一个方法。

– 一个线程调用某个对象的synchronized方法,就锁定了这个对象，直到它从这个方法中退出,才释放了这个锁定。因此一旦一个对象被某个线程锁定了,它的所有synchronized方法就都不允许别的线程调用了。

– 如果一个线程调用该方法，则方法体中访问的对象都被锁定，不允许其他线程访问，以解决共享资源的访问冲突问题。

1. public static synchronized intgetValue();

一个线程调用一个类的static synchronized方法,就锁定了这个类对象,也就控制了所有static synchronized方法。即类中所有的静态同步方法在每个时刻只有一个可以执行。

1. Synchronized(obj){…}

一个线程进入由synchronized修饰的程序块,就锁定了由obj指定的那个对象。

1. 一个对象在某一时刻只能被一个线程锁定；一个类可以有多个对象，不同的对象可以被不同的线程锁定；一个线程可以锁定多个对象。
2. synchronized方法或synchronized区段内的代码在同一时刻下可有多个线程执行，只要是对不同的对象调用该方法就可以。
3. 线程通信：线程之间交换信息，简单通信可用wait()、notify()方法
4. wait()：执行该方法的线程释放对象的锁，Java虚拟机将其放到该对象的等待池中等待其他线程将它唤醒。
5. notify()：执行该方法的线程随机唤醒在对象的等待池中等待的一个线程，Java虚拟机从对象的等待池中随机选择一个线程，把它转到对象的锁池中。
6. 一般每个共享对象的互斥锁存在两个队列：锁等待队列和锁申请队列。锁申请队列中的第一个线程可以对该共享对象进行操作，锁等待队列中的线程在某些情况下将移入锁申请队列中。
7. wait()，notify()，notifyAll()都是与同步相关联的方法，必须在已持有锁的情况下执行，即只能出现在synchronized修饰的方法中，不同步的方法或代码中则使用sleep()暂停线程。一个“在同步方法中等待”的线程自己不能恢复运行，只能等另一个也进入了该对象的synchronized方法的线程调用notify和notifyAll后才可能恢复运行。

wait()：释放已持有的锁，进入等待队列

notify()：唤醒wait队列中的第一个线程并把它移入锁申请队列

notifyAll()：唤醒wait队列中的所有线程并全部移入锁申请队列

1. 线程间要传送的数据较多时，必须使用共享主存、管道流等通信方式
2. 管道流通信
3. 管道：用来把一个程序的输出连接到另一个程序的输入，java.io中提供了类
4. 管道的输入/输出部件：PipedInputStream / PipedOutputStream
5. 构造方法中，管道的输入/输出流的参数为对应管道的输出/输入流
6. 管道输入/输出流还可以用connect()方法进行连接
7. 死锁：由于程序设计不合理而造成程序中所有的线程都陷入等待态或阻塞态，发生于多线程竞争使用多资源的程序中，几个线程循环等待另一个线程所持有的锁。
8. 死锁不是由于资源短缺造成的
9. 每个线程都不能继续运行，除非另一线程运行完同步程序块。
10. 因为哪个线程都不能继续运行，所以哪个线程都无法运行完同步程序块。
11. 另一种情况是在wait、notify结构中，如果所有的线程都wait了，就不会有线程来notify
12. 后台线程：又称守护线程，为其它线程提供服务的线程。
13. 守护线程不是应用程序的核心部分，当应用程序的所有非守护线程终止运行时，即是仍然有守护线程在运行，应用程序也将终止。只要有非守护线程在运行，应用程序就不会终止。
14. 主线程和由前台线程创建的线程默认为前台线程。
15. 通过调用isDaemon()方法来判断一个线程是否是守护线程
16. 通过调用setDaemon()方法将一个线程设为守护线程
17. 定时器Timer
18. TimerTask类表示定时器执行的一项任务
19. Timer类的schedule(TImerTask task, long delay, long period)方法用于设置定时器需要定时执行的任务。参数task表示任务；delay表示延迟执行的时间(ms)，即多久后开始执行第一次；period表示每次执行任务的间隔时间(ms)。
20. 多线程程序对系统的影响
21. 线程需要占用内存
22. 线程过多，会消耗大量CPU时间来跟踪线程
23. 多线程同时访问共享资源的问题，协调不好会发生死锁和资源竞争
24. 同一个任务的所有线程都共享相同的地址空间和任务的全局变量，需要考虑
25. Java网络程序设计
26. 计算机网络工作模式
27. 专用服务器结构(Server-Based)
28. 客户机/服务器模式(Client/Server. C/S)：一台或几台较大的计算机集中进行共享数据库的管理和存取，称为服务器；其他的应用处理工作分散到网络中其它微机上去做，构成分布式的处理系统。
29. 对等式网络(Peer to Peer)：每个工作站既可以起到客户机的作用，也可以起到服务器的作用。
30. 基础知识
31. 网络通信协议：计算机网络中实现通信的必须的约定，如对速率、传输代码、代码结构、传输控制步骤、出错控制等制定的标准。
32. 网络通信接口：为使两个节点之间能进行对话，必须在它们之间建立通信工具（即接口）。接口包括两部分：
33. 硬件装置：实现节点之间信息传递
34. 软件装置：规定双方进行通信的约定协议
35. 网络进程通信协议：网络进程之间通信必须遵循的预定的规则
36. TCP/IP：一组在Internet上的不同计算机之间进行通信的协议的总称，从下往上为四层结构：网络层、网络层、传输层、应用层。TCP/IP由应用层的HTTP、FTP、SMTP和传输层的TCP（传输控制协议）、UDP（用户数据报协议）以及网络层的IP（Internet协议）等一系列协议组成。
37. 数据的封装：发送方数据在网络模型的各层传送过程中加入头尾的过程
38. 数据的拆封：接受方收到数据后去除相应的头尾的过程
39. 网络进程：在网络结点计算机上运行的程序
40. 网络编程：指利用不同层次的通信协议提供的接口实现网络进程通信的编程
41. Java小应用程序属于应用层的网络编程，主要用于交互式的网页设计
42. 传输层的网络进程通信机制是进行网络编程的基础
43. 在网络上，计算机通过IP地址或主机名进行标识，使得位于不同地理位置的计算机可以互相访问和通信
44. 协议：在网络上进行通信要遵循的规则，最常用的传输层的网络通信协议是TCP和UDP，它们传输的都是一个byte stream/字节型的数据流
45. TCP：传输控制协议，是面向连接的通信协议，两个进程之间会建立连接，数据以流的形式顺序传播；
46. UDP：用户数据协议，是无连接通信协议，两个进程之间不建立特定的连接，不对数据到达的顺序进行检查。
47. 网络资源定位器（URL，Uniform Resource Locator）可以指向网络上的各种资源。通过网络资源定位器可以获取网络上的资源。
48. IP地址：IP由32位数字表示，通常分为4部分。本地回路的IP地址localhost为127.0.0.1

端口号port：逻辑上的数据传输通道，TCP/IP协议引入端口是为了确定网络数据流向的进程端口号的范围是0~65535，0~1023的端口用于一些知名的网络服务和应用（我们一般不用）。端口同一台网络计算机的一个特定进程关联，与进程建立的套接口绑定在一起。

1. 网络资源可以包括网络上的简单对象（如网络上的路径和文件）以及一些复杂对象（如数据库或搜索引擎）
2. URL：WWW资源统一资源定位器，规范了WWW资源网络定位地址的表示方法
3. URL通常由5个部分组成：协议、主机名、端口号、文件、引用
4. URL的基本表示格式：protocol://hostname:port/resourcename#anchor
5. –protocol：使用的协议，它可以是http、ftp、news、telnet等。
6. –hostname：主机名，指定域名服务器(DNS)能访问到的WWW服务的计算机，例www.sun.com
7. –port：所连接的端口号，可选，如缺省，将连接到协议缺省的端口。
8. –resourcename：资源名，是主机上能访问到的目录或文件。
9. –anchor：标记，可选，指定文件中有特定标记的位置。
10. socket：套接字，是网络驱动层提供给应用程序编程的接口和一种机制，在应用程序中创建，通过一种绑定机制与驱动程序建立关系，告诉自己对应的IP和port。
11. Java中网络编程类位于java.net包中
12. URL、URLConnection、Socket、ServerSocket类基于TCP协议，Socket类用于TCP通信的服务器和客户端，ServerSocket类用于TCP通信的服务器端
13. DatagramPacket、DatagramSocket、MulticastSocket类基于UDP协议，DatagramSocket类用于UDP通信。
14. TCP网络程序
15. 利用TCP协议进行通信的两个应用程序（即服务器程序和客户机程序）有主从之分，
16. TCP网络程序的工作原理
17. 客户端发出连接请求
18. 服务器端接收请求，并创建新的socket
19. 两个socket之间建立专线连接
20. TCP客户端程序与服务器端程序的交互过程
21. 服务器端创建一个ServerSocket，然后调用accept()方法等待客户来连接；
22. 客户端程序创建一个socket并请求与服务器建立连接；
23. 服务器端接收客户的连接请求，并创建一个新的Socket与该客户建立专线连接；
24. 建立了连接的两个socket在一个单独的线程(由服务器程序创建)上对话；
25. 服务器开始等待新的连接请求，当新的连接请求到达时，重复步骤2-5的过程。
26. InetAddress类：用于表示计算机IP地址的类
27. getByName(String host)：通过域名来构造实例对象
28. getByAddress(byte[] addr)：通过4字节的网络地址构造实例对象
29. ServerSocket类
30. 构造方法

|  |  |
| --- | --- |
| 构造方法 | 说明 |
| ServerSocket(); | 不能直接使用，需要调用bind()方法 |
| ServerSocket(int port); | 绑定到指定的port上，连接队列默认50 |
| ServerSocket(int port, int backlog); | 指定最大连接队列 |
| ServerSocket(int port, int backlog, InetAddress bindAddr); | 增加了指定相关的IP地址 |

1. 常用方法

|  |  |
| --- | --- |
| 常用方法 | 说明 |
| Socket accept(); | 侦听并接受到此套接字的连接，即获取客户端的连接 |
| void bind(SocketAddress endpoint); | 绑定到指定的本地地址（IP地址和端口号） |
| void bind(SocketAddress endpoint, int backlog); | 绑定到指定的本地地址并指定最大连接队列 |
| void close(); | 关闭此套接字 |
| void listen(); | 监听，让Socket侦听传入的连接尝试 |
| InetAddress getInetAddress(); | 返回此套接字连接到的远程IP地址/null |
| InetAddress getLocalAddress(); | 获取套接字绑定的本地地址/null |
| SocketAddress getLocalSocketAddress(); | 返回套接字绑定的本地端点的地址/null |
| SocketAddress getRemoteSocketAddress(); | 返回套接字连接的远程端点的地址/null |
| int getPort(); | 返回套接字连接到的远程端口，未连接为0 |
| int getLocalPort(); | 返回套接字绑定到的本地端口，未绑定为-1 |
| boolean isConnected(); | 如果将套接字成功连接到服务器，返回true |
| boolean isBound(); | 如果将套接字成功绑定到一个地址，返回true |
| boolean isClosed(); | 如果已经关闭了套接字，返回true |
| void setSoTimeout(int timeout); | 启用/禁用带有指定超时值的SO\_TIMEOUT |
| int getSoTimeout(); | 返回SO\_TIMEOUT的设置，0表示禁用 |
| void connect(SocketAddress endpoint); | 将此套接字连接到服务器 |
| void connect(SocketAddress endpoint, int timeout); | 将此套接字连接到服务器，并指定一个超时值 |
| InputStream getInputStream(); | 返回此套接字的输入流 |
| OutputStream getOutputStream(); | 返回此套接字的输出流 |

1. Client/Server程序：客户端和服务器端采用TCP协议进行通信，当客户端链接服务器端程序后，服务器程序向客户端程序发送客户端的IP地址和端口号。
2. UDP：用户数据报协议，一个无连接、发送独立数据包的协议，它不保证数据按顺序传送和正确到达。

数据报Socket又称为UDP套接字，无需建立、拆除连接，而是直接将信息打包传向指定的目的地，适于断续、非实时通信。

利用UDP通信的两个程序是平等的，没有主次之分，代码也可以完全一样。

1. DatagramSocket类
2. 构造方法

|  |  |
| --- | --- |
| 构造方法 | 说明 |
| DatagramSocket() | 构造一个数据报套接字，并将其绑定到本地主机上的任何可用端口 |
| DatagramSocket(DatagramSocketimpl impl) | 使用指定的DatagramSocketImpl创建一个未绑定的数据报套接字 |
| DatagramSocket(int port) | 构造一个数据报套接字，并将其绑定到本地主机上的指定端口 |
| DatagramSocket(int port, InetAddress laddr) | 构造一个数据报套接字，绑定到指定的本地地址 |
| DatagramSocket(SocketAddress bindaddr) | 构造一个数据报套接字，绑定到指定的本地套接字地址 |

1. 常用方法

|  |  |
| --- | --- |
| 构造方法 | 说明 |
| close() | 关闭此数据报包套接字 |
| send(DatagramPacket p) | 使用指定的DatagramSocketImpl创建一个未绑定的数据报套接字 |
| receive(DatagramPacket p) | 构造一个数据报套接字，并将其绑定到本地主机上的指定端口 |

1. DatagramPacket类
2. 构造方法

|  |  |
| --- | --- |
| 构造方法 | 说明 |
| DatagramPacket(byte[] buf, int length) | 构造一个数据报包来接收长度为length的数据包 |
| DatagramPacket(byte[] buf, int length, InetAddress address, int port) | 构造一个数据报包，用于将长度为length的数据包发送到指定主机上的指定端口号 |
| DatagramPacket(byte[] buf, int offset, int length) | 构造一个数据报包，用于发送长度为length的数据包，并在缓冲区中指定一个偏移量 |
| DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port) | 构造一个数据报包，用于发送长度偏移为指定主机上指定端口号的长度length的数据包 |
| DatagramPacket(byte[] buf, int offset, int length, SocketAddress address, int port) | 构造一个数据报包，用于发送长度偏移为指定主机上指定端口号的长度length的数据包 |
| DatagramPacket(byte[] buf, int length, SocketAddress address) | 构造一个数据报包，用于将长度为length的数据包发送到指定主机上的指定端口号 |

1. 常用方法

|  |  |
| --- | --- |
| 常用方法 | 说明 |
| InetAddress getAddress() | 返回发送/接收该数据报的机器的IP地址 |
| byte[] getData() | 返回数据缓冲区 |
| int getLength() | 返回要发送的数据的长度或接收的数据的长度 |
| int getOffset() | 返回要发送的数据的偏移量或接收到的数据的偏移量 |
| int getPort() | 返回发送/接收该数据报的远程主机的端口号 |
| SocketAddress getSocketAddress() | 获取此数据包发送到或来自的远程主机的SocketAddress（通常是IP地址+端口号） |
| void setAddress(InetAddress iaddr) | 设置将数据报发送到的机器的IP地址 |
| void setData(byte[] buf) | 设置此数据包的数据缓冲区 |
| void setData(byte[] buf, int offset, int length) | 设置此数据包的数据缓冲区 |
| void setLength(int length) | 设置此数据包的长度 |

1. UDP接收程序必须先启动运行，才能接收UDP发送程序发送的数据
2. UDP网络程序的编写步骤
3. 服务器端/客户端创建DatagramSocket实例对象；
4. 编写send()和receive()方法：

– DatagramSocket.send(DatagramPacket p);

– DatagramSocket.receive(DatagramPacketp);

1. DatagramSocket.close();