基本数据类型

列举

byte（字节型）、short（短整型）、int（整型）、long（长整型）、float（单精度浮点型）、double（双精度浮点型）、boolean（布尔型）、char（字符型）

对应包装类

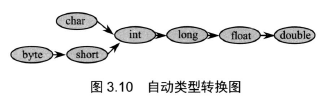
java.lang.Byte、java.lang.Short、java.lang.Integer、java.lang.Long、java.lang.Float、java.lang.Double、java.lang.Boolean、java.lang.Character

为什么不那么彻底？

性能的原因。

基本类型基于值，而对象类型则基于引用。

类型转换



long为什么转为float？<https://blog.csdn.net/writing_happy/article/details/78880606>

转换为字符串：+、String.valueof方法

整数默认int 小数默认double

数组初始化：(与C不同)

注意：不要同时使用静态初始化和动态初始化，也就是说，不要在进行数组初始化时，既指定数组的长度，也为每个数组元素分配初始值。

Java中把一个数组名赋值给另一个数组名：改变引用。

C/C++呢？没有这样的操作

foreach循环（遍历数组和集合更加简洁）但是不能改变原数组的值（丢掉了索引信息）

作用域：

Java类的成员变量：类变量，实例变量

局部变量

栈内存

存放：基本类型的变量、对象的引用变量（主要用来执行程序的）

堆可以动态地分配内存大小，生存期也不必事先告诉编译器，在运行时动态分配内存

数据会共享

堆内存

存放:由new创建的对象和数组（主要用来存放对象）

存在栈中的数据大小与生存必须是确定的

差异：

栈内存是线程私有的，堆内存是所有线程共有的

先进先出先进后出

**彻底理解Java中堆和栈的区别**：<https://blog.csdn.net/weixin_37618354/article/details/80239219>

变量生存周期：

1). **类变量**(static修饰的变量)：在程序加载时系统就为它在堆中开辟了内存，堆中的内存地址存放于栈以便高速访问。静态变量的生命周期—一直持续到整个“系统”关闭

2). **实例变量**：当你使用java关键字new的时候，系统在堆中开辟并不一定是连续的空间分配给变量(比如说类实例)，然后根据零散的堆内存地址，通过哈希算法换算为一长串数字以表征这个变量在堆中的“物理位置”。实例变量的生命周期–当实例变量的引用丢失后，将被GC(垃圾回收器)列如可回收“名单”中，但并不马上就释放堆中内存

3). **局部变量**：局部变量，由声明在某方法，或某代码段里(比如for循环)，执行到它的时候在堆中开辟内存，当局变量一旦脱离作用域，内存立即释放

常量池：存放基本类型常量和字符串常量（public static final）

静态域：存放静态成员(static定义的)

<https://www.iteye.com/blog/josh-persistence-1886949>

eg.数组初始化new与不new：

1、不同于String类，String由于实现了常量池 所以new 和不new 有区别:new的话，引用变量指向堆区。不new的话，引用变量指向常量池。  
2、对于数组的定义，初始化时用new与不用new 没区别，只是两种方式罢了，因为数组是引用数据类型，建立对象时，无论用不用new，数组实体都是放在堆内存中，引用变量放在栈内存。

访问域：



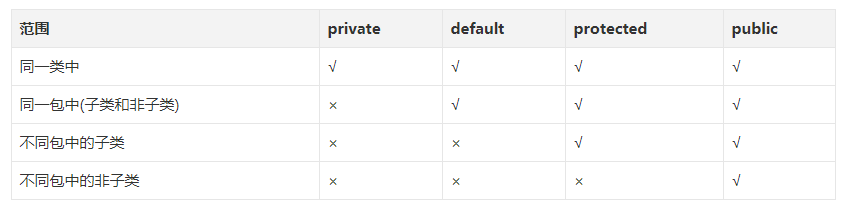
private:把成员变量隐藏在类的内部

default:相同包的其他类访问

protected:相同包以及不同包的子类：希望子类重写这个方法

public:所有都能访问

修饰符：



Static：区分成员变量、方法、内部类、初始化块属于类本身还是实例。

Static 静态，并非不能变

Static final 类似于全局常量

static修饰的方法可以使用类来直接调用，不能使用this，所以，不能访问没有static修饰的成员。

静态函数不能出现this或者super关键字。

当程序需要访问类变量的时候，尽量使用类作为主调，而非对象，这样可以避免产生歧义。

类，接口，抽象类：

接口：

Interface implement来实现

接口与类的区别：

接口无法被实例化，但是可以被实现。

接口没有构造方法。

接口中所有的方法必须是抽象方法。

接口不能包含成员变量，除了 static 和 final 变量。

接口不是被类继承了，而是要被类实现。

接口支持多继承。

接口特性：

1.接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 public abstract（只能是 public abstract，其他修饰符都会报错）。

2.接口中可以含有变量，但是接口中的变量会被隐式的指定为 public static final 变量（并且只能是 public，用 private 修饰会报编译错误）。

3.接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别：

1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。

2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 类型的。

3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。

4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

抽象类：

abstract

不能实例化（new），只能作为父类给子类继承，没有方法体

抽象类体现模板设计模式，抽象类作为多个子类的通用模板。

抽象方法：Abstract 关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

如果一个类包含抽象方法，那么该类必须是抽象类。

任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

构造方法，类方法（用 static 修饰的方法）不能声明为抽象方法。

抽象类是特殊的类，不能实例化，包含抽象方法

接口只有常量 未实现的方法

抽象类：有父子关系

接口：共有的行为

方法重载：类名相同，形参列表不同。

Java确定一个方法的三个要素：调用者（对象，类）、方法名、形参列表

方法重写（方法覆盖）发生在子父类之间，子类的同名方法覆盖类父类方法。

Collection(object)

├ List(接口)  
│├ LinkedList  
│├ ArrayList  
│└ Vector  
│　└ Stack  
└ Set(接口)

Map  
├ Hashtable  
├ HashMap  
└ WeakHashMap

List 有序,可重复

Set 无序 唯一

Map 集合类用于存储元素对（称作“键”和“值”）

遍历：迭代器，foreach

Iterator it = collection.iterator(); // 获得一个迭代子

while(it.hasNext())

{

Object obj = it.next(); // 得到下一个元素

}

泛型：

泛型类：public class pair<T>{} eg. ArrayList<string>

泛型方法：public static <T extends Compare> T min(T[] a) {} 类型变量在修饰符后，函数名前

类型变量的限定，见上

方法返回多参数：1.返回一个引用类型（对象，数组）

泛型擦除：

什么情况下擦除

为什么擦除

擦除带来的后果

通配符：

为什么引入？

就算容器中的类型之间存在继承关系，但是Plate和Plate两个容器之间是不存在继承关系的。

下界通配符：

<? super T>不影响往里面存储，但是读取出来的数据只能是Object类型。

下界通配符规定了元素最小的粒度，必须是T及其基类，那么我往里面存储T及其派生类都是可以的，因为它都可以隐式的转化为T类型。但是往外读就不好控制了，里面存储的都是T及其基类，无法转型为任何一种类型，只有Object基类才能装下。

编译期运行期 深入理解

内部类

为什么引入，解决什么问题

内部类方法可以访问该类定义所在作用域中的数据，包括被 private 修饰的私有数据

内部类可以对同一包中的其他类隐藏起来

内部类可以实现 java 单继承的缺陷

当我们想要定义一个回调函数却不想写大量代码的时候我们可以选择使用匿名内部类来实

<https://juejin.im/post/5a903ef96fb9a063435ef0c8>

最简单的用法示例

内部类类像是外部类的一个成员，成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）

当成员内部类拥有和外部类**同名的成员变量**或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：

外部类.this.成员变量

外部类.this.成员方法

外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，再通过指向这个对象的引用来访问

局部内部类和匿名内部类只能访问局部final变量

final修饰符

类：当用final修饰一个类时，表明这个类不能被继承。

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。

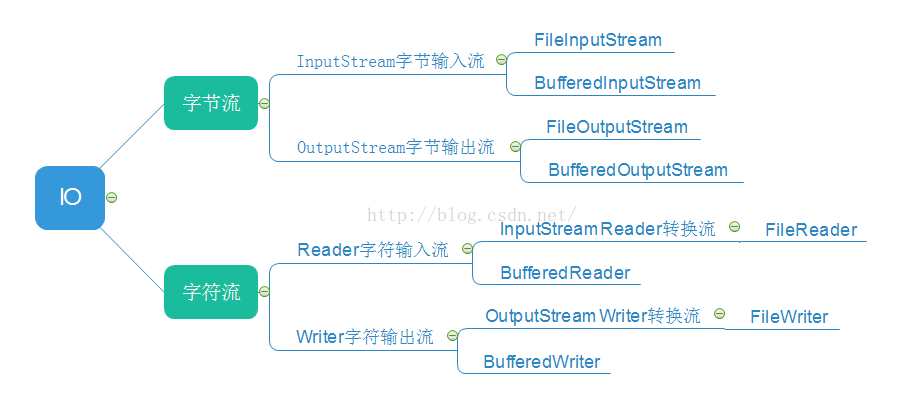
方法：

如果只有在想明确禁止 该方法在子类中被覆盖的情况下才将方法设置为final的。

　　注：类的private方法会隐式地被指定为final方法。

变量：对于一个final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。

JAVA IO



只要是处理纯文本数据，就优先考虑使用字符流。 除此之外都使用字节流。

Android架构简述

多线程：

start()方法的调用后并不是立即执行多线程代码，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。

从程序运行的结果可以发现，多线程程序是乱序执行。因此，只有乱序执行的代码才有必要设计为多线程。

实现Runnable接口比继承Thread类所具有的优势：

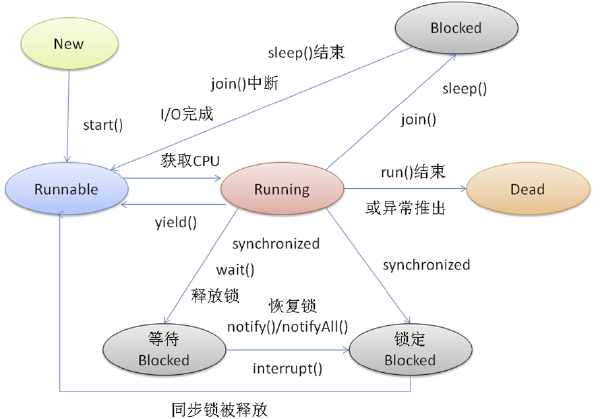
1）：适合多个相同的程序代码的线程去处理同一个资源

2）：可以避免java中的单继承的限制

3）：增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

4）：线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

Obj.wait()，与Obj.notify()必须要与synchronized(Obj)一起使用，也就是wait,与notify是针对已经获取了Obj锁进行操作，从语法角度来说就是Obj.wait(),Obj.notify必须在synchronized(Obj){...}语句块内。



同步方法：

synchronized关键字的作用域有二种：

1）是某个对象实例内，synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；

2）是某个类的范围，synchronized static aStaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用。

把synchronized当作函数修饰符时，示例代码如下：

Public synchronized void methodAAA()

{

//….

}

这也就是同步方法，那这时synchronized锁定的是哪个对象呢？它锁定的是调用这个同步方法对象。也就是说，当一个对象P1在不同的线程中执行这个同步方法时，它们之间会形成互斥，达到同步的效果。但是这个对象所属的Class所产生的另一对象P2却可以任意调用这个被加了synchronized关键字的方法。

<https://blog.csdn.net/Evankaka/article/details/44153709>

线程的优先级

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。

static int MAX\_PRIORITY

线程可以具有的最高优先级，取值为10。

static int MIN\_PRIORITY

线程可以具有的最低优先级，取值为1。

static int NORM\_PRIORITY

分配给线程的默认优先级，取值为5。

如果start方法依次调用run方法，为什么我们会选择去调用start方法？或者在java线程中调用start方法与run方法的区别在哪里？

调用start方法一个新线程将会被创建，并且在run方法中的代码将会在新线程上运行，当一个Java程序员开始学习线程的时候，他们首先会学着去继承Thread类，重载run方法或者实现Runnable接口，实现run方法，然后调用Thread实例的start方法。但是当他拥有一些经验之后，他通过查看API文档或者其他途径会发现start方法内部会调用run方法。

常用函数：

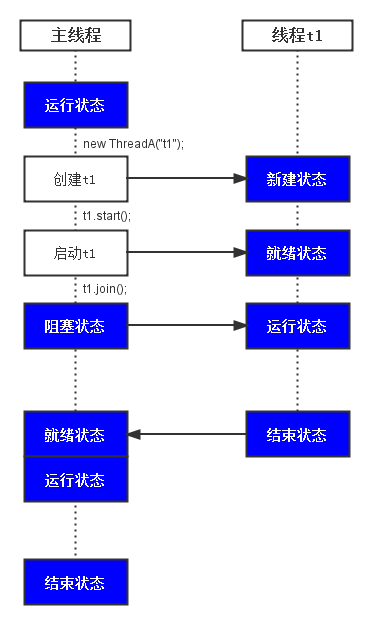
sleep(long millis)：在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）

1、线程睡眠是帮助所有线程获得运行机会的最好方法。

2、线程睡眠到期自动苏醒，并返回到可运行状态，不是运行状态。sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始执行。

3、sleep()是静态方法，只能控制当前正在运行的线程。

join():线程加入：让“主线程”等待“子线程”结束之后才能继续运行。



Yield():让当前运行[线程](http://www.java1995.cn/wiki/topic/6)回到可运行状态，以允许具有相同优先级的其他[线程](http://www.java1995.cn/wiki/topic/6)获得运行机会。

yield()从未导致[线程](http://www.java1995.cn/wiki/topic/6)转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致[线程](http://www.java1995.cn/wiki/topic/6)从运行状态转到可运行状态，但有可能没有效果。

当设计多[线程](http://www.java1995.cn/wiki/topic/6)应用程序的时候，一定不要依赖于[线程](http://www.java1995.cn/wiki/topic/6)的优先级。因为[线程](http://www.java1995.cn/wiki/topic/6)调度优先级操作是没有保障的，只能把[线程](http://www.java1995.cn/wiki/topic/6)优先级作用作为一种提高程序效率的方法，但是要保证程序不依赖这种操作。

interrupt():

不要以为它是中断某个线程！它只是线线程发送一个中断信号，让线程在无限等待时（如死锁时）能抛出抛出，从而结束线程，但是如果你吃掉了这个异常，那么这个线程还是不会中断的！

同步块synchronized(Obj)或者synchronized方法

同步块：public void method3(SomeObject so)

{

synchronized(so){ //…..}

}

Obj.wait()

Obj.notify()

Sleep():当在一个Synchronized块中调用Sleep()方法是，线程虽然休眠了，但是对象的机锁并木有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。

线程数据同步

Java异常



异常和错误：

①.Exception（异常）是应用程序中可能的可预测、可恢复问题。一般大多数异常表示中度到轻度的问题。

②.Error（错误）表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。

运行时异常

　RuntimeException类是Exception类的子类，它叫做运行时异常，Java中的所有运行时异常都会直接或者间接地继承自RuntimeException类。

  Java中凡是继承自Exception，而不继承自RuntimeException类的异常都是非运行时异常。

  一个try后面可以跟多个catch，但不管多少个，最多只会有一个catch块被执行。

对于非运行时异常（checked exception），必须要对其进行处理，否则无法通过编译。

　　处理方式有两种：

　　1.使用try..catch..finally进行捕获；

　　2.在产生异常的方法声明后面写上throws 某一个Exception类型，如throws Exception，将异常抛出到外面一层去。

对于运行时异常（runtime exception），可以对其进行处理，也可以不处理。推荐不对运行时异常进行处理。

checked exceptions: 通常是从一个可以恢复的程序中抛出来的，并且最好能够从这种异常中使用程序恢复。比如FileNotFoundException, ParseException等。检查了的异常发生在编译阶段，必须要使用try…catch（或者throws）否则编译不通过。

    unchecked exceptions: 通常是如果一切正常的话本不该发生的异常，但是的确发生了。发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的。比如ArrayIndexOutOfBoundException, ClassCastException等。从语言本身的角度讲，程序不该去catch这类异常，虽然能够从诸如RuntimeException这样的异常中catch并恢复，但是并不鼓励终端程序员这么做，因为完全没要必要。因为这类错误本身就是bug，应该被修复，出现此类错误时程序就应该立即停止执行。 因此，面对Errors和unchecked exceptions应该让程序自动终止执行，程序员不该做诸如try/catch这样的事情，而是应该查明原因，修改代码逻辑。

如果出现 RuntimeException，那么一定是程序员的错误。例如，可以通过检查数组下标和数组边界来避免数组越界访问异常。其他（IOException等等）checked异常一般是外部错误，例如试图从文件尾后读取数据等，这并不是程序本身的错误，而是在应用环境中出现的外部错误。