* 面向对象的特征
  1. 抽象：将一类对象的共同特征总结出来构造类，包括数据抽象和行为抽象
  2. 封装：把数据和操作数据的方法隐藏起来，对数据的访问只能通过已定义的外部接口
  3. 继承：从已有类得到继承信息创建新类的过程
  4. 多态：指允许不同子类型的对象对同一消息做出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情
     + 编译时多态：重载
     + 运行时多态：重写
  5. 底层怎样实现多态的
     + 定义：一个接口的不同实现方式
     + 前提：继承、重写和向上转型
     + 底层原理：动态绑定+方法表。父类方法和子类方法在方法表中的偏移量相同
* 面向对象6原则1法则
  1. 单一职责原则：一个类只做它该做的事情——**高内聚**
  2. 开闭原则：软件实体应当对扩展开放，对修改关闭
  3. 依赖倒转原则：尽可能使用抽象类型而不用具体类型
  4. 里氏替换原则：任何时候都可以用子类型替换掉父类型
  5. 接口隔离原则：接口要小而专，绝不能大而全
  6. 聚合复用原则：优先使用聚合关系复用代码
  7. 迪米特法则：最少知识原则，一个对象应当对其他对象有尽可能少的了解——**低耦合**
* 设计模式：开发人员在软件开发过程中面临的一般问题的解决方案
* 常用设计模式
  1. 工厂方法模式：在父类中提供一个创建对象的接口，允许子类决定实例化对象的类型
  2. 装饰器模式：通过将对象放入包含行为的特殊封装对象中来为原对象绑定新的行为
  3. 适配器模式：使接口不兼容的对象能够相互合作
  4. 享元模式：尽量使用内存中已有的对象，减少了创建对象的数量，从而减少内存占用和提高性能
* 单例类：将构造器私有，不允许外界通过构造器创建对象；通过公开的静态方法向外界返回类的唯一实例
  1. 饿汉式单例：

public class Singleton {

private Singleton(){}

private static Singleton instance = new Singleton();

public static Singleton getInstance(){

return instance;

}

}

* 1. 懒汉式单例：

public class Singleton {

private static Singleton instance = null;

private Singleton() {}

public static **synchronized** Singleton getInstance(){

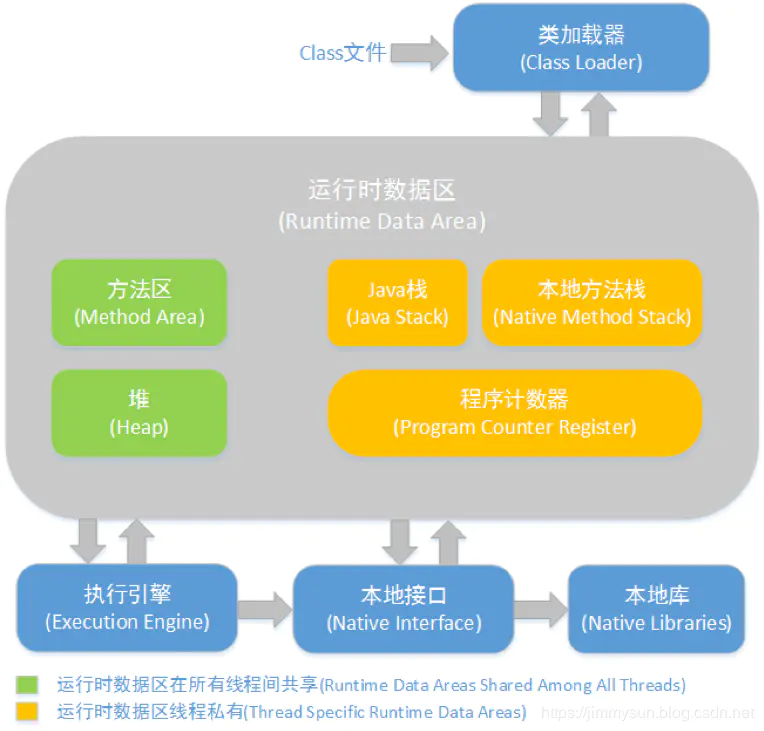
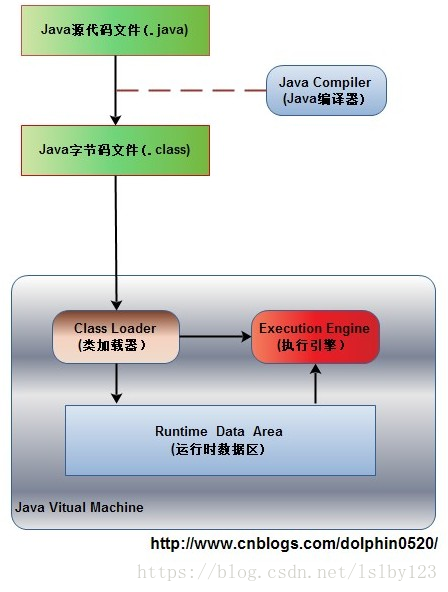
if (instance == null) instance ＝ new Singleton();

return instance;

}

}

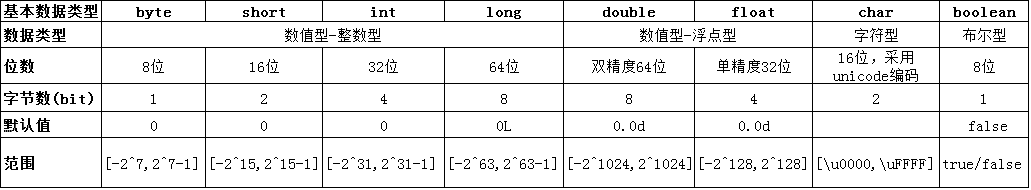
* JRE、JDK、JVM、JIT、JNI
  1. JRE：Java运行时环境，是运行Java引用所必需的
  2. JDK：Java开发工具
  3. JVM：Java虚拟机
  4. JIT：即时编译，当代码执行次数超过一定阈值时会将Java字节码转换为本地代码
  5. JNI：Java本地接口，通过使用 Java本地接口书写程序，可以确保代码在不同的平台上方便移植
* Java的内存分区



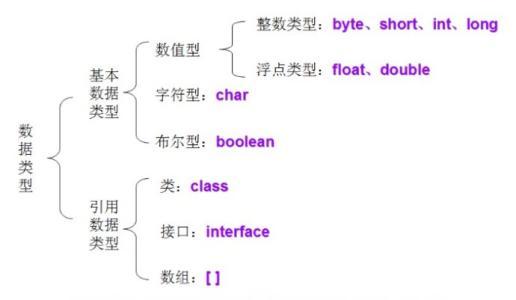
* 1. 栈存储的是方法内部的局部变量，若**声明为final则放入堆中**
  2. 堆存储的是new创建出的对象。
  3. 方法区存储的是已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码、运行时常量池；
  4. 程序计数器：JVM执行程序的流水线
* JVM加载class文件的原理
  1. 实现：用类加载器ClassLoader完成，包括根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）
  2. 步骤
     + 装载：将.class文件读入字节数组
     + 连接：验证、准备和解析
     + 初始化
* 内存泄漏：堆内存泄露，对象空间未被回收导致不可使用
* Java中是否存在内存泄露：不存在，有GC机制，但也有特殊情况
* Java GC
  1. 定义：垃圾回收机制，自动检测对象是否超过作用域从而达到自动回收内存的目的，从而有效的防止内存泄露
  2. 需要GC的内存区域：**主要集中于java堆和方法区中**
  3. GC的对象：没有存活的对象。判断方法
     + 引用计数：当对一个对象的引用数减为0时回收
     + 可达性分析：从GC Roots向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的
  4. GC Roots：是一些由堆外指向堆内的引用
     + 栈中引用对象
     + 方法区类静态属性实体引用对象
     + 方法区中常量引用对象
     + 本地方法栈中JNI引用的对象
  5. 触发时机
     + 手动调用System.gc
     + 系统自身决定
  6. 所做事：清理对象，整理内存
  7. 类型：Minor GC和Full GC
  8. 算法
     + 标记-清除算法：标记阶段，为每个对象更新标记位，检查对象是否死亡，清除阶段对死亡对象进行清除。容易造成内存碎片
     + 复制算法：将可用内存按容量划分为大小相等的两块，每次只用其中一块。当这一块用完了，就将还存活的对象复制到另外一块上面，然后把原始空间全部回收
     + 标记-整理算法：标记过程与标记-清除算法过程一样，但后面不是简单的清除，而是让所有存活的对象都向一端移动，然后直接清除掉其他对象
     + 分代收集算法主流JVM常用；根据对象生存周期，将堆分为新生代和老年代
       - 新生代：生存周期短，每次GC会大量死去，采用复制算法
       - 老年代：生存周期长，采用标记-xx算法
* Java引用
  1. 强引用：指创建一个对象并把这个对象赋给一个引用变量，强引用有引用变量指向时永远不会被垃圾回收，JVM宁愿抛出OutOfMemory错误也不会回收这种对象；如果想中断强引用和某个对象之间的关联，可以显示地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象
  2. 软引用：如果一个对象具有软引用，内存空间足够，垃圾回收器就不会回收它；

如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用

* 1. 弱引用：弱引用也是用来描述非必需对象的，当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象
  2. 虚引用：虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在java中用java.lang.ref.PhantomReference类表示。如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。要注意的是，虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动
* Java源文件中类的个数（非内部类）：可以有多个，但public只能有一个，且必须与文件名相同
* 基本数据类型和范围



* 1. 类型转换
     + 隐式类型转换：不能将高精度降低为低精度，如float f = 1.1；short s = 1， s = s + 1不可以，计算结果为int，不能赋给低精度short，但s**+=**1则可以
     + 强制转换：（目标类型）数值
  2. Char能否存储一个汉字：可以，char使用unicode编码存储字符，占**2**个字节
* Java数据类型有哪些？



* 基本数据类型和包装类的区别

|  |  |
| --- | --- |
| 基本数据类型 | 包装器类 |
| 内置类型 | 类类型 |
| 值传递 | 引用传递 |
| 存储在栈中 | 存储在堆中 |
| 初始值为常量 | 初始值为null |

* 为什么需要包装类？

Java是完全的面向对象的语言，包装器类可以让数据支持面向对象的特性——**装饰器模式的体现**

* 类和接口
  1. 类之间：单继承，但可以多层继承
  2. 类与接口：单实现或多实现
  3. 接口之间：单继承或多继承
* 缓存池——new Integer(字面值)和Integer.valueOf(字面值)的区别
  1. 前者每次都会创建一个新对象
  2. 后者会使用缓存池中的对象，多次调用会获得同一个对象的引用
  3. 基本类型对应的缓冲池如下：在使用这些基本类型对应的包装类型时，就可以直接使用缓冲池中的对象，**超出范围则会创建新对象**
     + Boolean： true，false
     + all byte values
     + short： -128 and 127
     + int ： -128 ~ 127
     + char ：\u0000 ~ \u007F
* Java参数传递：**值传递**
  1. 基本类型：传递数值
  2. 引用类型：传递对象地址
* 访问权限：public protected private，不加默认为包级可见



* 抽象类和接口
  1. 抽象类一般会包含抽象方法，抽象方法一定位于抽象类中；抽象类不能被实例化，需要继承抽象类才能实例化其子类；**可以继承抽象或具体类，也可以实现接口**
  2. 接口中的成员必须声明为public，且字段默认为 static final；**可以单继承或多继承接口**
  3. 比较：
     + **设计上，抽象类:IS-A，接口：LIKE-A**
     + **使用上，一个类只能继承一个抽象类，但可以实现多个接口**
     + **接口字段只能是static final，抽象类无限制**
     + **接口成员访问权限只能是public，抽象类成员可以有多种权限**
     + **接口只有抽象方法，抽象中抽象方法、非抽象方法都可**
* **抽象方法：只能用public或protected修饰，不能被static、native和sychronized修饰，不能有方法体**
  1. 抽象方法需要重写，static方法不能重写
  2. 抽象方法没有实现，native方法是由本地代码实现的
  3. 抽象方法没有实现，sychronized和实现细节有关
* **内部类访问权限：可以访问创建它的外部类的成员，包括私有成员**
* 静态嵌套类和内部类的不同：Static Nested Class是被声明静态的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化
* **内部匿名类是否可以继承其他类或实现其他接口？可以**
* **获取一个类的类型对象的方法**
  1. Class.forName(class name)：用于做类加载
  2. 对象.getClass()：用于获得对象的类型
  3. 类名.class：用于获得指定的类型，传参用
* **调用super.getClass()得到的类型？——子类**，因为getClass在父类中声明为final，子类继承而不覆盖，等于子类直接调用getClass；要获取父类，则需要getClass().getSuperClass()
* 如何通过反射获取函数
  1. 通过上述方法获取类型对象clazz
  2. 两种方法
     + 调用空参数的构造函数：Object obj = clazz.newInstance()
     + 调用带参数的构造函数：**先要获取指定参数列表的构造器**，然后通过该构造函数的对象的newInstance(实际参数)进行对象的初始化：

Constructor constructor = clazz.getConstructor(**String.class,int.class**);

Object obj = constructor.**newInstance("zhagnsan",30);**

//获取所有构造器。

Constructor[ ] constructors = **clazz.getConstructors**(); //只包含公共的

//constructors = clazz.getDeclaredConstructors(); //本类公有，包含私有的

for(Constructor con : constructors) {

System.out.println(con);

}

* 重写与重载的区别
  1. 重写override：存在于继承体系中，子类实现了一个与父类在方法声明上完全相同的一个方法，其必须满足：
     + **子类方法的可见性必须大于等于父类方法**
     + **子类方法的返回类型必须是父类方法返回类型或为其子类型**
     + **抛出异常可以减少或删除，一定不能抛出新的或者更广的异常**

使用@override注解，可让编译器帮忙检查是否满足上面两个限制条件

注意：**构造函数不能重写，只能重载**

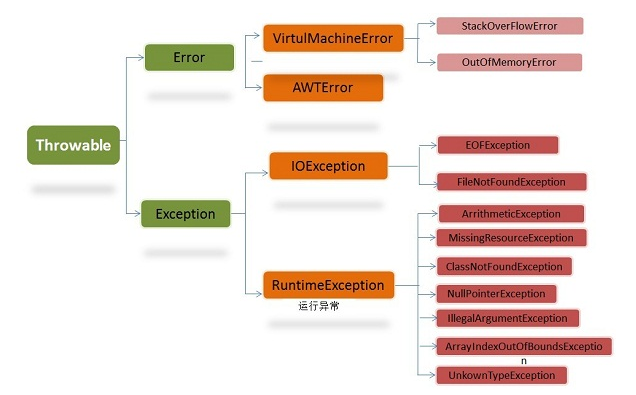
* 1. 重载overload：存在于同一个类中，指一个方法与已经存在的方法**名称上相同**，但是**参数类型、个数、顺序至少有一个不同**
* Object函数：equals和hashCode
  1. equal()相等的两个对象他们的hashCode()肯定相等，也就是用equal()对比是绝对可靠的。
  2. hashCode()相等的两个对象他们的equal()不一定相等，也就是hashCode()不是绝对可靠的。
  3. 每当需要对比的时候，首先用hashCode()去对比，如果hashCode()不一样，则表示这两个对象肯定不相等（也就是不必再用equal()去再对比了）,如果hashCode()相同，此时再对比他们的equal()，如果equal()也相同，则表示这两个对象是真的相同了，这样既能大大提高了效率也保证了对比的绝对正确性
* 语句、关键字、运算符
  1. final，finally和finalized区别
     + final：修饰类，不能被继承，其**成员方法隐式为final**；修饰方法，不能重写；修饰字段，必须只能赋一次值；修饰参数，方法内部不能修改值；修饰异常对象，catch中不能修改值；修饰局部变量，必须在访问前明确赋值，若为基本类型，不可修改值；若为引用类型，则不能被别的引用所指向，但可以修改内容
     + finally：异常处理，在catch后执行
     + finalized：Object类函数，定义了对象被GC前要做的事，相当于析构函数
  2. super：用于构造父类和访问父类函数
  3. switch语句：支持int及精度以下类型（byte，short，char），不支持long，**java 7 后支持string**
  4. Java有无goto关键字：保留字，目前版本中未使用
  5. &和&&的区别：逻辑与，后者有短路效应；前者可作为位运算符
  6. ==和equals：前者判断是否为一个对象，后者判断是否等价；重写equals后一般还要重写hashcode，从而正确使用Set，Map等数据结构
  7. 高效实现乘除2的幂：移位运算符<<,>>
  8. Final关键字修饰变量时，引用不能变还是对象不能变：引用不能指向其他对象，对象内容可以改变
  9. static作用：使声明的局部变量或成员在内存中只有一份拷贝；不能在static方法中访问非static成员
  10. 跳出多重嵌套循环的方法：外部声明标签A，内部break A；尽量避免这种写法
  11. throw和throws的区别
      + Throw：**抛出**一个异常对象，用在方法内
      + Throws：**声明**方法抛出的异常类，可以有多个
  12. 传统for循环、增强for循环、及迭代器的区别
      + 传统for循环：可以定义下表，通过下表操作元素
      + 增强for循环：对集合进行遍历，集合不能为空。只能获取集合元素，无法获取角标，不能对集合进行操作。增强for循环有一个局限性，必须要有被遍历的目标
      + 迭代器（Iterator）：除了遍历，还可以进行remove集合中元素的动作。如果是用ListIterator，还可以在遍历过程中进行增删改的动作
  13. assert作用：断言，调试一个boolean表达式，若不为true则系统将给出警告或退出
  14. **volatile关键字**：保证了并发的内存可见性和防止指令重排序（有序性）特征，但没有保证原子性；让变量每次在使用的时候，都从主存中取。而不是从各个线程的“工作内存”，也就是说，volatile变量对于每次使用，线程都能得到当前volatile变量的最新值。但是volatile变量并不保证并发的正确性。Volatile一般情况下不能代替sychronized，因为volatile不能保证操作的原子性，即使只是i++，实际上也是由多个原子操作组成；**volatile不会让线程阻塞，响应速度比synchronized高**
* String
  1. 不可继承，因为被**声明为final**
  2. 用final将String声明为不可变的好处
     + 可以缓存哈希值
     + 可以使用String常量池
     + 安全（网络连接参数；线程）
  3. 用相同的字面值初始化字符串时，二者指向同一内存，==结果为true；用new构造新实例时，两者为不同对象，==为false，内容判断用equals；调用**intern()**方法可以将自身引用的对象放在缓存池中，并返回对该常量的引用
  4. String, StringBuffer 和 StringBuilder区别

|  |  |  |  |
| --- | --- | --- | --- |
|  | String | StringBuilder | StringBuffer |
| 可变性 | 不可变 | 可变 | 可变 |
| 对equals和hashCode的覆盖 | 覆盖 |  | 未覆盖 |
| 线程安全 | 安全 | 不安全 | 安全 |
| 效率（速度） |  | 高 | 低 |

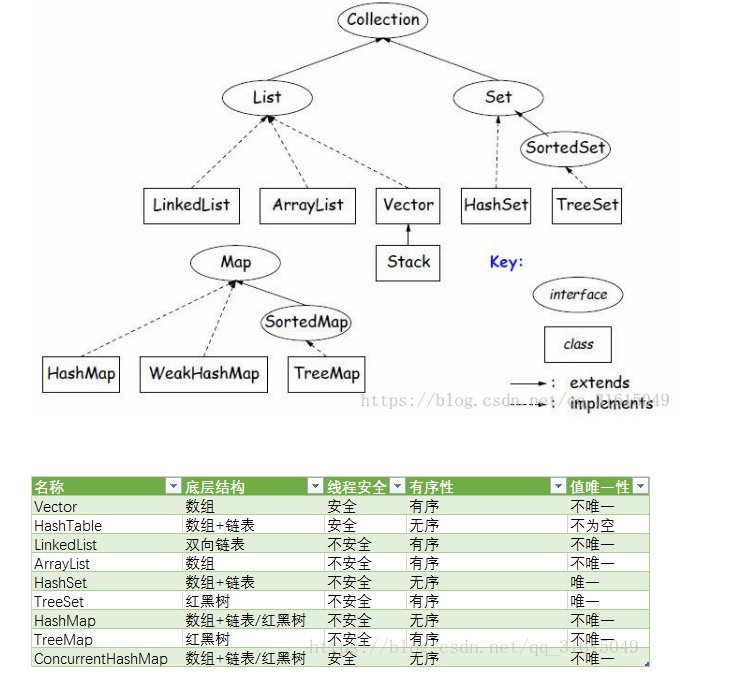
* 1. String s = null和String s =””的区别：前者声明了引用但未分配内存，后者创建了长度为0的空字符串，并分配了内存
  2. **new String(“xyz”)创建了几个对象？两个，堆中和常量池中，在运行时和类加载时**
  3. **只有使用字面值的方式创建的String对象之间使用“+”连接产生的新对象才会被加入常量池中——String str = “hello” + “world”；**

**对于所有包含new方式新建对象（包括null）的“+”连接表达式，它所产生的新对象都不会被加入字符串池中——String str = new String(“hello”) +＂ｗｏｒｌｄ＂**

* 1. **字符串编码转换：String s2 = new String(s1.getBytes("GB2312"), "ISO-8859-1");**
* Java和JS的比较
  1. 前者为静态语言，后者为动态语言
  2. 前者完全面向对象，后者基于对象
  3. 前者编译，后者解释
  4. 前者强类型变量，后者弱类型变量
  5. 代码格式不同
* Java的异常分类



* 1. Error: **系统级的错误和程序不必处理的异常**，是恢复不是不可能但很困难的情况下的一种**严重问题**，如内存溢出
  2. Exception：需要捕获进行处理的异常，是一种设计或实现问题
  3. 常见RuntimeException：ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException, IllegalArgumentException, ClassNotFoundException
  4. OutOfMemoryError属于哪种异常？——VirtualMachineError
* 编译时异常和运行时异常的区别
  1. 程序正确，但因为外在的环境条件不满足引发。Java编译器强制要求处理这类异常，如果不捕获这类异常，程序将不能被编译
  2. 运行时异常：运行时发生的逻辑错误，可以通过编译
* **Try中有return语句，则finally的代码是否会执行？——会执行，在返回前执行**
* Collection接口下有哪些子类



* ArrayList，LinkedList和Vector比较

|  |  |  |  |
| --- | --- | --- | --- |
|  | ArrayList | LinkedList | Vector |
| 底层实现 | 数组 | **双向**链表 | 数组 |
| 效率 | 查询快，增删慢 | 增删快，查询慢 |  |
| 线程安全 | 不安全 | 不安全 | 安全 |
| 扩容机制 | 起始容量为10，增长时\*1.5 |  | 指定增长系数，>0每次增加增长系数；<=0每次增加一倍 |

**使ArrayList和LinkedList安全的方法：使用Collections.synchronizedList(list)来“包装”该列表返回一个安全的列表对象**

* **Hashtable、Hashmap、ConcurrentHashmap的原理和区别**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Hashtable | Hashmap | ConcurrentHashmap |
| 底层实现 | 数组+链表 | 数组+链表/红黑树 | 分段的数组+链表/红黑树 |
| Key、value可否为null | 都不能 | 都可以 | 都不能 |
| 线程是否安全 | 安全 | 不安全 | 安全 |

**使HashMap安全的方法：使用Collections.synchronizeMap(map)包装返回一个安全的Map对象**

* **TreeMap和TreeSet排序时如何比较元素：键、元素类型必须实现Comparable接口**
* **Collections.sort()实现排序的原理：单参数：待排容器中类型实现了Comparable接口；两个参数：第二个参数是一个Comparable接口的实现类，内部重写compare(A,B)方法实现元素的比较，该方法返回一个整数，>0表示A>B，=0表示A=B，<0表示A<B**
* Thread类中的wait和sleep区别

|  |  |  |
| --- | --- | --- |
|  | wait | Sleep |
| 所属类 | Object | Thread |
| 是否释放同步锁 | 是 | 否 |
| 是否**必须**指定时间 | 否 | 是 |
| 使用区域 | 同步方法或代码块中 | 任何地方 |
| 是否必须捕获异常 | 否 | 是 |

* Thread类中的sleep和yield区别

|  |  |  |
| --- | --- | --- |
|  | sleep | yield |
| 优先级考虑 | 给低优先级线程机会 | 给相同或高优先级 |
| 状态转换 | 进入blocked状态 | 进入runnable状态 |
| 抛出异常 | 声明抛出InterruptedException | 未声明任何异常 |
| 可移植性 | 好 | 差 |

* Thread中的stop和suspend方法为什么不推荐使用？
  1. stop会释放线程获得的所有锁
  2. suspend不会释放任何锁，容易产生死锁
* **线程五种状态：New,Runnable,Running,Blocked,Dead**
  1. New：Tread t = new Tread(new Runnable（））
  2. Runnable：就绪，t.start（）
  3. Running：运行
  4. Blocked：阻塞，wait、sleep、同步锁被占用
  5. Dead：死亡，退出run方法
* **创建多线程方式：extends Thread；implements Runnable&new Thread(new Runnable(…))**
  1. 继承Thread类，重写run方法
  2. 实现runnable接口，重写run方法，用该类对象构造Thread类
  3. 区别：**前者简单，但不能继承别的类；后者可以实现继承，但比较复杂**
* **启动线程用start还是run？——start**
* **Java线程池：先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销——池化技术。**
  1. **实现方式**
     + **ThreadPoolExecutor**
     + **Executors**
       - **newSingleThreadExecutor**
       - **newFixedThreadPool**
       - **newScheduledThreadPool**
       - **newCachedThreadPool**
  2. **配置**
     + **CPU密集型：核心数+１**
     + **ＩＯ密集型：核心数＊２**
* 同步和异步
  1. 同步：阻塞式操作，如数据库事务执行中的排他锁，效率低，安全，保证可复现
  2. 异步：非阻塞操作，效率高，不安全
* Java线程同步机制:Sychronized关键字和Lock接口
  1. Synchronized关键字，对方法或代码块进行标记
  2. Lock类：Lock为接口，ReentrantLock为实现Lock的类
  3. 异同
     + Lock可以实现Synchronized所有功能
     + Synchronized自动释放锁，Lock要求程序员手动释放
     + Lock功能更强大
* **Java线程通讯方式**
  1. 利用锁生成Condition对象，然后调用await（）和signal（）或者signalAll（）
  2. 利用继承Object类中的wait（）和notify（）或者notifyAll（）唤醒方法
  3. 闭锁
  4. 栅栏
  5. 信号量
* 锁方法和锁代码块的比较：锁代码块粒度更细更灵活，线程等待时间短，效率高
* **对象锁和类锁：对实例和类上锁；可同时存在**

1. 对象锁：synchronized**普通成员函数或this**，对执行同步方法的**对象实例上锁**，执行后其他线程无法进入该对象的**其他对象锁同步方法**
2. 类锁：synchronized**静态成员函数或类名.class**，对执行同步方法的**类上锁**，执行后其他线程无法进入该类的**其他类锁同步方法**
3. **二者可以同时存在，互不影响**
4. 更优的对象锁写法：声明Object成员lock，再synchronized（lock）；若要对象之间互斥，则将lock声明为static

* Java序列化原理和意义：**实现serializable接口，通过对象流读写；解决对象流读写操作时可能引发的问题**
  + 原理：让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象
  + 意义：解决对象流读写操作时可能引发的问题
* Java流类型：**字节流，字符流，转换流**
  + 字节流：带Input和Output的流，只能用字节流构造
  + 字符流：带Reader和Writer的流，只能用字符流来构造
  + 转换流：转换字节和字符流的中间流，同时带xxput和xxder的流，用字节流对象构造，返回字符流对象进行下一步构造
* 利用jdbc进行查询的步骤（以查询为例）——加载jdbc驱动（Class.forName

(驱动名)）->创建连接对象->写SQL->用连接对象获取statement对象->执行sql得到结果集ResultSet对象->处理结果集（业务逻辑）->关闭各种资源（结果集，语句，连接）

* 1. **加载JDBC驱动程序**：导入驱动jar包，**Class.forName(“com.mysql.jdbc.Driver”);**
  2. **创建连接**：

Connection connection = null;

connection=DriverManager.getConnection(“jdbc:mysql://127.0.0.1:3306/jdbctest?useSSL=true&characterEncoding=utf-8&user=root&password=123”);

其中jdbctest是连接的数据库名，characterEncoding设置为GBK或者是UTF-8，这里设置为UTF-8，user是用户名，password是密码

* 1. **写sql**：String sql = “select \* from jdbc”;
  2. **获取statement对象**

PreparedStatement statement = null;

statement = connection.**prepareStatement**(sql);

// Statement st = connection.**createStatement**();

* + - 执行静态sql语句时，通常通过Statement实例实现。
    - 执行动态sql语句时，通常通过PreparedStatement实例实现。
    - 执行数据库存储过程，通常通过CallableStatement实例实现
  1. **执行sql 得到结果集**

ResultSet resultSet = null;

resultSet = statement.executeQuery(sql);

* + - executeQuery(String sql)：执行查询数据库的sql语句，返回一个结果集（ResultSet）对象。
    - int executeUpdate(String sql)：用于执行insert、update或Delete语句以及sql DDL语句。
    - execute(sql)：用于执行返回多个结果集、多个更新计数或二者组合的语句
  1. **处理结果集**

while(resultSet.next()){

System.out.println(resultSet.getInt(1));

System.out.println(resultSet.getString(2));

System.out.println(resultSet.getString(3));

}

* + - 执行更新返回的是本次操作影响的记录数
    - 执行查询返回的结果是一个ResultSet对象
  1. **关闭资源**
     + 关闭记录集ResultSet
     + 关闭语句Statement
     + 关闭连接对象Connection
* PreparedStatement和Statement的区别：**执行动态和静态语句；前者性能好，安全**
  1. 执行静态sql语句（不带参数）时，通常通过Statement实例实现；每次执行数据库都要**重新编译**语句
  2. 执行动态sql语句（带参数）时，通常通过PreparedStatement实例实现
  3. PreparedStatement优点
     + **效率高**：SQL命令会被数据库编译和解析,并放到命令缓冲区，供**重复使用**
     + 代码**可读性和可维护性好**
     + **安全**：可以防止SQL注入
* 使用jdbc提升性能的方法
  1. 提升读取数据的性能：ResultSet对象的**setFetchSize()**指定每次抓取的记录数
  2. 提升更新数据的性能：PreparedStatement**构建批处理——preparedStatement.addBatch()**，**将若干SQL语句置于一个批处理中执行——preparedStatement.executeBatch()**
* **进行数据库编程时，连接池有什么作用？——减少创建和释放连接造成的开销，池化技术**
* **什么是DAO模式？——DAO是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作，包含数据访问和数据封装两个方面**
* Jdbc如何进行事务处理：Connection提供了事务处理的方法
  1. 调用**setAutoCommit(false)**设置手动提交
  2. 事务完成后用**commit()**显式提交事务
  3. 在事务处理过程中发生异常则通过**rollback()**进行事务回滚
  4. Jdbc 3.0引入了**savepoint机制**，允许设置保存点并让事务回滚到指定点保存点
* 数据库的DDL和DML的用法和区别

1. DDL：数据定义语言，create, alter, drop, truncate, rename, comment
2. DML：数据管理语言，insert, update, delete, merge
3. 区别：DDL隐式提交，不可回滚；DML显式控制事务开始，提交，回滚

* **数据库ACID准则**
  + 原子性Atomic：事务不可分割，要么全部成功，要么全部失败
  + 一致性Consistency：事务百执行的结果必须是使数据库从一个一致度性状态变到另一个一致性状态。一致性状态是数据处于一种语义上的有意义且正确的状态。
  + 隔离性Isolation：事物之间相互隔离，不受影响
  + 持久性Durability：数据一旦提交不可改变