高频 Java 基础问题

- JVM、JRE、JDK 关系
- Java 是解释执行么？
- 采用字节码的好处
- JDK 1.8 之后有哪些新特性
- 构造器是否可以重写
- wait() 和 sleep 区别
- &和&&的区别
- Java 有哪些数据类型？
- this 关键字的用法
- super 关键字的用法
- 成员变量与局部变量的区别有哪些
- 动态代理是基于什么原理
- int 与 Integer 区别
- 面向对象四大特性
- 什么是多态机制？
- Java 语言是如何实现多态的？
- 重载与重写
- == 和 equals 的区别是什么
- 为什么要有 hashcode
- 面向对象的基本原则
- Exception
- Error
- JVM 如何处理异常？
- NoClassDefFoundError 和 ClassNotFoundException
- Java 常见异常有哪些？
- String
- StringBuilder
- StringBuffer
- HashMap 使用 String 作为 key 有什么好处
- 接口
- 抽象类
- 基本数据类型
- 对象引用类型
- 值传递和引用传递有什么区别？

Java 平台的理解

Java 是一种面向对象的语言，有两个明显特性：

- 跨平台能力：一次编写，到处运行（Write once，run anywhere）；
- 垃圾收集；

Java 通过字节码和 Java 虚拟机（JVM）这种跨平台的抽象，屏蔽了操作系统和硬件的细节，这也是实现「一次编译，到处执行」的基础。

Java 通过垃圾收集器（Garbage Collector）回收分配内存，大部分情况下，程序员不需要自己操心内存的分配和回收。

最常见的垃圾收集器，如 SerialGC、Parallel GC、 CMS、 G1 等，对于适用于什么样的工作负载最好也心里有数。

JVM、JRE、JDK 关系

`JVM Java Virtual Machine` 是 Java 虚拟机，Java 程序需要运行在虚拟机上，不同的平台有自己的虚拟机，因此 Java 语言可以实现跨平台。

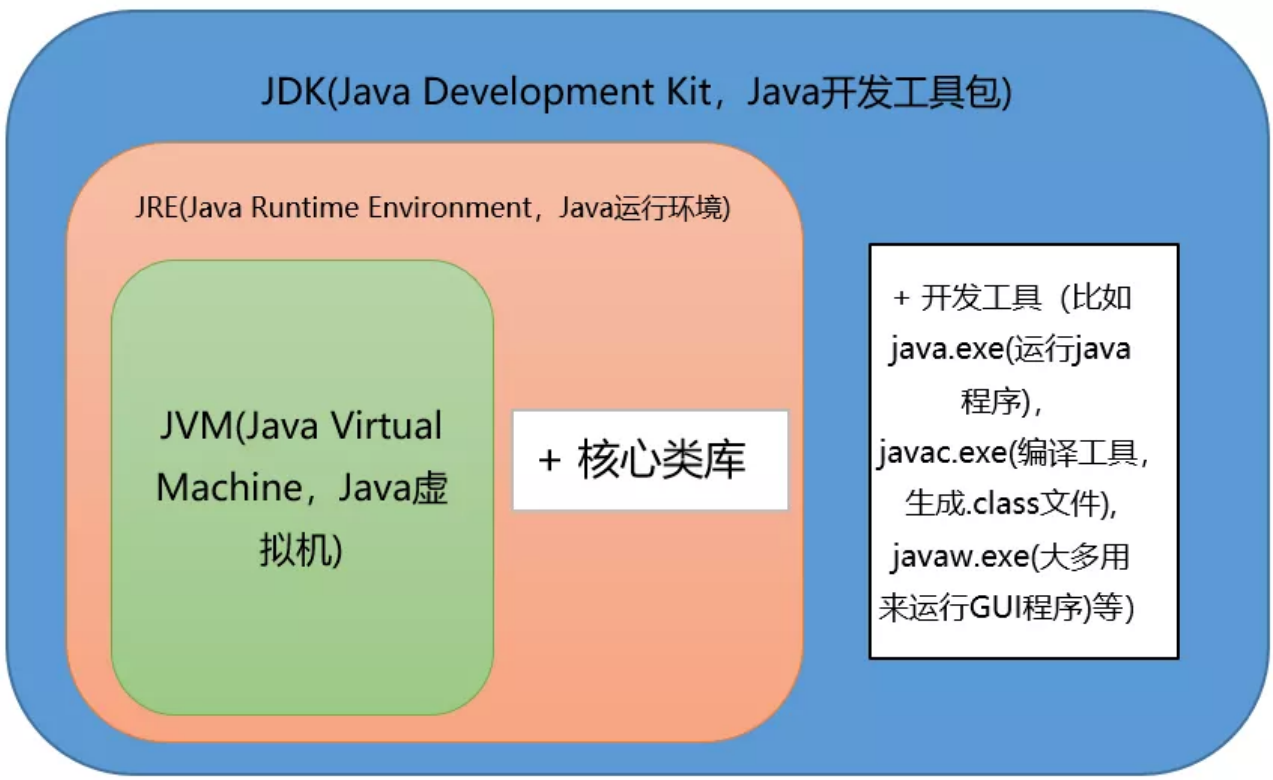
`JRE Java Runtime Environment`包括 Java 虚拟机和 Java 程序所需的核心类库等。

核心类库主要是 `java.lang` 包：包含了运行 Java 程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处理、线程、异常处理类等，系统缺省加载这个包

如果想要运行一个开发好的 Java 程序，计算机中只需要安装 JRE 即可。

`JDK Java Development Kit`是提供给 Java 开发人员使用的，其中包含了 Java 的开发工具，也包括了 JRE。

所以安装了 JDK，就无需再单独安装 JRE 了。其中的开发工具：编译工具(javac.exe)，打包工具(jar.exe) 等。



Java 是解释执行么?

这个说法不太准确。

我们开发的 Java 的源代码，首先通过 Javac 编译成为字节码（bytecode），在运行时，通过 Java 虚拟机（JVM）内嵌的解释器将字节码转换成为最终的机器码。

但是常见的 JVM，比如我们大多数情况使用的 Oracle JDK 提供的 Hotspot JVM，都提供了 `JIT（Just-In-Time）`编译器。

也就是通常说的动态编译器，JIT 能够在运行时将热点代码编译成机器码，这种情况下部分热点代码就属于编译执行，而不是解释执行了。

采用字节码的好处

字节码：Java 源代码经过虚拟机编译器编译后产生的文件（即扩展为.class 的文件），它不面向任何特定的处理器，只面向虚拟机。

采用字节码的好处：

众所周知，我们通常把 Java 分为编译期和运行时。这里说的 Java 的编译和 C/C++ 是有着不同的意义的，Javac 的编译，编译 Java 源码生成“.class”文件里面实际是字节码，而不是可以直接执行的机器码。Java 通过字节码和 Java 虚拟机（JVM）这种跨平台的抽象，屏蔽了操作系统和硬件的细节，这也是实现“一次编译，到处执行”的基础。

基础语法

JDK 1.8 之后有哪些新特性

接口默认方法：Java8 允许我们给接口添加一个非抽象的方法实现，只需要使用 default 关键字即可。

Lambda 表达式和函数式接口：Lambda 表达式本质上是一段匿名内部类，也可以是一段可以传递的代码。

Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中），使用 Lambda 表达式使代码更加简洁，但是也不要滥用，否则会有可读性等问题，《EffectiveJava》作者 JoshBloch 建议使用 Lambda 表达式最好不要超过 3 行。

StreamAPI：用函数式编程方式在集合类上进行复杂操作的工具，配合 Lambda 表达式可以方便的对集合进行处理。

Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。

使用 StreamAPI 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 StreamAPI 来并行执行操作。

简而言之，StreamAPI 提供了一种高效且易于使用的处理数据的方式。

方法引用：方法引用提供了非常有用的语法，可以直接引用已有 Java 类或对象（实例）的方法或构造器。

与 lambda 联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

日期时间 **API**：Java8 引入了新的日期时间 API 改进了日期时间的管理。

Optional 类：著名的 NullPointerException 是引起系统失败最常见的原因。

很久以前 GoogleGuava 项目引入了 Optional 作为解决空指针异常的一种方式，不赞成代码被 null 检查的代码污染，期望程序员写整洁的代码。

受 GoogleGuava 的鼓励，Optional 现在是 Java8 库的一部分。

新工具：新的编译工具，如：Nashorn 引擎 jjs、类依赖分析器 jdeps。

构造器是否可以重写

Constructor 不能被 override（重写），但是可以 overload（重载），所以你可以看到一个类中有多个构造函数的情况。

wait() 和 sleep 区别

来源不同：sleep()来自 Thread 类，wait()来自 Object 类。

对于同步锁的影响不同：sleep()不会该表同步锁的行为，如果当前线程持有同步锁，那么 sleep 是不会让线程释放同步锁的。

wait()会释放同步锁，让其他线程进入 synchronized 代码块执行。

使用范围不同：sleep()可以在任何地方使用。wait()只能在同步控制方法或者同步控制块里面使用，否则会抛 IllegalMonitorStateException。

恢复方式不同：两者会暂停当前线程，但是在恢复上不太一样。sleep()在时间到了之后会重新恢复；

wait()则需要其他线程调用同一对象的 notify()/nofityAll()才能重新恢复。

&和&&的区别

&运算符有两种用法：

1. 按位与；
2. 逻辑与。

&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是 true 整个表达式的值才是 true。

&&之所以称为短路运算，是因为如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。

注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

Java 有哪些数据类型?

Java 语言是强类型语言，对于每一种数据都定义了明确的具体的数据类型，在内存中分配了不同大小的内存空间。

分类

- 基本数据类型
  - 整数类型(byte,short,int,long)
  - 浮点类型(float,double)
  - 数值型
  - 字符型(char)
  - 布尔型(boolean)
- 引用数据类型
  - 类(class)
  - 接口(interface)
  - 数组([])

类型	类型名称	关键字	占用内存	取值范围	作为成员变量的默认值
整形	字节型	byte	1 字节	-128(-2^7) ~ 127(2^7-1)	0
	短整型	short	2 字节	-32,768(-2^15) ~ 32,767(2^15-1)	0
	整型	int	4 字节	-2,147,483,648(-2^31) ~ 2,147,483,647(2^31-1)	0
	长整型	long	8 字节	-9,223,372,036,854,775,808(-2^63) ~ 9,223,372,036,854,775,807(2^63-1)	0L
浮点型	单精度浮点型	float	4 字节	-3.403E38 ~ 3.403E38	0.0F
	双精度浮点型	double	8 字节	-1.798E308 ~ 1.798E308	0.0D
字符型	字符型	char	2 字节	表示一个字符，如('a','A','家')	'\u0000'
布尔型	布尔型	boolean	1 字节	只有两个值，true 或 false	false

this 关键字的用法

this 是自身的一个对象，代表对象本身，可以理解为：指向对象本身的一个指针。

this 的用法在 java 中大体可以分为 3 种：

1. 普通的直接引用，this 相当于是指向当前对象本身。
2. 形参与成员名字重名，用 this 来区分：

```
public Person(String name, int age) {
    this.name = name;
    this.age = age;
}
```

1. 引用本类的构造函数

```
class Person{
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }
    public Person(String name, int age) {
        this(name);
        this.age = age;
    }
}
```

super 关键字的用法

super 可以理解为是指向自己超（父）类对象的一个指针，而这个超类指的是离自己最近的一个父类。

super 也有三种用法：

1. 普通的直接引用：与 this 类似，super 相当于是指向当前对象的父类的引用，这样就可以用 super.xxx 来引用父类的成员。
2. 子类中的成员变量或方法与父类中的成员变量或方法同名时，用 super 进行区分

```
class Person{
    protected String name;

    public Person(String name) {
        this.name = name;
    }

}

class Student extends Person{
    private String name;

    public Student(String name, String name1) {
```

```
        super(name);
        this.name = name1;
    }

    public void getInfo(){
        System.out.println(this.name);      //Child
        System.out.println(super.name);     //Father
    }

}

public class Test {
    public static void main(String[] args) {
       Student s1 = new Student("Father","Child");
       s1.getInfo();

    }
}
```

3. 引用父类构造函数；

成员变量与局部变量的区别有哪些

变量：在程序执行的过程中，在某个范围内其值可以发生改变的量。从本质上讲，变量其实是内存中的一小块区域。

成员变量：方法外部，类内部定义的变量。

局部变量：类的方法中的变量。

区别如下：

作用域

成员变量：针对整个类有效。局部变量：只在某个范围内有效。(一般指的就是方法,语句体内)

存储位置

成员变量：随着对象的创建而存在，随着对象的消失而消失，存储在堆内存中。

局部变量：在方法被调用，或者语句被执行的时候存在，存储在栈内存中。当方法调用完，或者语句结束后，就自动释放。

生命周期

成员变量：随着对象的创建而存在，随着对象的消失而消失 局部变量：当方法调用完，或者语句结束后，就自动释放。

初始值

成员变量：有默认初始值。

局部变量：没有默认初始值，使用前必须赋值。

动态代理是基于什么原理

基于反射实现

反射机制是 Java 语言提供的一种基础功能，赋予程序在运行时自省（introspect，官方用语）的能力。通过反射我们可以直接操作类或者对象，比如获取某个对象的类定义，获取类声明的属性和方法，调用方法或者构造对象，甚至可以运行时修改类定义。

AOP 通过（动态）代理机制可以让开发者从这些繁琐事项中抽身出来，大幅度提高了代码的抽象程度和复用度。

包装 **RPC** 调用：通过代理可以让调用者与实现者之间解耦。比如进行 RPC 调用，框架内部的寻址、序列化、反序列化等，对于调用者往往是没有太大意义的，通过代理，可以提供更加友善的界面。

int 与 Integer 区别

Java 是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

- 原始类型: boolean，char，byte，short，int，long，float，double。
- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double。

int 是我们常说的整形数字，是 Java 的 8 个原始数据类型（Primitive Types，boolean、byte 、short、char、int、float、double、long）之一。Java 语言虽然号称一切都是对象，但原始数据类型是例外。

Integer 是 int 对应的包装类，它有一个 int 类型的字段存储数据，并且提供了基本操作，比如数学运算、int 和字符串之间转换等。在 Java 5 中，引入了自动装箱和自动拆箱功能（boxing/unboxing），Java 可以根据上下文，自动进行转换，极大地简化了相关编程。

> *Integer a= 127 与 Integer b = 127 相等吗*

对于对象引用类型：比较的是对象的内存地址。对于基本数据类型：比较的是值。

大部分数据操作都是集中在有限的、较小的数值范围，因而，在 Java 5 中新增了静态工厂方法 valueOf，在调用它的时候会利用一个缓存机制，带来了明显的性能改进。按照 Javadoc，这个值默认缓存是 -128 到 127 之间。

如果整型字面量的值在-128 到 127 之间，那么自动装箱时不会 new 新的 Integer 对象，而是直接引用常量池中的 Integer 对象，超过范围 a1==b1 的结果是 false。

```
public static void main(String[] args) {
    Integer a = new Integer(3);
    Integer b = 3;  // 将3自动装箱成Integer类型
    int c = 3;
    System.out.println(a == b); // false 两个引用没有引用同一对象
    System.out.println(a == c); // true a自动拆箱成int类型再和c比较
    System.out.println(b == c); // true

    Integer a1 = 128;
    Integer b1 = 128;
    System.out.println(a1 == b1); // false

    Integer a2 = 127;
    Integer b2 = 127;
    System.out.println(a2 == b2); // true
}
```

面向对象

> 面向对象与面向过程的区别是什么？

面向过程

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护、易复用、易扩展

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点：性能比面向过程低

面向过程是具体化的，流程化的，解决一个问题，你需要一步一步的分析，一步一步的实现。

面向对象是模型化的，你只需抽象出一个类，这是一个封闭的盒子，在这里你拥有数据也拥有解决问题的方法。需要什么功能直接使用就可以了，不必去一步一步的实现，至于这个功能是如何实现的，管我们什么事？我们会用就可以了。

面向对象的底层其实还是面向过程，把面向过程抽象成类，然后封装，方便我们使用的就是面向对象了。

面向对象编程因为其具有丰富的特性（封装、抽象、继承、多态），可以实现很多复杂的设计思路，是很多设计原则、设计模式等编码实现的基础。

面向对象四大特性

抽象

抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

另外，抽象是一个宽泛的设计思想，开发者能不能设计好代码，抽象能力也至关重要。

很多设计原则都体现了抽象这种设计思想，比如基于接口而非实现编程、开闭原则（对扩展开放、对修改关闭）、代码解耦（降低代码的耦合性）等。

在面对复杂系统的时候，人脑能承受的信息复杂程度是有限的，所以我们必须忽略掉一些非关键性的实现细节。

封装

把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。

通过封装，只需要暴露必要的方法给调用者，调用者不必了解背后的业务细节，用错的概率就减少。

继承

使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。

通过使用继承我们能够非常方便地复用以前的代码，需要注意的是，过度使用继承，层级深就会导致代码可读性和可维护性变差。

关于继承如下 3 点请记住：

1. 子类拥有父类非 private 的属性和方法。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。（以后介绍）。

多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定。

即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在 Java 中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

多态也是很多设计模式、设计原则、编程技巧的代码实现基础，比如策略模式、基于接口而非实现编程、依赖倒置原则、里式替换原则、利用多态去掉冗长的 if-else 语句等等。

什么是多态机制?

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

多态分为编译时多态和运行时多态。

其中编辑时多态是静态的，主要是指方法的重载，它是根据参数列表的不同来区分不同的函数，通过编辑之后会变成两个不同的函数，在运行时谈不上多态。

而运行时多态是动态的，它是通过动态绑定来实现的，也就是我们所说的多态性。

Java 语言是如何实现多态的?

Java 实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。

重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。

只有满足了上述三个条件，我们才能够在同一个继承结构中使用统一的逻辑实现代码处理不同的对象，从而达到执行不同的行为。

重载与重写

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。

重载：发生在同一个类中，方法名相同参数列表不同（参数类型不同、个数不同、顺序不同），与方法返回值和访问修饰符无关，即重载的方法不能根据返回类型进行区分。

重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛出的异常小于等于父类，访问修饰符大于等于父类（里氏代换原则）；如果父类方法访问修饰符为 private 则子类中就不是重写。

== 和 equals 的区别是什么

== : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象。(基本数据类型 == 比较的是值，引用数据类型 == 比较的是内存地址)。

equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。
- 类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

> 为什么重写 *equals* 时必须重写 *hashCode* 方法？

如果两个对象相等，则 hashcode 一定也是相同的

两个对象相等，对两个对象分别调用 equals 方法都返回 true

两个对象有相同的 hashcode 值，它们也不一定是相等的.

因此，**equals** 方法被覆盖过，则 **hashCode** 方法也必须被覆盖

为什么要有 hashcode

我们以**“HashSet** 如何检查重复**”**为例子来说明为什么要有 **hashCode**：

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的 hashcode，HashSet 会假设对象没有重复出现。

但是如果发现有相同 hashcode 值的对象，这时会调用 equals()方法来检查 hashcode 相等的对象是否真的相同。

如果两者相同，HashSet 就不会让其加入操作成功。

如果不同的话，就会重新散列到其他位置。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

面向对象的基本原则

这是面向对象编程的一种设计原则，对于每一种设计原则，我们需要掌握它的设计初衷，能解决哪些编程问题，有哪些应用场景。

- 单一职责原则 SRP(Single Responsibility Principle) 类的功能要单一，不能包罗万象，跟杂货铺似的。
- 开放封闭原则 OCP(Open－Close Principle) 一个模块对于拓展是开放的，对于修改是封闭的，想要增加功能热烈欢迎，想要修改，哼，一万个不乐意。

- 里式替换原则 LSP(the Liskov Substitution Principle LSP) 子类可以替换父类出现在父类能够出现的任何地方。比如你能代表你爸去你姥姥家干活。哈哈~~（其实多态就是一种这个原则的一种实现）。
- 接口分离原则 ISP(the Interface Segregation Principle ISP) 设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。就比如一个手机拥有打电话，看视频，玩游戏等功能，把这几个功能拆分成不同的接口，比在一个接口里要好的多。
- 依赖倒置原则 DIP(the Dependency Inversion Principle DIP) ：高层模块（high-level modules）不要依赖低层模块（low-level）。高层模块和低层模块应该通过抽象（abstractions）来互相依赖。除此之外，抽象（abstractions）不要依赖具体实现细节（details），具体实现细节（details）依赖抽象（abstractions）。
    - 抽象不应该依赖于具体实现，具体实现应该依赖于抽象。就是你出国要说你是中国人，而不能说你是哪个村子的。
    - 比如说中国人是抽象的，下面有具体的 xx 省，xx 市，xx 县。你要依赖的抽象是中国人，而不是你是 xx 村的。
    - 所谓高层模块和低层模块的划分，简单来说就是，在调用链上，调用者属于高层，被调用者属于低层。
    - Tomcat 就是高层模块，我们编写的 Web 应用程序代码就是低层模块。Tomcat 和应用程序代码之间并没有直接的依赖关系，两者都依赖同一个「抽象」，也就是 Servlet 规范。
    - Servlet 规范不依赖具体的 Tomcat 容器和应用程序的实现细节，而 Tomcat 容器和应用程序依赖 Servlet 规范。

接口隔离与单一职责有什么区别？

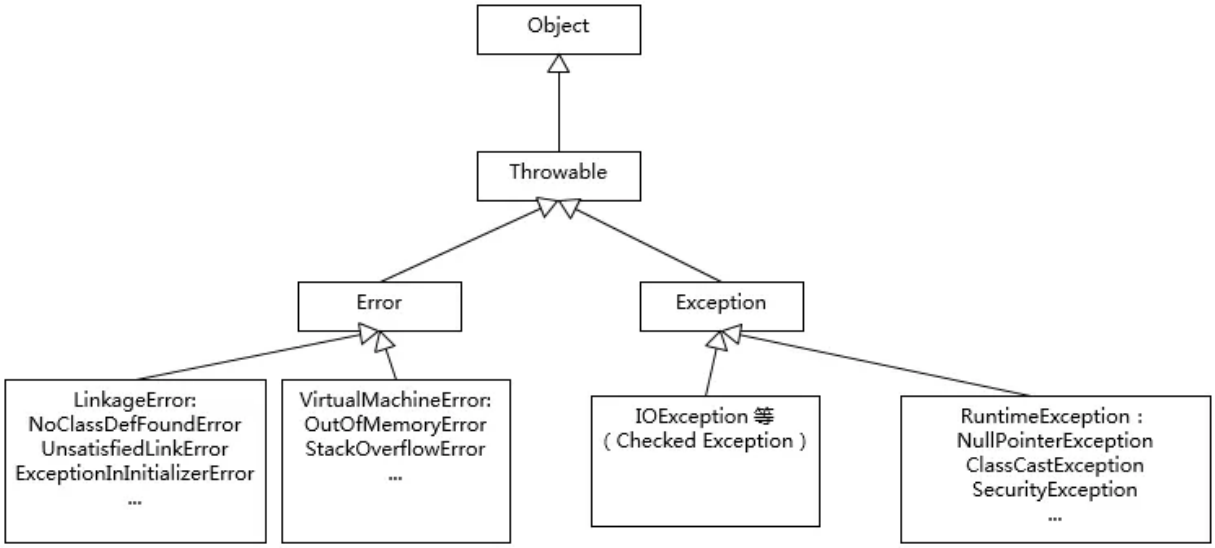
单一职责侧重点是模块、类、接口的设计思想。

接口隔离原则侧重于接口设计，提供了一种判断接口职责是否单一的标准。

Exception 与 Error 区别?

Exception 和 Error 都是继承了 Throwable 类，在 Java 中只有 Throwable 类型的实例才可以被抛出（throw）或者捕获（catch），它是异常处理机制的基本组成类型。

Exception 和 Error 体现了 Java 平台设计者对不同异常情况的分类。



异常使用规范：

- 尽量不要捕获类似 Exception 这样的通用异常，而是应该捕获特定异常
- 不要生吞（swallow）异常。这是异常处理中要特别注意的事情，因为很可能会导致非常难以诊断的诡异情况。

Exception

Exception 是程序正常运行中，可以预料的意外情况，可能并且应该被捕获，进行相应处理。

就好比开车去洗桑拿，前方道路施工，禁止通行。但是我们换条路就可以解决。

Exception 又分为可检查（checked）异常和不检查（unchecked）异常，可检查异常在源代码里必须显式地进行捕获处理，这是编译期检查的一部分。

不检查异常就是所谓的运行时异常，类似 NullPointerException、ArrayIndexOutOfBoundsException 之类，通常是可以编码避免的逻辑错误，具体根据需要来判断是否需要捕获，并不会在编译期强制要求。

Checked Exception 的假设是我们捕获了异常，然后恢复程序。但是，其实我们大多数情况下，根本就不可能恢复。

Checked Exception 的使用，已经大大偏离了最初的设计目的。Checked Exception 不兼容 functional 编程，如果你写过 Lambda/Stream 代码，相信深有体会。

Error

此类错误一般表示代码运行时 JVM 出现问题。通常有 Virtual MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）等。

比如 OutOfMemoryError：内存不足错误；StackOverflowError：栈溢出错误。此类错误发生时，JVM 将终止线程。

绝大多数导致程序不可恢复，这些错误是不受检异常，非代码性错误。因此，当此类错误发生时，应用程序不应该去处理此类错误。按照 Java 惯例，我们是不应该实现任何新的 Error 子类的！

比如开车去洗桑拿，老王出车祸了。无法洗了，只能去医院。

JVM 如何处理异常?

在一个方法中如果发生异常，这个方法会创建一个异常对象，并转交给 JVM，该异常对象包含异常名称，异常描述以及异常发生时应用程序的状态。

创建异常对象并转交给 JVM 的过程称为抛出异常。可能有一系列的方法调用，最终才进入抛出异常的方法，这一系列方法调用的有序列表叫做调用栈。

JVM 会顺着调用栈去查找看是否有可以处理异常的代码，如果有，则调用异常处理代码。

当 JVM 发现可以处理异常的代码时，会把发生的异常传递给它。如果 JVM 没有找到可以处理该异常的代码块，JVM 就会将该异常转交给默认的异常处理器（默认处理器为 JVM 的一部分），默认异常处理器打印出异常信息并终止应用程序。

NoClassDefFoundError 和 ClassNotFoundException

NoClassDefFoundError 是一个 Error 类型的异常，是由 JVM 引起的，不应该尝试捕获这个异常。

引起该异常的原因是 JVM 或 ClassLoader 尝试加载某类时在内存中找不到该类的定义，该动作发生在运行期间，即编译时该类存在，但是在运行时却找不到了，可能是变异后被删除了等原因导致；

ClassNotFoundException 是一个受查异常，需要显式地使用 try-catch 对其进行捕获和处理，或在方法签名中用 throws 关键字进行声明。

当使用 `Class.forName, ClassLoader.loadClass` 或 `ClassLoader.findSystemClass` 动态加载类到内存的时候，通过传入的类路径参数没有找到该类，就会抛出该异常；

另一种抛出该异常的可能原因是某个类已经由一个类加载器加载至内存中，另一个加载器又尝试去加载它。

Java 常见异常有哪些?

`java.lang.IllegalAccessError`：违法访问错误。当一个应用试图访问、修改某个类的域（Field）或者调用其方法，但是又违反域或方法的可见性声明，则抛出该异常。

`java.lang.InstantiationError`：实例化错误。当一个应用试图通过 Java 的 new 操作符构造一个抽象类或者接口时抛出该异常.

`java.lang.OutOfMemoryError`：内存不足错误。当可用内存不足以让 Java 虚拟机分配给一个对象时抛出该错误。

`java.lang.StackOverflowError`：堆栈溢出错误。当一个应用递归调用的层次太深而导致堆栈溢出或者陷入死循环时抛出该错误。

`java.lang.ClassCastException`：类造型异常。假设有类 A 和 B（A 不是 B 的父类或子类），O 是 A 的实例，那么当强制将 O 构造为类 B 的实例时抛出该异常。该异常经常被称为强制类型转换异常。

`java.lang.ClassNotFoundException`：找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历 CLASSPAH 之后找不到对应名称的 class 文件时，抛出该异常。

`java.lang.ArithmeticException`：算术条件异常。譬如：整数除零等。

`java.lang.ArrayIndexOutOfBoundsException`：数组索引越界异常。当对数组的索引值为负数或大于等于数组大小时抛出。

final、finally、finalize 区别?

除了名字相似，他们毫无关系！！！

- final 可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。
- finally 一般作用在 try-catch 代码块中，在处理异常的时候，通常我们将一定要执行的代码方法 finally 代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。
- finalize 是一个方法，属于 Object 类的一个方法，而 Object 类是所有类的父类，Java 中允许使用 finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

> “
>
> *final* 有什么用？

用于修饰类、属性和方法；

- 被 final 修饰的类不可以被继承
- 被 final 修饰的方法不可以被重写

- 被 final 修饰的变量不可以被改变，被 final 修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的。

> “
>
> *try-catch-finally* 中，如果 *catch* 中 *return* 了，*finally* 还会执行吗？

答：会执行，在 return 前执行。

注意：在 finally 中改变返回值的做法是不好的，因为如果存在 finally 代码块，try 中的 return 语句不会立马返回调用者，而是记录下返回值待 finally 代码块执行完毕之后再向调用者返回其值，然后如果在 finally 中修改了返回值，就会返回修改后的值。

显然，在 finally 中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java 中也可以通过提升编译器的语法检查级别来产生警告或错误。

```
public static int getInt() {
    int a = 10;
    try {
        System.out.println(a / 0);
        a = 20;
    } catch (ArithmeticException e) {
        a = 30;
        return a;
        /*
         * return a 在程序执行到这一步的时候，这里不是return a 而是 return 30;这个返回路径就形成了
         * 但是呢，它发现后面还有finally，所以继续执行finally的内容，a=40
         * 再次回到以前的路径,继续走return 30，形成返回路径之后，这里的a就不是a变量了，而是常量30
         */
    } finally {
        a = 40;
    }
 return a;
}
```

执行结果：30。

```
public static int getInt() {
    int a = 10;
    try {
        System.out.println(a / 0);
        a = 20;
    } catch (ArithmeticException e) {
        a = 30;
        return a;
    } finally {
```

```
            a = 40;
            //如果这样，就又重新形成了一条返回路径，由于只能通过1个return返回，所以这里直接返回40
            return a;
        }

    }
```

执行结果：40。

强引用、软引用、弱引用、虚引用

> “
>
> 强引用、软引用、弱引用、幻象引用有什么区别？具体使用场景是什么？

不同的引用类型，主要体现的是对象不同的可达性（reachable）状态和对垃圾收集的影响。

强引用

通过 new 创建的对象就是强引用，强引用指向一个对象，就表示这个对象还活着，垃圾回收不会去收集。

软引用

是一种相对强引用弱化一些的引用，只有当 JVM 认为内存不足时，才会去试图回收软引用指向的对象。

JVM 会确保在抛出 **OutOfMemoryError** 之前，清理软引用指向的对象。

软引用通常用来实现内存敏感的缓存，如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

弱引用

`ThreadlocalMap` 中的 key 就是用了弱引用，因为 `ThreadlocalMap` 被 thread 对象持有，所以如果是强引用的话，只有当 thread 结束时才能被回收，而弱引用则可以在使用完后立即回收，不必等待 thread 结束。

虚引用

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （`ReferenceQueue`）联合使用。

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，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

String 篇章

可变性

String 类中使用字符数组保存字符串，private final char value[]，所以 string 对象是不可变的。StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串，char[] value，这两种象都是可变的。·线程安全性

String 中的对象是不可变的，也就可以理解为常量，线程安全。
AbstractStringBuilder 是 StringBuilder 与 StringBuffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共方法。

StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。
StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。

StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StirngBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结

如果要操作少量的数据用 = String

单线程操作字符串缓冲区下操作大量数据 = StringBuilder

多线程操作字符串缓冲区下操作大量数据 = StringBuffer

String

String 是 Java 语言非常基础和重要的类，提供了构造和管理字符串的各种基本逻辑。它是典型的 Immutable 类，被声明成为 final class，所有属性也都是 final 的。

也由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。

StringBuilder

`StringBuilder` 是 Java 1.5 中新增的，在能力上和 `StringBuffer` 没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

StringBuffer

`StringBuffer` 是为解决上面提到拼接产生太多中间对象的问题而提供的一个类，我们可以用 `append` 或者 `add` 方法，把字符串添加到已有序列的末尾或者指定位置。

`StringBuffer` 本质是一个线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销，所以除非有线程安全的需要，不然还是推荐使用它的后继者，也就是 `StringBuilder`。

HashMap 使用 String 作为 key 有什么好处

`HashMap` 内部实现是通过 `key` 的 `hashcode` 来确定 `value` 的存储位置，因为字符串是不可变的，所以当创建字符串时，它的 `hashcode` 被缓存下来，不需要再次计算，所以相比于其他对象更快。

接口和抽象类有什么区别?

抽象类是用来捕捉子类的通用特性的。接口是抽象方法的集合。

接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：

- 行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量少用抽象类。
- 选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用的功能。

相同点

- 接口和抽象类都不能实例化
- 都位于继承的顶端，用于被其他实现或继承
- 都包含抽象方法，其子类都必须覆写这些抽象方法

接口

接口定义了协议，是面向对象编程（封装、继承多态）基础，通过接口我们能很好的实现单一职责、接口隔离、内聚。

- 不能实例化；
- 不能包含任何非常量成员，任何 field 都是隐含着 public static final 的意义；
- 同时，没有非静态方法实现，也就是说要么是抽象方法，要么是静态方法。

Java8 中接口中引入默认方法和静态方法，并且不用强制子类来实现它。以此来减少抽象类和接口之间的差异。

抽象类

抽象类是不能实例化的类，用 `abstract` 关键字修饰 class，其目的主要是代码重用。

从设计层面来说，抽象类是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

除了不能实例化，形式上和一般的 Java 类并没有太大区别。

可以有一个或者多个抽象方法，也可以没有抽象方法。抽象类大多用于抽取相关 Java 类的共用方法实现或者是共同成员变量，然后通过继承的方式达到代码复用的目的。

> 抽象类能用 *`final`* 修饰么？

不能，定义抽象类就是让其他类继承的，如果定义为 `final` 该类就不能被继承，这样彼此就会产生矛盾，所以 `final` 不能修饰抽象类

值传递

> “
>
> 当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递?

是值传递。

Java 语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。

对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。

> “
>
> 为什么 *Java* 只有值传递？

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。按值调用**(call by value)**表示方法接收的是调用者提供的值，而按引用调用（**call by reference)**表示方法接收的是调用者提供的变量地址。

一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。

它用来描述各种程序设计语言（不只是 Java)中方法参数传递方式。

Java 程序设计语言总是采用按值调用。也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

基本数据类型

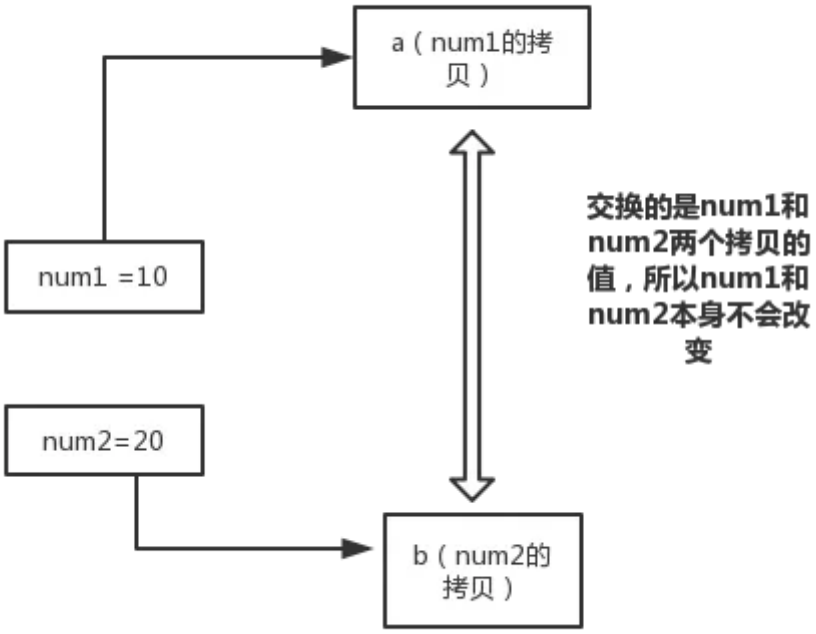
例子如下：

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 20;

    swap(num1, num2);

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;

    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

执行结果：

```
a = 20
b = 10
num1 = 10
num2 = 20
```

解析：



在 swap 方法中，a、b 的值进行交换，并不会影响到 num1、num2。

因为，a、b 中的值，只是从 num1、num2 的复制过来的。

也就是说，a、b 相当于 num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

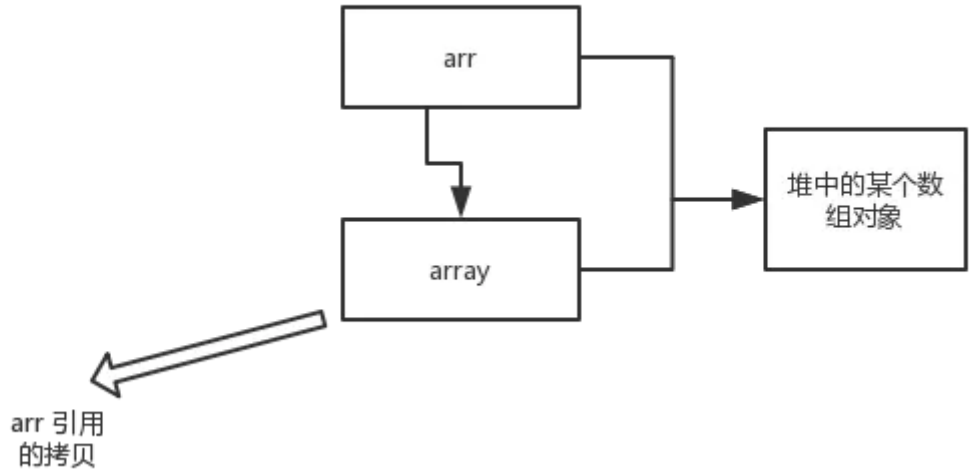
对象引用类型

```
    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 4, 5 };
        System.out.println(arr[0]);
        change(arr);
        System.out.println(arr[0]);
    }

    public static void change(int[] array) {
        // 将数组的第一个元素变为0
        array[0] = 0;
    }
```

结果：

```
1
0
```

解析：



array 被初始化 arr 的拷贝也就是一个对象的引用，也就是说 array 和 arr 指向的时同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象上。

通过 **example2** 我们已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，**C++**和 **Pascal)**提供了两种参数传递的方式：值调用和引用调用。

有些程序员认为 **Java** 程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。

值传递和引用传递有什么区别?

值传递：指的是在方法调用时，传递的参数是按值的拷贝传递，传递的是值的拷贝，也就是说传递后就互不相关了。

引用传递：指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所对应的内存空间的地址。传递的是值的引用，也就是说传递前和传递后都指向同一个引用（也就是同一个内存空间）。