Java基础

Java 语言有哪些特点?

1. 简单易学;
2. 面向对象（封装，继承，多态）；
3. 平台无关性（ Java 虚拟机实现平台无关性）；
4. 支持多线程（ C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；
5. 可靠性;
6. 安全性;
7. 支持网络编程并且很方便（ Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
8. 编译与解释并存;

JVM

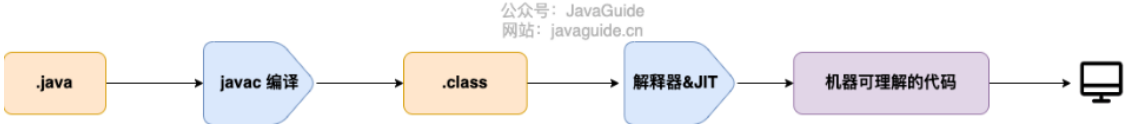
- Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows，Linux，macOS），目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。
- **JVM 并不是只有一种！只要满足 JVM 规范，每个公司、组织或者个人都可以开发自己的专属 JVM。** 也就是说我们平时接触到的 HotSpot VM 仅仅是是 JVM 规范的一种实现而已。
- OS运行在硬件之上，JVM运行在OS之上。

JDK和JRE

1. JDK 是 Java Development Kit 缩写，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc 和 jdb）。它能够创建和编译程序。
2. JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。
3. 如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet。

什么是字节码？采用字节码的好处是什么?

- 在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 `.class` 的文件），**它不面向任何特定的处理器，只面向虚拟机（也就是说字节码是跑在虚拟机上的）**。Java 语言通过字节码的方式，**在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。**所以， Java 程序运行时相对来说还是高效的（不过，和 C++，Rust，Go 等语言还是有一定差距的），而且，**由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。**
- Java 程序从源代码到运行的过程如下图所示：



- 我们需要格外注意的是 `.class->机器码` 这一步。在这一步 **JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。**而且，有些方法和代码块是经常需要被调用的(也就是所谓的热点代码)，所以后面引进了 JIT（just-in-time compilation） 编译器，而 JIT 属于运行时编译。**当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。**而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 **Java 是编译与解释共存的语言** 。
- JDK 9 引入了一种新的编译模式 AOT(Ahead of Time Compilation)，它是直接将字节码编译成机器码，这样就避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。

为什么不全部使用AOT呢?

- 长话短说，这和 Java 语言的动态特性有千丝万缕的联系了。举个例子，**CGLIB 动态代理使用的是 ASM 技术，而这种技术大致原理是运行时直接在内存中生成并加载修改后的字节码文件也就是 `.class` 文件，如果全部使用 AOT 提前编译，也就不能使用 ASM 技术了。**为了支持类似的动态特性，所以选择使用 JIT 即时编译器。

为什么说Java语言“编译与解释并存”?

- 这是因为 Java 语言既具有编译型语言的特征，也具有解释型语言的特征。因为 Java 程序要经过先编译，后解释两个步骤，由 Java 编写的程序需要先经过编译步骤，生成字节码（`.class` 文件），这种字节码必须由 Java 解释器来解释执行。

Oracle JDK和OpenJDK

- OpenJDK开源；Oracle JDK闭源，11之后，个人使用免费，但是商用收费。
- Oracle JDK 比 OpenJDK 更稳定
- 在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能

Java和C++的区别?

- Java 不提供指针来直接访问内存，程序内存更加安全。
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。
- Java 有自动内存管理垃圾回收机制(GC)，不需要程序员手动释放无用内存。
- C ++同时支持方法重载和操作符重载，但是 Java 只支持方法重载（操作符重载增加了复杂性，这与 Java 最初的设计思想不符）。

Java中的三种移位运算符

- `<<` :左移运算符，向左移若干位，高位丢弃，低位补零。`x << 1`,相当于 x 乘以 2(不溢出的情况下)。
- `>>` :带符号右移，向右移若干位，高位补符号位，低位丢弃。正数高位补 0,负数高位补 1。`x >> 1`,相当于 x 除以 2。
- `>>>` :无符号右移，忽略符号位，空位都以 0 补齐。
- 由于 `double`，`float` 在二进制中的表现比较特殊，因此不能来进行移位操作。
- 移位操作符实际上支持的类型只有`int`和`long`，编译器在对`short`、`byte`、`char`类型进行移位前，都会将其转换为`int`类型再操作。
- 如果移位的位数超过数值所占有的位数会怎样?
- 当 int 类型左移/右移位数大于等于 32 位操作时，会先求余（%）后再进行左移/右移操作。也就是说左移/右移 32 位相当于不进行移位操作（32%32=0），左移/右移 42 位相当于左移/右移 10 位（42%32=10）。当 long 类型进行左移/右移操作时，由于 long 对应的二进制是 64 位，因此求余操作的基数也变成了 64。

成员变量和局部变量的区别?

- **语法形式**：从语法形式上看，成员变量是属于类的，而局部变量是在代码块或方法中定义的变量或是方法的参数；成员变量可以被 `public`,`private`,`static` 等修饰符所修饰，而局部变量不能被访问控制修饰符及 `static` 所修饰；但是，成员变量和局部变量都能被 `final` 所修饰。
- **存储方式**：从变量在内存中的存储方式来看,如果成员变量是使用 `static` 修饰的，那么这个成员变量是属于类的，如果没有使用 `static` 修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。
- **生存时间**：从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动生成，随着方法的调用结束而消亡。
- **默认值**：从变量是否有默认值来看，成员变量如果没有被赋初始值，则会自动以类型的默认值而赋值（一种情况例外:被 `final` 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值。

静态方法为什么不能调用非静态成员?

1. 静态方法是属于类的，在类加载的时候就会分配内存，可以通过类名直接访问。而非静态成员属于实例对象，只有在对象实例化之后才存在，需要通过类的实例对象去访问。
2. 在类的非静态成员不存在的时候静态方法就已经存在了，此时调用在内存中还不存在的非静态成员，属于非法操作。

重载和重写的区别?

重载

- 发生在同一个类中（或者父类和子类之间），方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。

重写

- 重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。
- **方法名、参数列表必须相同，子类方法返回值类型应比父类方法返回值类型更小或相等，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。**
- **如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法，但是被 `static` 修饰的方法能够被再次声明（此时两个重名的static方法没有任何关系）。**
- 构造方法无法被重写
- **重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变。**

区别点	重载方法	重写方法
发生范围	同一个类	子类
参数列表	必须修改	一定不能修改
返回类型	可修改	子类方法返回值类型应比父类方法返回值类型更小或相等
异常	可修改	子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等;
访问修饰符	可修改	一定不能做更严格的限制（可以降低限制）
发生阶段	编译期	运行期

- 方法的重写要遵循“**两同两小一大**”：

- “两同”即方法名相同、形参列表相同；
- “两小”指的是子类方法返回值类型应比父类方法返回值类型更小或相等，子类方法声明抛出的异常类应比父类方法声明抛出的异常类更小或相等；
- “一大”指的是子类方法的访问权限应比父类方法的访问权限更大或相等。

可变长参数遇到方法重载怎么办？会优先匹配固定参数还是可变参数的方法呢？

- 答案是会优先匹配固定参数的方法，因为固定参数的方法匹配度更高。
- 另外，Java 的可变参数编译后实际会被转换成一个数组，我们看编译后生成的 `class` 文件就可以看出来了。

Java中的基本数据类型

- 6 种数字类型：
  - 4 种整数型：`byte`、`short`、`int`、`long`
  - 2 种浮点型：`float`、`double`
- 1 种字符类型：`char`
- 1 种布尔型：`boolean`
- 这 8 种基本数据类型的默认值以及所占空间的大小如下：

基本类型	位数	字节	默认值	取值范围
`byte`	8	1	0	-128 ~ 127
`short`	16	2	0	-32768 ~ 32767
`int`	32	4	0	-2147483648 ~ 2147483647
`long`	64	8	0L	-9223372036854775808 ~ 9223372036854775807
`char`	16	2	'u0000'	0 ~ 65535
`float`	32	4	0f	1.4E-45 ~ 3.4028235E38
`double`	64	8	0d	4.9E-324 ~ 1.7976931348623157E308
`boolean`	1		false	true、false

- 对于 `boolean`，官方文档未明确定义，它**依赖于 JVM 厂商的具体实现**。逻辑上理解是占用 1 位，但是实际中会考虑计算机高效存储因素。
- 另外，Java 的每种基本类型所占存储空间的大小不会像其他大多数语言那样随机器硬件架构的变化而变化。**这种所占存储空间大小的不变性是 Java 程序比用其他大多数语言编写的程序更具可移植性的原因之一。**
- 注意：
  1. Java 里使用 `long` 类型的数据一定要在数值后面加上 **L**，否则将作为整型解析。
  2. 这八种基本类型都有对应的包装类分别为：`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`、`Character`、`Boolean`

基本类型和包装类型的区别？

1. 成员变量包装类型不赋值就是 `null` ，而基本类型有默认值且不是 `null`。

2. 包装类型可用于泛型，而基本类型不可以。
3. 基本数据类型的局部变量存放在 Java 虚拟机栈中的局部变量表中，基本数据类型的成员变量（未被 `static` 修饰 ）存放在 Java 虚拟机的堆中。包装类型属于对象类型，我们知道**几乎所有对象实例都存在于堆中。**
4. 相比于对象类型， 基本数据类型占用的空间非常小。

- 注意： **基本数据类型存放在栈中是一个常见的误区！** 基本数据类型的成员变量如果没有被 `static` 修饰的话（不建议这么使用，应该要使用基本数据类型对应的包装类型），就存放在堆中。

所有的对象实例都存在堆中吗?

- 不是，是几乎所有对象实例都存在堆中，**为什么说是几乎所有对象实例呢?** 这是因为 HotSpot 虚拟机引入了 JIT 优化之后，会对对象进行逃逸分析，**如果发现某一个对象并没有逃逸到方法外部，那么就可能通过标量替换来实现栈上分配，而避免堆上分配内存。**
- **标量替换**：通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而是**将该对象成员变量分解若干个被这个方法使用的成员变量所代替，这些代替的成员变量在栈帧或寄存器上分配空间，这样随着方法调用结束，栈帧也会销毁。有效的减少了堆中创建对象及gc的次数**

包装类型的缓存机制了解么?

- Java 基本数据类型的包装类型的大部分都用到了缓存机制来提升性能。
- `Byte`,`Short`,`Integer`,`Long` 这 4 种包装类默认创建了数值 **[-128，127]** 的相应类型的缓存数据，`Character` 创建了数值在 **[0,127]** 范围的缓存数据，`Boolean` 直接返回 `True` or `False`。
- 如果**超出对应范围仍然会去创建新的对象**，缓存的范围区间的大小只是在性能和资源之间的权衡。
- 两种浮点数类型的包装类 `Float`,`Double` 并没有实现缓存机制。
- Integer缓存源码（自动装箱用的就是valueOf方法）：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)//如果范围在缓存中，直接返回缓存的数据
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);//否则新建对象
}
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static {
        // high value may be configured by property
        int h = 127;
    }
}
```

- 记住：**所有整型包装类对象之间值的比较，全部使用 equals 方法比较**：

> 7. 【强制】所有整型包装类对象之间值的比较，全部使用 equals 方法比较。
>
> 说明：对于 Integer var = ? 在-128 至 127 之间的赋值，Integer 对象是在 IntegerCache.cache 产生，会复用已有对象，这个区间内的 Integer 值可以直接使用==进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 equals 方法进行判断。

自动装箱与拆箱了解吗？原理是什么？

- 从字节码中，我们发现**装箱其实就是调用了 包装类的`valueOf()`方法，拆箱其实就是调用了`xxxValue()`方法。**
- 因此：
  - `Integer i = 10` 等价于 `Integer i = Integer.valueOf(10)`
  - `int n = i` 等价于 `int n = i.intValue()`;
- 注意：**如果频繁拆装箱的话，也会严重影响系统的性能。我们应该尽量避免不必要的拆装箱操作。**

为什么浮点数运算的时候会有精度丢失的风险？

- 这个和计算机保存浮点数的机制有很大关系。我们知道计算机是二进制的，而且**计算机在表示一个数字时，宽度是有限的，无限循环的小数存储在计算机时，只能被截断，所以就会导致小数精度发生损失的情况。这也就是解释了为什么浮点数没有办法用二进制精确表示。**
- 比如十进制的0.2要存放在计算机中，那肯定要转换成2进制来存储，而转成的二进制可能是无限长的，所以必须截断：

```
// 0.2 转换为二进制数的过程为，不断乘以 2，直到不存在小数为止，
// 在这个计算过程中，得到的整数部分从上到下排列就是二进制的结果。
0.2 * 2 = 0.4 -> 0 //第一位二进制小数，取整数部分0，余下的参与下次运算
0.4 * 2 = 0.8 -> 0 //第二位二进制小数，取整数部分0，余下的参与下次运算
0.8 * 2 = 1.6 -> 1 //第三位二进制小数，取整数部分1，余下的参与下次运算
0.6 * 2 = 1.2 -> 1 //第四位二进制小数，取整数部分1，余下的参与下次运算
0.2 * 2 = 0.4 -> 0（发生循环）//第五位二进制小数，取整数部分0，余下的参与下次运算，在这里发生循环，也就意味着0.2转换成的二进制将会是无限循环的，所以必须截断
...
```

如何解决浮点数运算的精度丢失问题？

- `BigDecimal` 可以实现对浮点数的运算，不会造成精度丢失。通常情况下，大部分需要浮点数精确运算结果的业务场景（比如涉及到钱的场景）都是通过 `BigDecimal` 来做的。
- BigDecimal中提供各种各样的方法来实现BigDecimal对象之间的运算，所以运算的时候要使用其中的方法。

超过long整型的数据应该如何表示？

- 在 Java 中，64 位 long 整型是最大的整数类型。超过了可以用`BigInteger`，其内部使用 `int[]` 数组来存储任意大小的整形数据。和BigDecimal一样，其中提供了各种供对象间进行运算的方法。
- 相对于常规整数类型的运算来说，`BigInteger` 运算的效率会相对较低。

面向对象和面向过程的区别

- 两者的主要区别在于解决问题的方式不同：
  - 面向过程把解决问题的过程拆成一个个方法，通过一个个方法的执行解决问题。
  - 面向对象会先抽象出对象，然后用对象执行方法的方式解决问题。
- 另外，面向对象开发的程序一般更易维护、易复用、易扩展。

创建一个对象用什么运算符?对象实体与对象引用有何不同?

- new 运算符，new 创建对象实例（对象实例在堆内存中），对象引用指向对象实例（就是指针，指向对象实例的位置）。
- 一个对象引用可以指向 0 个或 1 个对象（一根绳子可以不系气球，也可以系一个气球）;一个对象可以有 n 个引用指向它（可以用 n 条绳子系住一个气球）。

对象相等和引用相等的区别?

- 对象的相等一般比较的是内存中存放的内容是否相等。
- 引用相等一般比较的是他们指向的内存地址是否相等。

类的构造方法的作用是什么?

- 构造方法是一种特殊的方法，主要作用是完成对象的初始化工作。

如果一个类没有声明构造方法，该程序能正确执行吗?

- 如果一个类没有声明构造方法，也可以执行！因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。如果我们自己添加了类的构造方法（无论是否有参），Java 就不会再添加默认的无参数的构造方法了，我们一直在不知不觉地使用构造方法，这也是为什么我们在创建对象的时候后面要加一个括号（因为要调用无参的构造方法）。如果我们重载了有参的构造方法，记得都要把无参的构造方法也写出来（无论是否用到），因为这可以帮助我们在创建对象的时候少踩坑。

构造方法有哪些特点？是否可被 override?

- 构造方法特点如下：
    1. 名字与类名相同。
    2. 没有返回值，但不能用 void 声明构造函数。
    3. 生成类的对象时自动执行，无需调用。
- 构造方法不能被 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。

面向对象四大特征

抽象

- 就是指一种思想，我们可以将客观存在的一切事物都抽象成java类，其中属性表示事物的状态信息，方法表示事物可以有的行为。

封装

- 封装是指**把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性。**就好像我们看不到挂在墙上的空调的内部的零件信息（也就是属性），但是可以通过遥控器（方法）来控制空调。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。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。就好像如果没有空调遥控器，那么我们就无法操控空凋制冷，空调本身就没有意义了（当然现在还有很多其他方法，这里只是为了举例子）。

继承

- 不同类型的对象，相互之间经常有一定数量的共同点。例如，小明同学、小红同学、小李同学，都共享学生的特性（班级、学号等）。同时，每一个对象还定义了额外的特性使得他们与众不同。例如小明的数学比较好，小红的性格惹人喜爱；小李的力气比较大。**继承是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，可以快速地创建新的类，可以提高代码的重用，程序的可维护性，节省大量创建新类的时间 ，提高我们的开发效率。**
- 关于继承如下 3 点请记住：
    1. 子类拥有父类对象所有的属性和方法（**包括私有属性和私有方法**），但是父类中的私有属性和方法子类是无法访问，**只是拥有。**
    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。

多态

- 多态，顾名思义，表示一个对象具有多种的状态，具体表现为父类的引用指向子类的实例。
- 多态的特点：
    1. 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
    2. 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
    3. 多态不能调用“只在子类存在但在父类不存在”的方法；
    4. 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。

接口和抽象类有什么共同点和区别?

- 共同点：
    1. 都不能被实例化。
    2. 都可以包含抽象方法。
    3. 都可以有默认实现的方法（Java 8 可以用 `default` 关键字在接口中定义默认方法）。
- 区别：
    1. 接口主要用于对类的行为进行约束，你实现了某个接口就具有了对应的行为。抽象类主要用于代码复用，强调的是所属关系。
    2. 一个类只能继承一个类，但是可以实现多个接口。
    3. 接口中的成员变量只能是 `public static final` 类型的，不能被修改且必须有初始值，而抽象类的成员变量默认 default，可在子类中被重新定义，也可被重新赋值。

深拷贝和浅拷贝区别了解吗？什么是引用拷贝?

- **浅拷贝**：浅拷贝会在堆上创建一个新的对象（区别于引用拷贝的一点），不过，如果原对象内部的属性是引用类型的话，浅拷贝会直接复制内部对象的引用地址，也就是说拷贝对象和原对象共用同一个内部对象。
- **深拷贝** ：深拷贝会完全复制整个对象，包括这个对象所包含的内部对象。
- **引用拷贝**：引用拷贝就是两个不同的引用指向同一个对象。

Object类的常见方法有哪些?

- Object 类是一个特殊的类，是所有类的父类。它主要提供了以下 11 个方法：

```
```

```
/**
 * native 方法，用于返回当前运行时对象的 Class 对象，使用了 final 关键字修饰，故不允许子类重写。
 */
public final native Class<?> getClass()
/**
 * native 方法，用于返回对象的哈希码，主要使用在哈希表中，比如 JDK 中的HashMap。
 */
public native int hashCode()
/**
 * 用于比较 2 个对象的内存地址是否相等，String 类对该方法进行了重写以用于比较字符串的值是否相等。
 */
public boolean equals(Object obj)
/**
 * naitive 方法，用于创建并返回当前对象的一份拷贝。
 */
protected native Object clone() throws CloneNotSupportedException
/**
 * 返回类的名字实例的哈希码的 16 进制的字符串。建议 Object 所有的子类都重写这个方法。
 */
public String toString()
/**
 * native 方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。
 */
public final native void notify()
/**
 * native 方法，并且不能重写。跟 notify 一样，唯一的区别就是会唤醒在此对象监视器上等待的所有线程，而不是一个线程。
 */
public final native void notifyAll()
/**
 * native方法，并且不能重写。暂停线程的执行。注意：sleep 方法没有释放锁，而 wait 方法释放了锁 ，timeout 是等待时间。
 */
public final native void wait(long timeout) throws InterruptedException
/**
 * 多了 nanos 参数，这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。 所以超时的时间还需要加上 nanos 毫秒。。
 */
public final void wait(long timeout, int nanos) throws InterruptedException
/**
 * 跟之前的2个wait方法一样，只不过该方法一直等待，没有超时时间这个概念
 */
public final void wait() throws InterruptedException
/**
 * 实例被垃圾回收器回收的时候触发的操作，第一次回收触发这个方法，然后会将对象放到一个队列中，此时对象的内存并没有被真正释放；第二次回收的时候会判断队列中的对象是否通过finalize方法重新建立起了引用，如果有，那就不回收了，如果没有，直接回收掉，这次是真回收，会释放内存空间。
 */
protected void finalize() throws Throwable { }
```

==和equals()的区别

- `==` 对于基本类型和引用类型的作用效果是不同的：
  - 对于基本数据类型来说，`==` 比较的是值。
  - 对于引用数据类型来说，`==` 比较的是对象的内存地址。
- 因为 Java 只有值传递，所以，对于 == 来说，不管是比较基本数据类型，还是引用数据类型的变量，其本质比较的都是值，只是引用类型变量存的值是对象的地址。
- `equals()` **不能用于判断基本数据类型的变量，只能用来判断两个对象是否相等（这个相等的定义是自己定义的）。** `equals()` 方法存在于 `Object` 类中，而 `Object` 类是所有类的直接或间接父类，因此所有的类都有 `equals()` 方法。
- `equals()` 方法存在两种使用情况：
  1. **类没有重写 `equals()` 方法**： 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，使用的默认是 `Object` 类 `equals()` 方法。
  2. **类重写了 `equals()` 方法**： 一般我们都重写 `equals()` 方法来比较两个对象中的属性是否相等；若它们的属性相等，则返回 true(即，认为这两个对象相等)。

hashCode()有什么用?

- `hashCode()` 的作用是获取哈希码（`int` 整数），也称为散列码。这个哈希码的作用是确定该对象在哈希表中的索引位置。
- `hashCode()` 定义在 JDK 的 `Object` 类中，这就意味着 Java 中的任何类都包含有 `hashCode()` 函数。另外需要注意的是： `Object` 的 `hashCode()` 方法是本地方法，也就是用 C 语言或 C++ 实现的，该方法通常用来将对象的内存地址转换为整数之后返回。
- 散列表存储的是键值对(key-value)，它的特点是：**能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）**
- 可以**优化对象比较大小时的速度**，通过先比较hashcode，如果hashcode不同，那么这两个对象一定是不同的，如果hashcode相同，再通过equals()方法来进一步比较确定两者是否相同。

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`？
- 下面这段内容摘自我的 Java 启蒙书《Head First Java》：

  > 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。

  其实， `hashCode()` 和 `equals()` 都是用于比较两个对象是否相等。
- **那为什么 JDK 还要同时提供这两个方法呢？** 这是因为在一些容器（比如 `HashMap`、 `HashSet`）中，有了 `hashCode()` 之后，判断元素是否在对应容器中的效率会更高（参考添加元素进 `HashSet` 的过程）！我们在前面也提到了添加元素进 `HashSet` 的过程，如果 `HashSet` 在对比的时候，同样的 `hashCode` 有多个对象，它会继续使用 `equals()` 来判断是否真的相同。也就是说 `hashCode` 帮助我们大大缩小了查找成本。
- **那为什么不只提供 `hashCode()` 方法呢？** 这是因为两个对象的 `hashCode` 值相等并不代表两个对象就相等。

- **那为什么两个对象有相同的 `hashCode` 值，它们也不一定是相等的？** 因为 `hashCode()` 所使用的哈希算法也许刚好会让多个对象传回相同的哈希值。越糟糕的哈希算法越容易碰撞，但这也与数据值域分布的特性有关（所谓哈希碰撞也就是指的是不同的对象得到相同的 `hashCode` ）。
- 总结下来就是：
    1. 如果两个对象的 `hashCode` 值相等，那这两个对象不一定相等（哈希碰撞）。
    2. 如果两个对象的 `hashCode` 值相等并且 `equals()` 方法也返回 `true`，我们才认为这两个对象相等。
    3. 如果两个对象的 `hashCode` 值不相等，我们就可以直接认为这两个对象不相等。

为什么重写equals()时必须重写hashCode()方法?

- 因为两个相等的对象的 `hashCode` 值必须是相等。也就是说如果 `equals` 方法判断两个对象是相等的，那这两个对象的 `hashCode` 值也要相等。
- 如果重写 `equals()` 时没有重写 `hashCode()` 方法的话就可能会导致 `equals` 方法判断是相等的两个对象， `hashCode` 值却不相等。
- **思考**： 重写 `equals()` 时没有重写 `hashCode()` 方法的话，使用 `HashMap` 可能会出现什么问题。
- **总结**：
    - `equals` 方法判断两个对象是相等的，那这两个对象的 `hashCode` 值也要相等。
    - 两个对象有相同的 `hashCode` 值，他们也不一定是相等的（哈希碰撞）。

String、StringBuffer、StringBuilder 的区别?

可变性

- `String` 是不可变的（后面会详细分析原因）。
- `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，不过没有使用 `final` 和 `private` 关键字修饰，最关键的是这个 `AbstractStringBuilder` 类还提供了很多修改字符串的方法比如 `append` 方法。

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    char[] value;
    public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();
        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);
        count += len;
        return this;
    }
    //...
}
```

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 `StringBuilder` 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

1. 操作少量的数据: 适用 `String`
2. 单线程操作字符串缓冲区下操作大量数据: 适用 `StringBuilder`
3. 多线程操作字符串缓冲区下操作大量数据: 适用 `StringBuffer`

String为什么是不可变的?

1. 保存字符串的字符数组（jdk1.9之后是字节数组）被 `final` 修饰且为私有的，并且`String` 类没有提供/暴露修改这个字符串的方法。
2. `String` 类被 `final` 修饰导致其不能被继承，进而避免了子类破坏 `String` 不可变。

String不可变的好处?

便于实现String常量池

- 在程序编写过程中往往会大量用到String常量，如果每次都为新的String常量申请内存空间的话会对空间造成很大的浪费，所以我们需要字符串常量池来对String常量做一个优化，把之前用过的String保存下来，以后如果再用的话可以直接拿。
- 从上面我们知道了我们需要一个字符串常量池来对使用String做优化，而只有当字符串是不可变的，字符串池才有可能实现。**字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，因为这样的话，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。**

可以很方便地作为Map的key

- 以String作为HashMap的key，String的不可变性保证了其hash值的不可变性，防止出现key值发生变化导致key对象的hash值发生变化，之后再用key的那个对象在map中找对应的value就找不到了（因为hash值变了，找的位置就不对了）。

避免网络安全问题

- 如果字符串是可变的，那么会引起很严重的安全问题。譬如，数据库的用户名、密码都是以字符串的形式传入来获得数据库的连接，或者在socket编程中，主机名和端口都是以字符串的形式传入。因为字符串是不可变的，所以它的值是不可改变的，否则黑客们可以钻到空子，改变字符串指向的对象的值，造成安全漏洞。

使多线程安全

- 因为字符串是不可变的，所以是多线程安全的，同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。

避免本地安全性问题

- 类加载器要用到字符串，不可变性提供了安全性，以便正确的类被加载。譬如你想加载java.sql.Connection类，而这个值被改成了myhacked.Connection，那么会对你的数据库造成不可知的破坏。

加快字符串处理速度

- 因为字符串是不可变的，所以在它创建的时候hashcode就被缓存在对象头中了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

Java9为何要将String的底层实现由char[]改成了byte[]?

- 新版的 String 其实支持两个编码方案： Latin-1 和 UTF-16。JDK 官方就说了绝大部分字符串对象只包含 Latin-1 可表示的字符。如果字符串中包含的汉字没有超过 Latin-1 可表示范围内的字符，那就会使用 Latin-1 作为编码方案。Latin-1 编码方案下，`byte` 占一个字节(8 位)，`char` 占用 2 个字节（16），`byte` 相较 `char` 节省一半的内存空间。
- 如果字符串中包含的汉字超过 Latin-1 可表示范围内的字符，`byte` 和 `char` 所占用的空间是一样的。

字符串拼接用“+”还是StringBuilder?

- Java 语言本身并不支持运算符重载，“+”和“+=”是专门为 String 类重载过的运算符，也是 Java 中仅有的两个重载过的运算符。
- 通过编译后形成的字节码文件的内容可以看出，**字符串对象通过“+”的字符串拼接方式，实际上是通过 `StringBuilder` 调用 `append()` 方法实现的，拼接完成之后调用 `toString()` 得到一个 `String` 对象** 。
- 不过，在循环内使用“+”进行字符串的拼接的话，存在比较明显的缺陷：**编译器不会创建单个 `StringBuilder` 以复用，会导致创建过多的 `StringBuilder` 对象。**
- 所以**推荐在循环中拼接字符串时应该事先在循环外面声明好StringBuilder然后在循环中使用StringBuilder的append()方法。**

字符串常量池的作用了解吗?

- **字符串常量池（jdk1.7之前在方法区的常量池中，jdk1.7的时候字符串常量池被从常量池中单独拎出来放到了堆中；jdk1.8的时候原本在方法区中的每个类特有的静态域也被放到了堆中，具体存放在堆中类在加载阶段生成的class对象的尾部）** 是 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，**主要目的是为了避免字符串的重复创建。**

String s1 = new String("abc");这句话创建了几个字符串对象?

- 会创建 1 或 2 个字符串对象。
- 如果字符串常量池中不存在字符串对象“abc”的引用，那么会在堆中（看这说法应该默认是jdk1.7之后的版本）创建 2 个字符串对象“abc”；反之则只会创建1个。

intern方法有什么作用?

- `String.intern()` 是一个 native（本地）方法，**其作用是将指定的字符串对象的引用保存在字符串常量池中**，可以简单分为两种情况：
  1. 如果字符串常量池中保存了对应的字符串对象的引用，就直接返回该引用。
  2. 如果字符串常量池中没有保存了对应的字符串对象的引用，那就在常量池中创建一个指向该字符串对象的引用并返回。

String 类型的引用和字面量做“+”运算时发生了什么?

- 先来看字符串不加 `final` 关键字拼接的情况（JDK1.8）：

```
String str1 = "str";
String str2 = "ing";
String str3 = "str" + "ing";
String str4 = str1 + str2;
String str5 = "string";
System.out.println(str3 == str4);//false
System.out.println(str3 == str5);//true
System.out.println(str4 == str5);//false
```

- **对于编译期可以确定值的字符串，也就是字面量字符串 ，jvm 会将其存入字符串常量池。并且，字符串常量拼接得到的字符串字面量在编译阶段就已经被存放字符串常量池，这个得益于编译器的优化。**
- 在编译过程中，Javac 编译器（下文中统称为编译器）会进行一个叫做 **常量折叠(Constant Folding)** 的代码优化。**常量折叠会把常量表达式的值求出来作为常量嵌在最终生成的代码中**，这是 Javac 编译器会对源代码做的极少量优化措施之一(代码优化几乎都在即时编译器中进行)。
- 所以，对于 `String str3 = "str" + "ing";` 编译器会给你优化成 `String str3 = "string";` 。
- 不过要注意，**并不是所有的常量/变量都会进行折叠，只有编译器在程序编译期就可以确定值的常量/变量才可以**：
  - 基本数据类型(`byte`、`boolean`、`short`、`char`、`int`、`float`、`long`、`double`)以及字符串的字面量。
  - `final` 修饰的基本数据类型和字符串引用
  - 字符串通过 “+”拼接得到的字符串、基本数据类型之间算数运算（加减乘除）、基本数据类型的位运算（<<、>>、>>> ）
- **引用的值在程序编译期是无法确定的（因为值是地址，非final的地址程序不跑起来定不下来），编译器无法对其进行优化（除非引用是final的，final的引用的值在编译期是确定的）。**
- 对象引用和“+”的字符串拼接方式，实际上是通过 `StringBuilder` 调用 `append()` 方法实现的，拼接完成之后调用 `toString()` 得到一个 `String` 对象 。所以我们在平时写代码的时候，尽量避免多个字符串对象拼接，因为这样会重新创建对象。如果需要改变字符串的话，可以使用 `StringBuilder` 或者 `StringBuffer`。
- 不过，**字符串引用使用的都是被 `final` 关键字修饰的字符串引用时，可以让编译器把这个引用当做字面量来做优化处理，因为final引用的值在编译期就会被确定。**

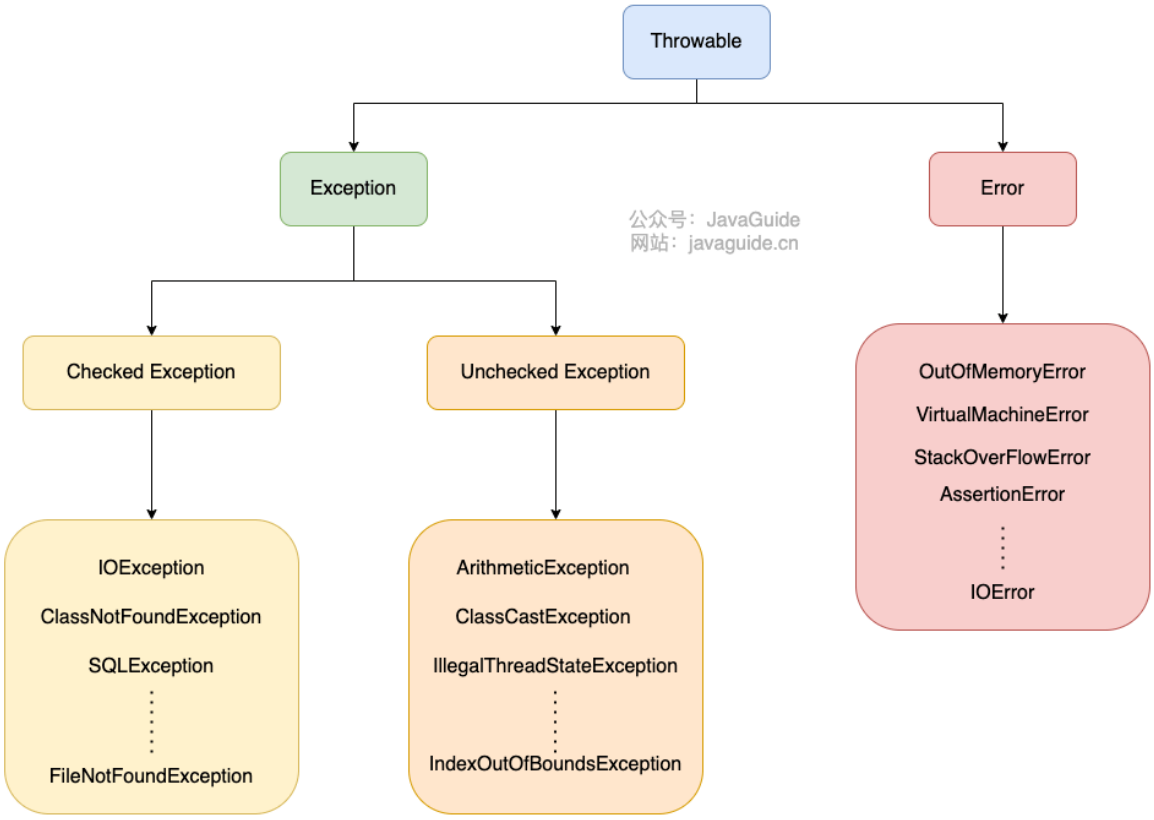
```
final String str1 = "str";
final String str2 = "ing";
// 下面两个表达式其实是等价的
String c = "str" + "ing";// 常量池中的对象
String d = str1 + str2; // 常量池中的对象
System.out.println(c == d);// true
```

- 被 `final` 关键字修改之后的 `String` 会被编译器当做常量来处理，编译器在程序编译期就可以确定它的值，其效果就相当于访问常量。
- 如果，编译器在运行时才能知道其确切值的话，就无法对其优化：

```
final String str1 = "str";
final String str2 = getStr();
String c = "str" + "ing";// 常量池中的对象
String d = str1 + str2; // 在堆上创建的新的对象
System.out.println(c == d);// false
public static String getStr() {
      return "ing";
}
```

- **总之就是一点，看编译时变量/常量的值是否能被确定，如果能，那么编译器就会做常量折叠优化；反之则不做优化。**

Java异常类层次结构图概览



Exception和Error有什么区别?

- 在 Java 中，**所有的异常都有一个共同的祖先 `java.lang` 包中的 `Throwable` 类。** `Throwable` 类有两个重要的子类:

- `Exception`:程序本身可以处理的异常，可以通过 `catch` 来进行捕获。`Exception` 又可以分为 Checked Exception (受检查异常，必须处理) 和 Unchecked Exception (不受检查异常，可以不处理)。
- `Error`：`Error` 属于程序无法处理的错误，我们没办法通过 `catch` 来进行捕获不建议通过`catch`捕获 。例如 Java 虚拟机运行错误（`Virtual MachineError`）、虚拟机内存不够错误(`OutOfMemoryError`)、类定义错误（`NoClassDefFoundError`）等 。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

Checked Exception 和 Unchecked Exception 有什么区别?

- **Checked Exception** 即 受检查异常，Java 代码在编译过程中，如果受检查异常没有被 `catch`或者`throws` 关键字处理的话，就没办法通过编译。
- **除了`RuntimeException`及其子类以外，其他的`Exception`类及其子类都属于受检查异常。**常见的受检查异常有： IO 相关的异常、 `ClassNotFoundException` 、 `SQLException`...
- **Unchecked Exception** 即 **不受检查异常** ，Java 代码在编译过程中 ，我们即使不处理不受检查异常也可以正常通过编译。
- `RuntimeException` **及其子类都统称为非受检查异常**，常见的有（建议记下来，日常开发中会经常用到）：
  - `NullPointerException`(空指针错误)
  - `IllegalArgumentException`(参数错误比如方法入参类型错误)
  - `NumberFormatException`（字符串转换为数字格式错误，`IllegalArgumentException`的子类)
  - `ArrayIndexOutOfBoundsException`（数组越界错误)
  - `ClassCastException`（类型转换错误)
  - `ArithmeticException`（算术错误)
  - `SecurityException` （安全错误比如权限不够)
  - `UnsupportedOperationException`(不支持的操作错误比如重复创建同一用户)
  - ......

Throwable类常用方法有哪些?

- `String getMessage()`: 返回异常发生时的简要描述
- `String toString()`: 返回异常发生时的详细信息
- `String getLocalizedMessage()`: 返回异常对象的本地化信息。使用 `Throwable` 的子类覆盖这个方法，可以生成本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与 `getMessage()` 返回的结果相同
- `void printStackTrace()`: 在控制台上打印 `Throwable` 对象封装的异常信息

try-catch-finally如何使用?

JVM

java对象在堆中的结构

- 主要是分为三部分：**对象头、实例数据、对齐填充**

对象头（header）

- 分为两大块，第一块是**占8个字节的markword，其中存放着锁信息、hashcode、GC信息**，第二块是**占4个字节（压缩前是8个字节，默认压缩）的类型指针（class pointer），这个没啥好说的，就是存放的是这个对象的类的元数据（在方法区中）的地址。**
- 也就是说一般情况下对象头占12个字节。
- 平时所说的拿到了对象的锁，实际上从底层来看就是修改了对象头中的markword中的锁信息
- 在调用一次hashcode方法之后，这个对象的hashcode就会被记录到对象头的markword中，下次使用到的时候可以直接拿
- 对象头的markword中的GC信息记录的是这个对象的GC分代年龄

实例数据（instance data）

- 这一部分**存放着对象的所有实例属性。**

对齐填充（padding）

- 所有对象的大小（以字节为单位）都是8的倍数，这是为了契合64位处理器的位宽，这也是64位处理器能一次处理的数据大小（也就是一个字长的大小），所以对于那些前两个部分加起来不是8的倍数的对象，我们要**补齐最后一段，从而让对象的大小是8的倍数**，这一部分仅作补齐，没有其他含义。

对象如何定位

- 当用一个引用指向一个对象的时候，这个引用是如何找到那个对象的呢?

通过指针直接定位

- 这种方式也是hotspot虚拟机使用的方式，**引用中存放着的是对象的地址**，通过引用可以直接访问到对象。
- 好处就是**访问效率高**，坏处就是GC的时候对象会移动，而引用的值也要随之而改变，**GC效率相较间接定位的方式偏低。**

通过句柄间接定位

- 这种方式，**引用中存放的是一个句柄的地址**，而这个**句柄中又有两个直接指针分别指向对象和它的类的元数据**，这种方式相当于通过句柄间接访问对象。
- 句柄通常存放在句柄池中，而**句柄池通常是在堆中。**
- 这种方式的好处在于，在进行GC的时候，内存中的对象的位置通常会发生改变（），而使用这种间接定位的方式，由于引用指向的是句柄，而句柄的位置不变，所以引用的值不需要改变，由于引用的值控制着很多GC的参数，所以它不变的话**可以提高GC的效率。**坏处就在于需要两次才能定位对象，**定位的效率相较直接定位方式偏低。**