Oracle JDK 和 OpenJDK 的对比

对于Java 7，没什么关键的地方。OpenJDK项目主要基于Sun捐赠的HotSpot源代码。此外，OpenJDK被选为Java 7的参考实现，由Oracle工程师维护。

问：OpenJDK存储库中的源代码与用于构建Oracle JDK的代码之间有什么区别？

答：非常接近 - 我们的Oracle JDK版本构建过程基于OpenJDK 7构建，只添加了几个部分，例如部署代码，其中包括Oracle的Java插件和Java WebStart的实现，以及一些封闭的源代码派对组件，如图形光栅化器，一些开源的第三方组件，如Rhino，以及一些零碎的东西，如附加文档或第三方字体。展望未来，我们的目的是开源Oracle JDK的所有部分，除了我们考虑商业功能的部分。

penJDK 是一个参考模型并且是完全开源的，而Oracle JDK是OpenJDK的一个实现，并不是完全开源的；

Oracle JDK 比 OpenJDK 更稳定。OpenJDK和Oracle JDK的代码几乎相同，但Oracle JDK有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到Oracle JDK就可以解决问题；

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要确定每种基本类型所占存储空间的大小。它们的大小并不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。

基本类型	大小	最小值	最大值	包装器类型
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode $2^{16}-1$	**Character**
byte	8 bits	-128	+127	**Byte**
short	16 bits	-2^{15}	$+2^{15}-1$	**Short**
int	32 bits	-2^{31}	$+2^{31}-1$	**Integer**
long	64 bits	-2^{63}	$+2^{63}-1$	**Long**
float	32 bits	IEEE754	IEEE754	**Float**
double	64 bits	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

构造器 Constructor 是否可被 override?

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以 Constructor 也就不能被 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。

重载和重写的区别

重载： 发生在同一个类中，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，发生在编译时。
重写： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，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 则子类就不能重写该方法。
在 Java 中定义一个不做事且没有参数的构造方法的作用
Java 程序在执行子类的构造方法之前，如果没有用 super()来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 super()来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

接口和抽象类的区别是什么？
接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法。
接口中除了static、final变量，不能有其他变量，而抽象类中则不一定。
一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过extends关键字扩展多个接口。
接口方法默认修饰符是public，抽象方法可以有public、protected和default这些修饰符（抽象方法就是为了被重写所以不能使用private关键字修饰！）。
从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。
备注：在JDK8中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。

多态中成员变量的访问特点
成员变量：编译看左边，执行看左边；
成员方法：编译看左边，执行看右边。

```
public class A{
   public int age = 20;       // (1)
   public void print(){       // (2)
      System.out.println("父类输出");
   }
}

public class B extends A{
```

```
    public int age = 30;        // (3)
    public void print(){       // (4)
        System.out.println("子类输出");
    }
    public void subClass(){    // (5)
        System.out.println("子类特有的方法");
    }
}

public class Test{
    public static void main(String[] args){
        A a = new B();
        a.print();    // 调用(4)
        a.age;        // 20
        // a.subClass();// A中没有此方法报错，A a = new B();引用看左边，执行看右边
    }
}
```

多态的好处和弊端

好处：提高了程序的扩展性，定义方法的时候，使用父类型作为参数，将来使用的时候，使用具体的子类型参与操作。

弊端：不能使用子类的特有功能

== 与 equals(重要)

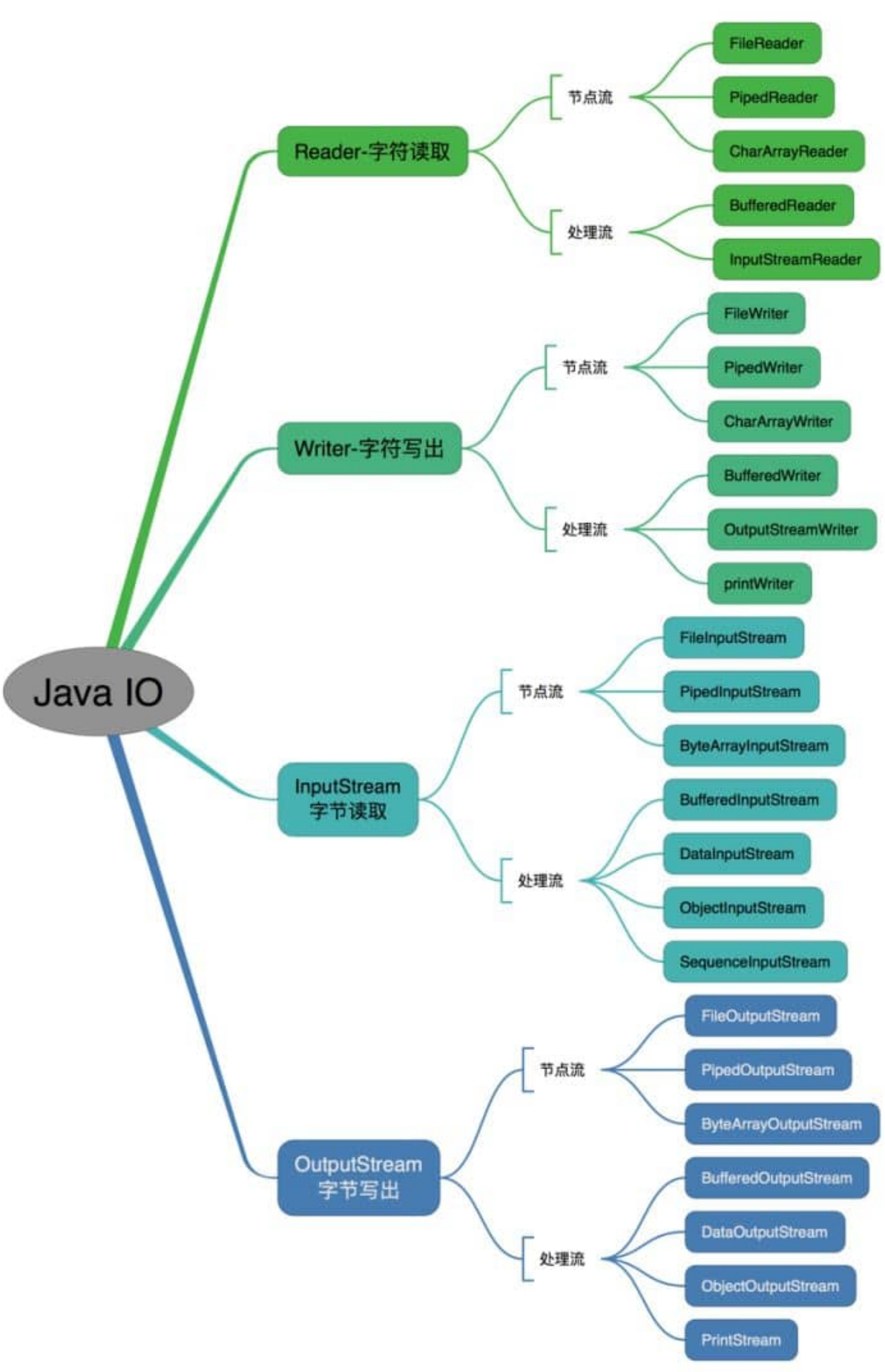
== ：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)。

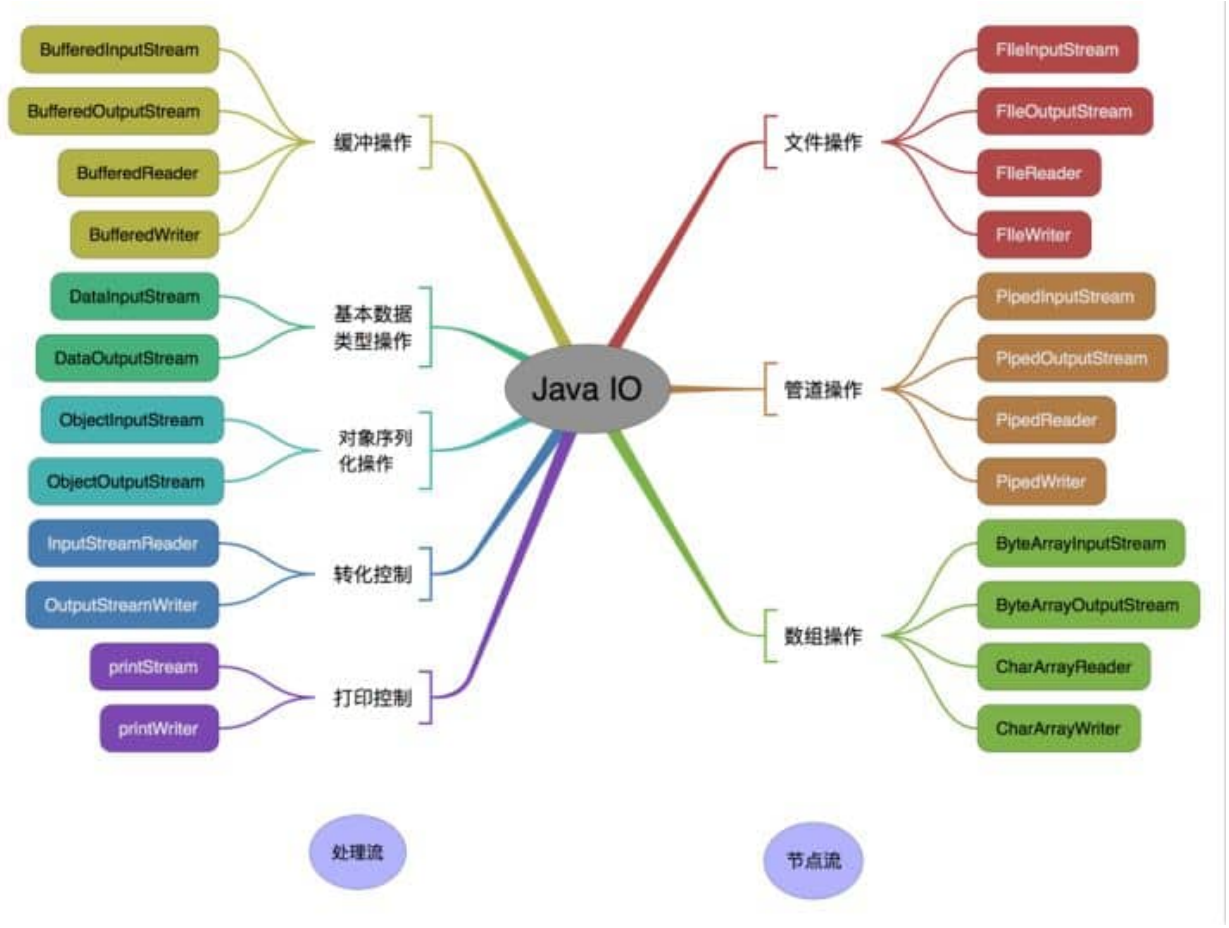
equals() ：它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

情况1：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。

情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

Java IO


Java IO
Reader-字符读取
节点流
FileReader
PipedReader
CharArrayReader
处理流
BufferedReader
InputStreamReader
Writer-字符写出
节点流
FileWriter
PipedWriter
CharArrayWriter
处理流
BufferedWriter
OutputStreamWriter
printWriter
InputStream
字节读取
节点流
FileInputStream
PipedInputStream
ByteArrayInputStream
处理流
BufferedInputStream
DataInputStream
ObjectInputStream
SequenceInputStream
OutputStream
字节写出
节点流
FileOutputStream
PipedOutputStream
ByteArrayOutputStream
处理流
BufferedOutputStream
DataOutputStream
ObjectOutputStream
PrintStream



既然有了字节流,为什么还要有字符流?

问题本质想问：不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？

回答：字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以， I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

BIO,NIO,AIO 有什么区别?

BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的N可以理解为Non-

blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。 NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

浅拷贝：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝。

深拷贝：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝。