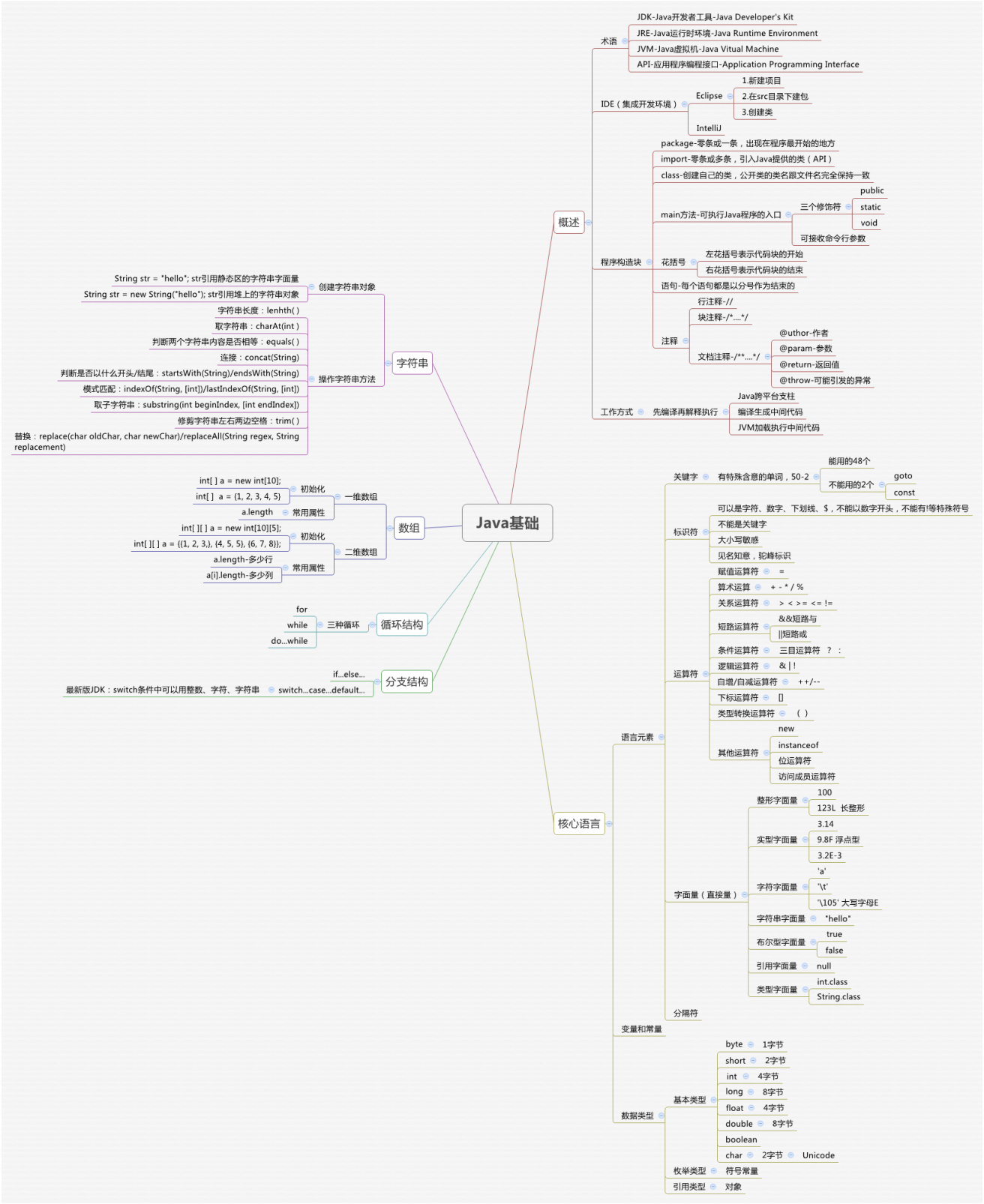
Java 基础 - 图谱 & Q/A

对Java基础知识体系小结，同时结合一些Q&A进行理解。

知识体系



Q&A

Java 中应该使用什么数据类型来代表价格*?*

如果不是特别关心内存和性能的话，使用BigDecimal，否则使用预定义精度的 double 类型。

怎么将 *byte* 转换为 *String?*

可以使用 String 接收 byte[] 参数的构造器来进行转换，需要注意的点是要使用的正确的编码，否则会使用平台默认编码，这个编码可能跟原来的编码相同，也可能不同。

Java 中怎样将 *bytes* 转换为 *long* 类型*?*

String接收bytes的构造器转成String，再Long.parseLong

我们能将 *int* 强制转换为 *byte* 类型的变量吗*?* 如果该值大于 *byte* 类型的范围，将会出现什么现象*?*

是的，我们可以做强制转换，但是 Java 中 int 是 32 位的，而 byte 是 8 位的，所以，如果强制转化是，int 类型的高 24 位将会被丢弃，byte 类型的范围是从 -128 到 127。

存在两个类，*B* 继承 *A*，*C* 继承 *B*，我们能将 *B* 转换为 *C* 么*?* 如 *C = (C) B*；

可以，向下转型。但是不建议使用，容易出现类型转型异常.

哪个类包含 *clone* 方法*?* 是 *Cloneable* 还是 *Object?*

java.lang.Cloneable 是一个标示性接口，不包含任何方法，clone 方法在 object 类中定义。并且需要知道 clone() 方法是一个本地方法，这意味着它是由 c 或 c++ 或 其他本地语言实现的。

Java 中 ++ 操作符是线程安全的吗*?*

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。还会存在竞态条件(读取-修改-写入)。

a = a + b 与 *a += b* 的区别

+= 隐式的将加操作的结果类型强制转换为持有结果的类型。如果两这个整型相加，如 byte、short 或者 int，首先会将它们提升到 int 类型，然后在执行加法操作。

```
byte a = 127;
byte b = 127;
b = a + b; // error : cannot convert from int to byte
b += a; // ok
```

(因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错)

我能在不进行强制转换的情况下将一个 *double* 值赋值给 *long* 类型的变量吗*?*

不行，你不能在没有强制类型转换的前提下将一个 double 值赋值给 long 类型的变量，因为 double 类型的范围比 long 类型更广，所以必须要进行强制转换。

*3*0.1 == 0.3* 将会返回什么*? true* 还是 *false?*

false，因为有些浮点数不能完全精确的表示出来。

int 和 *Integer* 哪个会占用更多的内存*?*

Integer 对象会占用更多的内存。Integer 是一个对象，需要存储对象的元数据。但是 int 是一个原始类型的数据，所以占用的空间更少。

为什么 *Java* 中的 *String* 是不可变的*(Immutable)?*

Java 中的 String 不可变是因为 Java 的设计者认为字符串使用非常频繁，将字符串设置为不可变可以允许多个客户端之间共享相同的字符串。更详细的内容参见答案。

我们能在 *Switch* 中使用 *String* 吗*?*

从 Java 7 开始，我们可以在 switch case 中使用字符串，但这仅仅是一个语法糖。内部实现在 switch 中使用字符串的 hash code。

Java 中的构造器链是什么*?*

当你从一个构造器中调用另一个构造器，就是Java 中的构造器链。这种情况只在重载了类的构造器的时候才会出现。

枚举类

JDK1.5出现 每个枚举值都需要调用一次构造函数

什么是不可变对象*(immutable object)? Java* 中怎么创建一个不可变对象*?*

不可变对象指对象一旦被创建，状态就不能再改变。任何修改都会创建一个新的对象，如 String、Integer及其它包装类。

如何在Java中写出Immutable的类?

要写出这样的类，需要遵循以下几个原则:

1)immutable对象的状态在创建之后就不能发生改变，任何对它的改变都应该产生一个新的对象。

2)Immutable类的所有的属性都应该是final的。

3)对象必须被正确的创建，比如: 对象引用在对象创建过程中不能泄露(leak)。

4)对象应该是final的，以此来限制子类继承父类，以避免子类改变了父类的immutable特性。

5)如果类中包含mutable类对象，那么返回给客户端的时候，返回该对象的一个拷贝，而不是该对象本身(该条可以归为第一条中的一个特例)

我们能创建一个包含可变对象的不可变对象吗*?*

是的，我们是可以创建一个包含可变对象的不可变对象的，你只需要谨慎一点，不要共享可变对象的引用就可以了，如果需要变化时，就返回原对象的一个拷贝。最常见的例子就是对象中包含一个日期对象的引用。

有没有可能两个不相等的对象有有相同的 *hashcode?*

有可能，两个不相等的对象可能会有相同的 hashcode 值，这就是为什么在 hashmap 中会有冲突。相等 hashcode 值的规定只是说如果两个对象相等，必须有相同的hashcode 值，但是没有关于不相等对象的任何规定。

两个相同的对象会有不同的的 *hash code* 吗*?*

不能，根据 hash code 的规定，这是不可能的。

我们可以在 *hashcode()* 中使用随机数字吗*?*

不行，因为对象的 hashcode 值必须是相同的。

Java 中，*Comparator* 与 *Comparable* 有什么不同*?*

Comparable 接口用于定义对象的自然顺序，而 comparator 通常用于定义用户定制的顺序。Comparable 总是只有一个，但是可以有多个 comparator 来定义对象的顺序。

为什么在重写 *equals* 方法的时候需要重写 *hashCode* 方法*?*

因为有强制的规范指定需要同时重写 hashcode 与 equal 是方法，许多容器类，如 HashMap、HashSet 都依赖于 hashcode 与 equals 的规定。

*“a==b”*和*”a.equals(b)”*有什么区别*?*

如果 a 和 b 都是对象，则 a==b 是比较两个对象的引用，只有当 a 和 b 指向的是堆中的同一个对象才会返回 true，而 a.equals(b) 是进行逻辑比较，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

a.hashCode() 有什么用*?* 与 *a.equals(b)* 有什么关系*?*

简介: hashCode() 方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap等等。它与 equals() 方法关系特别紧密。根据 Java 规范，两个使用 equal() 方法来判断相等的对象，必须具有相同的 hash code。

1、hashcode的作用

List和Set，如何保证Set不重复呢? 通过迭代使用equals方法来判断，数据量小还可以接受，数据量大怎么解决? 引入hashcode，实际上hashcode扮演的角色就是寻址，大大减少查询匹配次数。

2、hashcode重要吗

对于数组、List集合就是一个累赘。而对于hashmap, hashset, hashtable就异常重要了。

3、equals方法遵循的原则

- 对称性 若x.equals(y)true，则y.equals(x)true
- 自反性 x.equals(x)必须true

- 传递性 若x.equals(y)true,y.equals(z)true,则x.equals(z)必为true
- 一致性 只要x,y内容不变，无论调用多少次结果不变
- 其他 x.equals(null) 永远false，x.equals(和x数据类型不同)始终false

final、*finalize* 和 *finally* 的不同之处*?*

- final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。
- Java 技术允许使用 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。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的，但是什么时候调用 finalize 没有保证。
- finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

Java 中的编译期常量是什么*?* 使用它又什么风险*?*

变量也就是我们所说的编译期常量，这里的 public 可选的。实际上这些变量在编译时会被替换掉，因为编译器知道这些变量的值，并且知道这些变量在运行时不能改变。这种方式存在的一个问题是你使用了一个内部的或第三方库中的公有编译时常量，但是这个值后面被其他人改变了，但是你的客户端仍然在使用老的值，甚至你已经部署了一个新的jar。为了避免这种情况，当你在更新依赖 JAR 文件时，确保重新编译你的程序。

静态内部类与顶级类有什么区别*?*

一个公共的顶级类的源文件名称与类名相同，而嵌套静态类没有这个要求。一个嵌套类位于顶级类内部，需要使用顶级类的名称来引用嵌套静态类，如 HashMap.Entry 是一个嵌套静态类，HashMap 是一个顶级类，Entry是一个嵌套静态类。

Java 中，*Serializable* 与 *Externalizable* 的区别*?*

Serializable 接口是一个序列化 Java 类的接口，以便于它们可以在网络上传输或者可以将它们的状态保存在磁盘上，是 JVM 内嵌的默认序列化方式，成本高、脆弱而且不安全。Externalizable 允许你控制整个序列化过程，指定特定的二进制格式，增加安全机制。

说出 *JDK 1.7* 中的三个新特性*?*

虽然 JDK 1.7 不像 JDK 5 和 8 一样的大版本，但是，还是有很多新的特性，如 try-with-resource 语句，这样你在使用流或者资源的时候，就不需要手动关闭，Java 会自动关闭。Fork-Join 池某种程度上实现 Java 版的 Map-reduce。允许 Switch 中有 String 变量和文本。菱形操作符(<>)用于泛型推断，不再需要在变量声明的右边申明泛型，因此可以写出可读写更强、更简洁的代码。另一个值得一提的特性是改善异常处理，如允许在同一个 catch 块中捕获多个异常。

说出 *5* 个 *JDK 1.8* 引入的新特性*?*

Java 8 在 Java 历史上是一个开创新的版本，下面 JDK 8 中 5 个主要的特性：Lambda 表达式，允许像对象一样传递匿名函数 Stream API，充分利用现代多核 CPU，可以写出很简洁的代码 Date 与 Time API，最终，有一个稳定、简单的日期和时间库可供你使用 扩展方法，现在，接口中可以有静态、默认方法。 重复注解，现在你可以将相同的注解在同一类型上使用多次。

下述包含 Java 面试过程中关于 SOLID 的设计原则，OOP 基础，如类，对象，接口，继承，多态，封装，抽象以及更高级的一些概念，如组合、聚合及关联。也包含了 GOF 设计模式的问题。

接口是什么*?* 为什么要使用接口而不是直接使用具体类*?*

接口用于定义 API。它定义了类必须得遵循的规则。同时，它提供了一种抽象，因为客户端只使用接口，这样可以有多重实现，如 List 接口，你可以使用可随机访问的 ArrayList，也可以使用方便插入和删除的 LinkedList。接口中不允许普通方法，以此来保证抽象，但是 Java 8 中你可以在接口声明静态方法和默认普通方法。

Java 中，抽象类与接口之间有什么不同*?*

Java 中，抽象类和接口有很多不同之处，但是最重要的一个是 Java 中限制一个类只能继承一个类，但是可以实现多个接口。抽象类可以很好的定义一个家族类的默认行为，而接口能更好的定义类型，有助于后面实现多态机制参见第六条。

*Object*有哪些公用方法*?*

clone equals hashcode wait notify notifyall finalize toString getClass 除了clone和finalize其他均为公共方法。

11个方法，wait被重载了两次

*equals*与==的区别

区别1. ==是一个运算符 equals是Object类的方法

区别2. 比较时的区别

- 用于基本类型的变量比较时: ==用于比较值是否相等，equals不能直接用于基本数据类型的比较，需要转换为其对应的包装类型。
- 用于引用类型的比较时。==和equals都是比较栈内存中的地址是否相等 。相等为true 否则为false。但是通常会重写equals方法去实现对象内容的比较。

String、*StringBuffer*与*StringBuilder*的区别

第一点: 可变和适用范围。String对象是不可变的，而StringBuffer和StringBuilder是可变字符序列。每次对String的操作相当于生成一个新的String对象，而对StringBuffer和StringBuilder的操作是对对象本身的操作，而不会生成新的对象，所以对于频繁改变内容的字符串避免使用String，因为频繁的生成对象将会对系统性能产生影响。

第二点: 线程安全。String由于有final修饰，是immutable的，安全性是简单而纯粹的。StringBuilder和StringBuffer的区别在于StringBuilder不保证同步，也就是说如果需要线程安全需要使用StringBuffer，不需要同步的StringBuilder效率更高。

*switch*能否用*String*做参数

Java1.7开始支持，但实际这是一颗Java语法糖。除此之外，byte，short，int，枚举均可用于switch，而boolean和浮点型不可以。

接口与抽象类

- 一个子类只能继承一个抽象类,但能实现多个接口
- 抽象类可以有构造方法,接口没有构造方法
- 抽象类可以有普通成员变量,接口没有普通成员变量
- 抽象类和接口都可有静态成员变量,抽象类中静态成员变量访问类型任意，接口只能public static final(默认)
- 抽象类可以没有抽象方法,抽象类可以有普通方法,接口中都是抽象方法
- 抽象类可以有静态方法，接口不能有静态方法
- 抽象类中的方法可以是public、protected;接口方法只有public abstract

抽象类和最终类

抽象类可以没有抽象方法, 最终类可以没有最终方法

最终类不能被继承, 最终方法不能被重写(可以重载)

异常

相关的关键字 throw、throws、try...catch、finally

- throws 用在方法签名上, 以便抛出的异常可以被调用者处理
- throw 方法内部通过throw抛出异常
- try 用于检测包住的语句块, 若有异常, catch子句捕获并执行catch块

关于*finally*

- finally不管有没有异常都要处理
- 当try和catch中有return时，finally仍然会执行，finally比return先执行
- 不管有木有异常抛出, finally在return返回前执行
- finally是在return后面的表达式运算后执行的(此时并没有返回运算后的值，而是先把要返回的值保存起来，管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值)，所以函数返回值是在finally执行前确定的

注意: finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值

finally不执行的几种情况: 程序提前终止如调用了System.exit, 病毒，断电

受检查异常和运行时异常

- 受检查的异常(checked exceptions),其必须被try...catch语句块所捕获, 或者在方法签名里通过throws子句声明。受检查的异常必须在编译时被捕捉处理,命名为Checked Exception是因为Java编译器要进行检查, Java虚拟机也要进行检查, 以确保这个规则得到遵守。

常见的checked exception: ClassNotFoundException IOException FileNotFoundException EOFException

- 运行时异常(runtime exceptions), 需要程序员自己分析代码决定是否捕获和处理,比如空指针,被0除...

常见的runtime exception: NullPointerException ArithmeticException ClassCastException IllegalArgumentException IllegalStateException IndexOutOfBoundsException NoSuchElementException

- Error的，则属于严重错误，如系统崩溃、虚拟机错误、动态链接失败等，这些错误无法恢复或者不可能捕捉，将导致应用程序中断，Error不需要捕获。

*super*出现在父类的子类中。有三种存在方式

- super.xxx(xxx为变量名或对象名)意思是获取父类中xxx的变量或引用
- super.xxx(); (xxx为方法名)意思是直接访问并调用父类中的方法
- super() 调用父类构造

注: super只能指代其直接父类

*this() & super()*在构造方法中的区别

- 调用super()必须写在子类构造方法的第一行, 否则编译不通过
- super从子类调用父类构造, this在同一类中调用其他构造均需要放在第一行
- 尽管可以用this调用一个构造器, 却不能调用2个
- this和super不能出现在同一个构造器中, 否则编译不通过
- this()、super()都指的对象,不可以在static环境中使用
- 本质this指向本对象的指针。super是一个关键字

构造内部类和静态内部类对象

```
public class Enclosingone {
    public class Insideone {}
    public static class Insideone{}
}

public class Test {
    public static void main(String[] args) {
    // 构造内部类对象需要外部类的引用
    Enclosingone.Insideone obj1 = new Enclosingone().new Insideone();
    // 构造静态内部类的对象
    Enclosingone.Insideone obj2 = new Enclosingone.Insideone();
    }
}
```

静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。非静态内部类能够访问外部类的静态和非静态成员。静态内部类不能访问外部类的非静态成员，只能访问外部类的静态成员。

序列化

声明为static和transient类型的数据不能被序列化， 反序列化需要一个无参构造函数

*Java*移位运算符

java中有三种移位运算符

- << :左移运算符,x << 1,相当于x乘以2(不溢出的情况下),低位补0
- >> :带符号右移,x >> 1,相当于x除以2,正数高位补0,负数高位补1
- >>> :无符号右移,忽略符号位,空位都以0补齐

形参&实参

形式参数可被视为local variable.形参和局部变量一样都不能离开方法。只有在方法中使用，不会在方法外可见。 形式参数只能用final修饰符，其它任何修饰符都会引起编译器错误。但是用这个修饰符也有一定的限制，就是在方法中不能对参数做任何修改。不过一般情况下，一个方法的形参不用final修饰。只有在特殊情况下，那就是: 方法内部类。一个方法内的内部类如果使用了这个方法的参数或者局部变量的话，这个参数或局部变量应该是final。 形参的值在调用时根据调用者更改，实参则用自身的值更改形参的值(指针、引用皆在此列)，也就是说真正被传递的是实参。

局部变量为什么要初始化

局部变量是指类方法中的变量，必须初始化。局部变量运行时被分配在栈中，量大，生命周期短，如果虚拟机给每个局部变量都初始化一下，是一笔很大的开销，但变量不初始化为默认值就使用是不安全的。出于速度和安全性两个方面的综合考虑，解决方案就是虚拟机不初始化，但要求编写者一定要在使用前给变量赋值。

*Java*语言的鲁棒性

Java在编译和运行程序时，都要对可能出现的问题进行检查，以消除错误的产生。它提供自动垃圾收集来进行内存管理，防止程序员在管理内存时容易产生的错误。通过集成的面向对象的例外处理机制，在编译时，Java揭示出可能出现但未被处理的异常，帮助程序员正确地进行选择以防止系统的崩溃。另外，Java在编译时还可捕获类型声明中的许多常见错误，防止动态运行时不匹配问题的出现。