第六章 接口、lambda 表达式与内部类

6.1 接口

6.1.1 接口概念

- 接口（interface），主要用来描述**类**具有什么功能，而并不给出每个功能的具体实现。一个类可以实现一个或多个接口，并在需要接口的地方， 随时使用实现了相应接口的对象。
- 接口不是类，而是对类的一组需求描述，这些类要遵从接口描述的统一格式进行定义。
- 接口中的所有方法自动地属于`public`。因此， 在接口中声明方法时， 不必提供关键字`public`
- 有些接口可能包含多个方法, 接口中还可以定义常量，接口绝不能含有实例域。提供实例域和方法实现的任务应该由实现接口的那个类来完成。

为了让类实现一个接口， 通常需要下面两个步骤：

- 1) 将类声明为实现给定的接口。
- 2) 对接口中的所有方法进行定义。

6.1.2 接口的特性

- 接口不是类，尤其不能使用new 运算符实例化一个接口
- 尽管不能构造接口的对象，却能声明接口的变量
- 接口变量必须引用实现了接口的类对象
- 与可以建立类的继承关系一样，接口也可以被扩展
- 在接口中不能包含实例域或静态方法，但却可以包含常量
- 与接口中的方法都自动地被设置为`public`一样，接口中的域将被自动设为`public static final`。
- 有些接口只定义了常量， 而没有定义方法。
- 尽管每个类只能够拥有一个超类， 但却可以实现多个接口。

6.1.3 接口与抽象类

使用抽象类表示通用属性存在这样一个问题： 每个类只能扩展于一个类。每个类可以实现多个接口。有些程序设计语言允许一个类有多个超类， 例如`C++`。我们将此特性称为多重继承(multiple inheritance)。 而`Java`的设计者选择了不支持多继承，其主要原因是多继承会让语言本身变得非常复杂（如同`C++`），效率也会降低（如同`Eiffel`）。实际上， 接口可以提供多重继承的大多数好处，同时还能避免多重继承的复杂性和低效性。

6.1.4 静态方法

在`Java SE 8`中，允许在接口中增加静态方法。 通常的做法都是将静态方法放在伴随类中。

6.1.5 默认方法

可以为接口方法提供一个默认实现。必须用`default`修饰符标记这样一个方法。
默认方法的一个重要用法是“ 接口演化” （interface evolution)。

6.1.6 解决默认方法冲突

如果先在一个接口中将一个方法定义为默认方法， 然后又在超类或另一个接口中定义了同样的方法，`Java`规则：

- 1) 超类优先。如果超类提供了一个具体方法， 同名而且有相同参数类型的默认方法会被忽略。

- 2) 接口冲突。如果一个超接口提供了一个默认方法， 另一个接口提供了一个同名而且参数类型（不论是否是默认参数）相同的方法， 必须覆盖这个方法来解决冲突。

一个类扩展了一个超类，同时实现了一个接口，并从超类和接口继承了相同的方法。在这种情况下， 只会考虑超类方法， 接口的所有默认方法都会被忽略。

6.2 接口示例

6.2.1 接口与回调

6.2.2 comparator接口

6.2.3 对象克隆

6.3 lambda表达式

如何使用 `lambda` 表达式采用一种简洁的语法定义代码块， 以及如何编写处理 `lambda` 表达式的代码。

6.3.1 为什么引入lambda 表达式

`lambda` 表达式是一个可传递的代码块， 可以在以后执行一次或多次。在Java 中传递一个代码段并不容易， 不能直接传递代码段。 `Java` 是一种面向对象语言， 所以必须构造一个对象，这个对象的类需要有一个方法能包含所需的代码.

6.3.2 lambda表达式语法

`lambda` 表达式就是一个代码块， 以及必须传入代码的变量规范。
`lambda` 表达式形式：参数， 箭头（->) 以及一个表达式。如果代码要完成的计算无法放在一个表达式中，就可以像写方法一样，把这些代码放在{}中，并包含显式的 `return` 语句.

6.3.3 函数式接口

对于只有一个抽象方法的接口， 需要这种接口的对象时， 就可以提供一个 `lambda` 表达式。这种接口称为函数式接口（ functional interface)。在 `Java` 中， 对 `lambda` 表达式所能做的也只是能转换为函数式接口。

6.3.4 方法引用

6.3.5 构造器引用

构造器引用与方法引用很类似，只不过方法名为 `new`。

6.3.6 变量作用域

6.3.7 处理lambda表达式

6.3.8 comparator延伸

`Comparator` 接口包含很多方便的静态方法来创建比较器。这些方法可以用于 `lambda` 表达式或方法引用。 静态 `comparing` 方法取一个“ 键提取器” 函数， 它将类型T 映射为一个可比较的类型(如String)。对要比较的对象应用这个函数， 然后对返回的键完成比较。

6.4 内部类

内部类（inner class） ： 定义在另一个类中的类。
使用内部类的原因：

- 内部类方法可以访问该类定义所在的作用域中的数据，包括私有的数据。
- 内部类可以对同一个包中的其他类隐藏起来。
- 当想要定义一个回调函数且不想编写大量代码时，使用匿名（anonymous) 内部类比较便捷。

6.4.1 使用内部类访问对象状态

6.4.2 内部类的特殊语法规则

内部类中声明的所有静态域都必须是`final`。一个静态域只有一个实例， 不过对于每个外部对象， 会分别有一个单独的内部类实例。如果这个域不是`final`, 它可能就不是唯一的。
内部类不能有`static`方法。

6.4.3 内部类是否有用、必要和安全

6.4.4 局部内部类

局部类不能用`public`或`private`访问说明符进行声明。它的作用域被限定在声明这个局部类的块中。局部类有一个优势， 即对外部世界可以完全地隐藏起来.

6.4.5 由外部方法访问变量

与其他内部类相比较， 局部类还有一个优点。它们不仅能够访问包含它们的外部类， 还可以访问局部变量。不过， 那些局部变量必须事实上为`final`。

6.4.6 匿名内部类

将局部内部类的使用再深人一步。假如只创建这个类的一个对象，就不必命名了, 这种类被称为匿名内部类（anonymous inner class)。

6.4.7 静态内部类

有时候， 使用内部类只是为了把一个类隐藏在另外一个类的内部，并不需要内部类引用外围类对象。为此，可以将内部类声明为`static`, 以便取消产生的引用。

6.5 代理

6.5.1 何时使用代理

6.5.2 创建代理对象

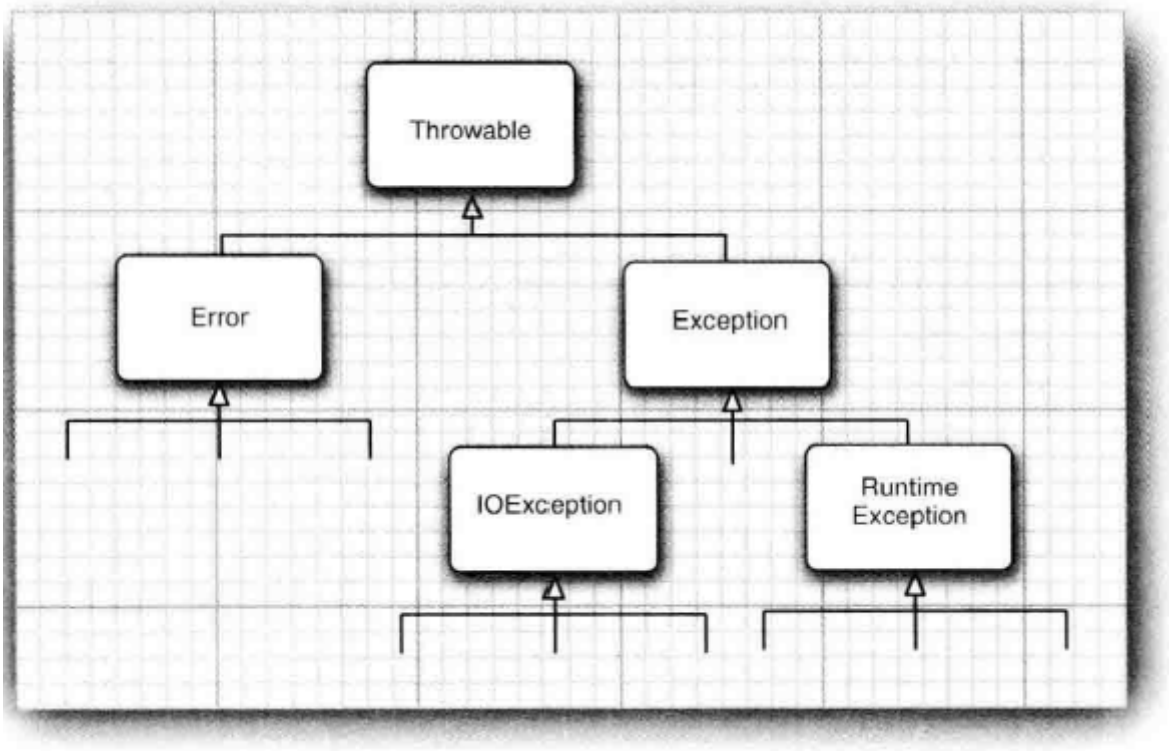
6.5.3 代理类的特性

第七章： 异常、断言和日志

7.1 处理错误

7.1.1 异常分类

异常对象都是派生于`Throwable`类的一个实例。如果`Java`中内置的异常类不能够满足需求，用户可以创建自己的异常类。
`java`中的异常层次结构：



7.1.2 声明受查异常

在自己编写方法时， 不必将所有可能抛出的异常都进行声明。至于什么时候需要在方法中用`throws`子句声明异常， 什么异常必须使用`throws`子句声明， 需要记住在遇到下面4 种
情况时应该抛出异常：

- 1) 调用一个抛出受查异常的方法， 例如， FileInputStream 构造器。
- 2) 程序运行过程中发现错误， 并且利用throw 语句抛出一个受查异常（下一节将详细地介绍throw 语句)。
- 3) 程序出现错误， 例如，a[-l]=0 会抛出一个ArrayIndexOutOffloundsException 这样的非受查异常。
- 4) Java 虚拟机和运行时库出现的内部错误。

7.1.3 如何抛出异常

首先要决定应该抛出什么类型的异常。
在前面已经看到， 对于一个已经存在的异常类， 将其抛出非常容易,在这种情况下：

- 1) 找到一个合适的异常类。
- 2) 创建这个类的一个对象。
- 3) 将对象抛出。
  一旦方法抛出了异常， 这个方法就不可能返回到调用者。也就是说， 不必为返回的默认值或错误代码担忧。

7.1.4 创建异常类

在程序中， 可能会遇到任何标准异常类都没有能够充分地描述清楚的问题. 定义一个派生于`Exception`的类， 或者派生于`Exception`子类的类。 习惯上， 定义的类应该包含两个构造器， 一个是默认的构造器；另一个是带有详细描述信息的构造器.

7.2 捕获异常

7.2.1 捕获异常

如果某个异常发生的时候没有在任何地方进行捕获，那程序就会终止执行，并在控制台上打印出异常信息， 其中包括异常的类型和堆栈的内容。

7.2.2 捕获多个异常

在一个`try`语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理。

7.2.3 再次抛出异常与异常链

7.2.4 `finally`子句

当代码抛出一个异常时， 就会终止方法中剩余代码的处理，并退出这个方法的执行。如果方法获得了一些本地资源，并且只有这个方法自己知道，又如果这些资源在退出方法之前必须被回收，那么就会产生资源回收问题。一种解决方案是捕获并重新抛出所有的异常。但是，这种解决方案比较乏味，这是因为需要在两个地方清除所分配的资源。一个在正常的代码中；另一个在异常代码中。
`Java`有一种更好的解决方案， 这就是`finally`子句.

7.2.5 带资源的`try`语句

7.2.6 分析堆栈轨迹元素

堆栈轨迹（ stack trace) 是一个方法调用过程的列表， 它包含了程序执行过程中方法调用的特定位置前面已经看到过这种列表， 当`Java`程序正常终止， 而没有捕获异常时， 这个列表就会显示出来。
可以调用`Throwable`类的`printStackTrace`方法访问堆栈轨迹的文本描述信息

7.3 使用异常机制的技巧

- 1. 异常处理不能代替简单的测试
- 2. 不要过分地细化异常
- 3. 利用异常层次结构
- 4. 不要压制异常
- 5. 在检测错误时，“ 苛刻” 要比放任更好
- 6. 不要羞于传递异常

7.4 使用断言

7.4.1 断言的概念

断言机制允许在测试期间向代码中插入一些检查语句。当代码发布时，这些插人的检测语句将会被自动地移走。

7.4.2 启用和禁用断言

在默认情况下， 断言被禁用。可以在运行程序时用`-enableassertions`或`-ea`选项启用
在启用或禁用断言时不必重新编译程序。启用或禁用断言是类加载器(class loader) 的功能。当断言被禁用时, 类加载器将跳过断言代码， 因此，不会降低程序运行的速度。

7.4.3 使用断言完成参数检查

7.4.4 为文档假设使用断言

7.5 记录日志

7.5.1 基本日志

要生成简单的日志记录，可以使用全局日志记录器（global logger) 并调用其`info`方法

7.5.2 高级日志

7.5.3 修改日志管理器配置

可以通过编辑配置文件来修改日志系统的各种属性

7.5.4 本地化

本地化的应用程序包含资源包（ resource bundle) 中的本地特定信息。资源包由各个地区
(如美国或德国）的映射集合组成。

7.5.5 处理器

在默认情况下t 日志记录器将记录发送到`ConsoleHandler` 中， 并由它输出到`System.err`
流中。特别是， 日志记录器还会将记录发送到父处理器中，而最终的处理器（命名为“ ”）有一个
`ConsoleHandler`。
与日志记录器一样， 处理器也有日志记录级别。对于一个要被记录的日志记录，它的日志记录级别必须
高于日志记录器和处理器的阈值。
在默认情况下， 日志记录器将记录发送到自己的处理器和父处理器。

7.5.6 过滤器

7.5.7 格式化器

7.5.6 日志记录说明

7.6 调试技巧

建议：

- 1) 可以打印或记录变量的值： `System.out.println("x=" + x);`
- 2) 一个不太为人所知但却非常有效的技巧是在每一个类中放置一个单独的main 方法。
  这样就可以对每一个类进行单元测试。
- 3) 如果喜欢使用前面所讲述的技巧， 就应该到http://junit.org 网站上查看一下`JUnit`。
  `JUiiit`是一个非常常见的单元测试框架， 利用它可以很容易地组织测试用例套件。只要修改
  类，就需要运行测试。在发现`bug`时，还要补充一些其他的测试用例。
- 4) 日志代理（ logging proxy) 是一个子类的对象， 它可以截获方法调用， 并进行日志记录， 然后
  调用超类中的方法。
- 5) 利用`Throwable`类提供的`printStackTmce` 方法，可以从任何一个异常对象中获得堆栈情况
- 6. 一般来说， 堆栈轨迹显示在`System.err`上。也可以利用`printStackTrace(PrintWriter s)`方法将它发送到一个文件中.
- 7) 通常， 将一个程序中的错误信息保存在一个文件中是非常有用的。然而，错误信息被发送到
  `System.err` 中，而不是`System.out`中。
- 8) 让非捕获异常的堆栈轨迹出现在`System.err`中并不是一个很理想的方法。比较好的方
  式是将这些内容记录到一个文件中。可以调用静态的
  `Thread.setDefaultUncaughtExceptionHandler`方法改变非捕获异常的处理器
- 9) 要想观察类的加载过程， 可以用`-verbose` 标志启动`Java` 虚拟机。
- 10) `-Xlint`选项告诉编译器对一些普遍容易出现的代码问题进行检查。
- 11) `java` 虚拟机增加了对`Java` 应用程序进行监控（monitoring) 和管理（management) 的支
  持。它允许利用虚拟机中的代理装置跟踪内存消耗、线程使用、类加载等情况。

- 12) 可以使用 `jmap` 实用工具获得一个堆的转储， 其中显示了堆中的每个对象
- 13) 如果使用 `-Xprof` 标志运行 `Java` 虚拟机， 就会运行一个基本的剖析器来跟踪那些代码中经常被调用的方法。剖析信息将发送给 `System.out` 。输出结果中还会显示哪些方法是由即时编译器编译的。