Java软件开发

第一部分：基础知识

Java开发环境安装与配置

安装JDK

- 从Oracel Java官网下载JDK7
- 傻瓜式安装

环境配置

- 什么是环境？为什么需要配置环境？
- “我的电脑”右击->属性->高级->环境变量->系统变量

- 新建JAVA_HOME变量，值为jdk的安装目录，默认为：C:\Program Files\Java\jdk1.7.0_03

- 修改Path的值，添加%JAVA_HOME%/bin;%JAVA_HOME%/jre/bin，使得系统可以在任何路径下识别java命令

- 新建CLASS_PATH，值为.;%JAVA_HOME%/lib/dt.jar;%JAVA_HOME%/lib/tools.jar ，CLASSPATH为java加载类(class or lib)路径，只有类在classpath中，java命令才能识别。

测试

- 打开cmd，输入javac,若输出一些命令选项，或输入java -version，若输出jdk版本号，则表明安装成功。

开发工具

- 可直接在文本编辑器(挑选一个定制性强，有代码高亮的哦，比如vim, Notepad++, sublime text)，然后cmd中输入javac filename.java编译源码，若没有错误信息输出则输入java filename（没有.class的哦）运行

- 选择一款优秀的IDE，推荐**eclipse**

Java “Hello World” 之旅

"Hello World"

- 包结构
- Java程序全是类(class)
- 每个文件只允许有一个public class，并且类名与文件名必须相同
- 编码风格：类名-"大驼峰"命名方式，方法名-"小驼峰"命名方式(除了main)
- 默认导入java.lang.*
- 加载寻找main方法

Hello, Date

- 若需要java.lang.*之外的类库，就需要显示import导入

- 字符串 + 对象 = 字符串 + 对象.toString()

编译执行过程

- javac 将源码编译成中间字节码
- java 由jvm来执行中间字节码

- 由于最终程序是在虚拟机上执行，所以java是完全跨平台的，“一次编译，到处运行”，甚至连基本类型变量所占内存空间在各个平台上都是一致的。

基本语法

操作符

- 赋值 =
- 算术操作符 +， -， *， /
- 自增自减 ++，--
- 关系操作符 <，>，<=，>=，==，!=(等于和不等于适用于所有的基本数据类型，而其他比较符不适用于boolean类型)
- 逻辑操作符 &&，||，!
- 按位操作符 &(按位与)，|(按位或)，^(按位异或)，~(按位非)

操作符

- 移位操作符：只可用来处理整数类型。左移位<<，右移位>>
- 三元操作符 ？ ： ,相当于if-else结构
- 字符串操作符 +，+= 。（系统做的操作符重载）

练习1：编写程序练习上述操作符

控制流程

- 条件选择：if-else结构(else可选)，switch(选择因子必须是int或char那样的整数值)
- 迭代：while, do-while , for
- Java新增foreach控制结构

  示例control/ForeachInt.java

控制流程

- **return** ： 一方面指定一个方法返回什么值，另一方面它会导致当前的方法退出，并返回那个值。
- 如果在返回void的方法中没有return语句，那么在该方法的结尾处会有一个隐式的return，因此在方法中并非必须要有一个return语句

控制流程

- break: 强制退出循环，不执行循环中的剩余语句。
- continue: 停止执行当前的迭代，然后退回起始处，开始下一次迭代。
- 示例：control/BreakAndContinue.java

面向对象技术

什么是面向对象？

- **面向过程 VS. 面向对象**
- **面向过程**是按照解决一个问题的先后步骤流程来组织代码，比较直观。
- **面向对象**相对面向过程，更加抽象，认为任何一件事情都可能看做是事物与事物之间的交互。
- 任何事物都可以进行归类，并且归类又是有层次的

面向对象的三大特征

- **封装**：把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

- **继承**：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。
  通过继承创建的新类称为 “子类” 或 “派生类” 。
  被继承的类称为 “基类” 、 “父类” 或 “超类” 。
  继承的过程，就是从一般到特殊的过程。

- **多态**：允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。

为什么需要面向对象？

- “计算机编程的本质就是尽量控制(降低)程序的复杂度” --- Brian Kernighan

- 随着软件越来越大型，内含逻辑越来越复杂，面向过程语言组织程序的方式以无法将程序的复杂度控制在适当的范围之类，从而极大地影响了生产效率以及软件的健壮性。
- 所以需要新的编程范式，面向对象就是随着大型软件的产生而被提出来的新编程范式之一。

类与对象

- **类**可直观地将其对应于现实世界中的**一类**东西，是一种抽象的概念，比如中国公民是一个抽象的名词，是一类人，是具有中国国籍的每一个人所属的类。

- 类也就是类型，一种类型可以用属性和行为的结合来定义

- **对象**则是一个个具体的活生生的中国公民，是对中国公民这一抽象概念的具体化。
- 对象是类的一个实例化。

示例：ObjectOriented/ChinesePeopleSample.java

初始化与构造器

- 构造器：在实例化对象时自动被系统调用，用来初始化对象的属性，或者做一些其他的预处理工作。名称与类名相同。一个类可以有多个构造器（方法重载），根据实例化对象时提供的参数列表来选择哪个构造器。
- 如果类定义中未提供构造函数，那么实例化对象时系统会自动使用一个默认构造函数。
- 构造器是一种特殊类型的方法，因为它没有返回值
- 示例：ObjectOriented/Constructor.java

static关键字

- static修改的属性或方法都是类的属性或方法。
- 无论创建多少个对象，静态数据都只占用一份存储区域。
- static关键字不能作用于局部变量。

- 如果静态数据是基本类型的，且没有对它进行显式初始化，那么它就会获得基本类型的标准初值；如果它是一个对象引用，那么它的默认初始化值就是null。
- 为什么需要static关键字？方便对象之间共享信息。
- 示例：ObjectOriented/StaticSample.java

代码复用

- 面向过程语言复用的单元是函数
- 面向对象语言复用的单元是类(类是属性与方法的结合)。
- 面向对象类复用的基本方式有：继承，组合以及代理

练习：编写一个Student类，并实例化该类

继承

- 进一步加强了代码复用的能力。
- 是一个逐步具体化的过程。
- 关键字 extends
- 若没有显示继承，则类隐式继承自基类Object

```
class supClass {
    ...
}
class SubClass extends SupClass {
   ...
}
```

示例: ObjectOriented/Cartoon.java

子类实例化对象的过程

- 当系统实例化某个类时，如果发现它继承自另一个类，就会先实例化父类，调用父类构造器，初始化父类属性，然后再实例化子类。 Python不是这样的

- 示例：
  ObjectOriented/Cartoon.java

代理

- Java并没有对代理提供直接支持，这是继承与组合之间的中庸之道，因为我们要将一个成员对象置于所要构造的类中（就像组合），但与此同时我们在新类中暴露了该成员对象的所有方法（就像继承）。
- 示例：ObjectOriented/SpaceShipDelegation.java

多态(动态绑定，后期绑定，运行时绑定)

- 多态通过分离做什么和怎么做，消除类型之间的耦合，不但能够改善代码的组织结构和可读性，还能够创建可扩展的程序---即无论在项目最初创建时还是在需要添加新功能时都可以“生长”的程序。

- 多态方法调用允许一种类型表现出与其他相似类型之间的区别，只要它们都是从同一基类导出而来的。这种区别是根据方法行为的不同而表示出来的。

多态

- 示例：ObjectOriented/DuoTai.java
- 方法调用绑定：将一个方法调用同一个方法主体关联起来称作绑定。若在程序执行前进行绑定（如果有的话，由编译器和连接程序实现），叫做前期绑定。
- 后期绑定：在运行时根据对象的类型进行绑定。如果一种语言想实现后期绑定，就必须具有某种机制，以便在运行时能判断对象的类型，从而调用恰当的方法。也就是说，编译器一直不知道对象的类型，但是方法调用机制能找到正确的方法体，并加以调用。

多态

- Java中除了static方法和final方法（private方法属于final方法），之外，其他所有的方法都是后期绑定。
- 为什么要将某个方法声明为final呢？可以防止其他人覆盖该方法。但更重要的一点或许是：这样做可以有效地"关闭"动态绑定。这样编译器就可以为final方法调用生成更有效的代码。

抽象类

- 如果某个类仅用来被其他类继承，且不允许实例化，那么就可以设计为抽象类。

- 如果一个方法仅有声明而没有方法体，那么就称为抽象方法。包含抽象方法的类叫做抽象类。如果一个类包含一个或多个抽象方法，该类必须被限定为抽象的。

- 如果从一个抽象类继承，并想创建该新类的对象，那么就必须在新类中实现基类中的所有抽象方法。如果不这样做(可以选择不做)，那么导出类便也是抽象类。
- 示例：ObjectOriented/AbstractClass.java

接口

- interface关键字使抽象的概念更向前迈了一步。abstract关键字允许人们在类中创建一个或多个没有任何定义的方法，但是没有提供相应的具体实现，这些实现由此类的继承者创建。
- interface这个关键字声明一个完全抽象的类，它根本没有任何具体实现。
- 接口只提供了形式，而未提供任何具体实现。

接口

- 一个接口表示“所有实现了该特定接口的类看起来都像这样”。
- 要让一个类遵循某个特定接口(或者是一组接口)，需要使用implements关键字，它表示：“interface只是它的外貌，但是现在我要声明它是如何工作的。”
- 示例：ObjectOriented/InterfaceSample.java

接口

- 可以选择在接口中显式地将方法声明为public的，但即使不这么做，它们也是public的。因此，当要实现一个接口时，在接口中被定义的方法必须被定义为是public的；否则，它们将只能得到默认的包访问权限，这样在方法被继承的过程中，其可访问权限就降低了，这是Java编译器所不允许的。

接口与多重继承

- 一个类可以实现多个接口

练习：某个人能吹各种乐器

参考代码：
ObjectOriented/PlayInstruments.java

访问权限控制

为什么需要控制代码的访问权限？

- 访问控制(或隐藏具体实现)与"最初的实现并不恰当"有关
- 在你想要改进的代码时，通常总是会有一些消费者(即客户端程序员)需要你的代码在某些方面保持不变，由此产生了一个面向对象设计中需要考虑的基本问题：“如何把变动的事物与保持不变的事物区分开来?”

为什么需要控制代码的访问权限？

- 为了解决这个问题，Java提供了访问权限修饰词，以供类库开发人员向客户端程序员指明哪些是可用的，哪些是不可用的。

- 访问权限控制的等级，从最大权限到最小权限依次为：public, protected, 包访问权限(没有关键字，默认)和private。

包访问权限

- 包访问权限意味着当前的包中的所有其他类对那个成员都有访问权限，但对于这个包之外的所有类，这个成员却是private的。
- 由于一个编译单元(即一个文件)，只能隶属一个包，所以经由包访问权限，处于同一个编译单元中的所有类彼此之间都是自动可访问的。

- 示例：
  AccessControl/packageAccessCaller.java
  AccessControl/packageAccessCallee.java

public

- 使用关键字public，就意味着public之后紧跟着的成员声明自己对每个人都是可用的，尤其是使用类库的客户端程序员更是如此。

- 示例(练习)：AccessControl/PublicAccess.java

private

- 关键字private的意思是，除了包含该成员的类之外，其他任何类都无法访问这个成员。

protected

- 继承访问权限
- 如果创建了一个新包，并自另一个包中继承类，那么唯一可以访问的成员就是源包的public成员。
- 如果在同一个包内执行继承工作，就可以操纵所有的拥有包访问权限的成员。
- 有时，基类的创建者会希望有某个特定成员，把对它的访问权限赋予派生类而不是所有类。这就需要protected来实现。
- protected也提供包访问权限，所以，相同包内的其他类可以访问protected成员。

封装

- 访问权限的控制常称为是具体实现的隐藏。把数据和方法包装进类中，以及具体实现的隐藏，常共同被称作是**封装**
- 其结果是一个同时带有特征和行为的数据类型。

巧妙的内部类

概念

- **可以将一个类的定义放在另一个类的定义内部**，这就是内部类。

- 内部类是一种非常有用的特性，因为它允许你把一些逻辑相关的类组织在一起，并控制位于内部的类的可视性。

- 示例：`innerClasses/Parcel1.java`

典型情况

- 外部类将有一个方法，该方法返回一个指向内部类的引用。
- 示例：innerClasses/Parcel2.java

链接到外部类

- 到目前为止，内部类似乎还只是一种名字隐藏和组织代码的模式。这很有用，但还有其他用途：当生成一个内部类的对象时，此对象与制造它的外围对象(enclosing object)之间就有了联系，所以它能访问其外围对象的所有成员，而不需要任何特殊条件。此外，内部类还拥有外围类的所有元素访问权。
- 示例： innerClasses/Sequence.java

使用.this和.new

- 如果你需要生成对外部类对象的引用，可以使用外部类的名字后面紧跟着圆点和this。这样产生的引用自动地具有正确的类型。
- 示例：innerClasses/AboutThis.java

使用.this和.new

- 有时你可能想要告知某些其他对象，去创建其某个内部类的对象。要实现此目的，必须在new表达式中提供对其他外部类对象的引用，需要使用.new语法。
- 示例：innerClasses/AboutNew.java

匿名内部类

- 先来看个示例：innerClasses/Parcel3.java

- 这种奇怪的语法是指"创建一个继承自Contents的匿名类对象。"通过new表达式返回的引用被自动向上转型为对Contents的引用。

闭包与回调

通过异常处理错误

基本理念

- “结构不佳的代码不能运行”
- 发现错误的理想时机是在编译阶段，然而，编译期间并不能找出所有的错误，余下的问题必须在运行期间解决。这就需要错误源能通过某种方式，把适当的信息传递给某个接收者---该接收者将知道如何正确处理这个问题。
- 异常处理的目的：以少量的附加代码来确保大型程序的可靠性。
- 好的应用程序中没有未处理的错误。

C以及早期语言

- 返回某个特殊值或者设置某个标志；
- 并且假定接收者将对这个返回值或标志进行检查，以判定是否发生了错误。

异常处理的程序结构

```
try {
  // 可能产生异常的代码
} catch(Type1 id1) {
  // 处理异常类型1的代码
} catch(Type2 id2) {
  // 处理异常类型2的代码
} catch(Type3 id3) {
  // ...
}
...
```

捕获异常

- 使用try块去监控可能产生异常的代码段，后面跟着catch块来处理这些异常。
- 如果在方法内部抛出了异常(或者方法内部调用的其他方法抛出了异常)，这个方法将在抛出异常的过程中结束。如果不希望方法就此结束，那就去捕获异常。
- 异常处理：针对每种要捕获的异常，得准备相应的处理程序，所以一个try块后面一般会跟着多个catch块。

终止与恢复

- 异常处理理论上有两种基本模型：终止模型与恢复模型
- Java支持终止模型 --- 假设错误非常关键，以至于程序无法返回到异常发生的地方继续执行。一旦异常被抛出，就表明错误已无法挽回，也不能回来继续执行。

可能发生异常的代码

- 访问资源：访问硬盘(例如打开文件)，获取内存，网络连接(比如发送Email)等等。
- 谨慎起见，应该在所有访问资源的代码段外部加上try...catch结构

创建自定义异常

- Java提供的异常体系不可能预见所有希望加以报告的错误，所以很多时候需要程序员定义异常类来表示程序中可能遇到的特定问题。
- 要自己定义异常类，必须从已有的异常类继承，最好是选择意思相近的异常类继承。
- 实例程序：
  `exceptions/InheritingExceptions.java`

异常记录日志

- 可以使用java.util.logging工具将输出记录到日志中。
- 示例：exceptions/LoggingExceptions.java

捕获所有的异常

- 可以只写一个异常处理程序来捕获所有类型的异常。最好放在处理程序列表的末尾，以防它抢在其他处理程序之前先捕获了异常。

```
try {
   ...
} catch(Exception e) {
   ...
}
```

重新抛出异常

- 有时希望把刚捕获的异常重新抛出，尤其是在使用Exception捕获所有异常的时候。

```
catch(Exception e) {
   System.out.println("A exception was
    thrown");
   throw e;
}
```

finally子句

- 对于一些代码，可能会希望无论try块中的异常是否抛出，它们都能得到执行。为了达到这个效果，可以在异常处理程序后面加上finally子句。
- 示例程序：exceptions/FinallyWorks.java
- 完整的异常处理程序：

```
try {
    ...
}catch(A a1) {
    ...
}catch(B b1){
    ...
}finally {
    // 每次都会执行的操作
}
```

finally的作用

- 对于没有垃圾回收和析构函数自动调用机制的语言来说，finally非常重要。它能使程序员保证：无论try块里发生了什么，内存总能得到释放。但Java有垃圾回收机制，那么finally子句的作用就是把除内存之外的资源恢复到它们的初始状态，这种需要清理的资源包括：已经打开的文件或网络连接，在屏幕上画的图形等等。

异常匹配

- 当有多个catch子句时，异常处理系统会按照代码书写的先后顺序找出相匹配 “最近”的那个处理程序。找到匹配的处理程序之后，就认为异常已得到处理，然后就不再继续查找，这也就是为什么要把`catch(Exception e){}`子句放在末尾了。

JAVA I/O

文件读写

网络编程