

Java 编程入门手册

0.1 版本 2018-09-01

章铁飞 浙江工商大学计算机学院
tfzhang@zjgsu.edu.cn

简介

本讲义是面向一定编程基础（学过 C 语言）的计算机软件相关专业本科生的 Java 入门教材。主要特点是弱化语法知识：没有面面俱到地介绍语法，只关注最基本、重要、常用的 Java 概念。理由有二。其一，Java 的知识点太多，很多不重要的概念，只是偶尔用到，只要临场去查阅官方文档即可，没必要讲授；其二，人的认知能力有限，学生的胃口不大，与其不分糟粕地把学生灌饱吃撑，不如挑拣精华让学生慢慢品味消化。

本讲义的另外一个特点是强调动手实践能力。很多学过 Java 编程语言的人，知识点了解的不少，但一个几百行代码的小程序却写不下来。为什么？主要原因不在学生，而在教学。传统入门教材中，几乎所有案例都是几十行的小程序，从来都不告诉你稍有规模（几百上千行）的程序应该如何开发？我们都知道，小学生的造句和高中生的八百字文章虽然都是中文，但完全是两码事，你造句再厉害，也不代表会写文章。学语文，最终的一大目的不就是为了写作么？没有人会满足于造句吧？同样地，没有人会满足于几十行的示例程序吧，学编程语言的最终目的不就是开发有用的程序（代码量都是上规模的）么？因此本讲义将介绍基本的实操知识，比如采用什么框架看待游戏程序，如何组织大量源代码文件，采用什么设计模式，如何分解程序开发过程等等。与语法知识不同，实操知识不仅靠教师讲授，更要依赖学生自身的实践，教师点到位，学生做到位，缺一不可。

本讲义主要包含两部分内容：Java 开发准备工作和 Java 基本内容。后者分为 9 个章节，其中 1~6 是基本概念的介绍，7~9 介绍基本的开发方法。笔者水平有限，错误在所难免，欢迎读者来信告知 tfzhang@zjgsu.edu.cn。

第一部分 Java 开发准备工作

1. 开发环境的设置
2. 如何查阅 **jdk** 官方文档
3. 如何查看 **java** 源代码
4. 如何做实证？

1. 开发环境的设置

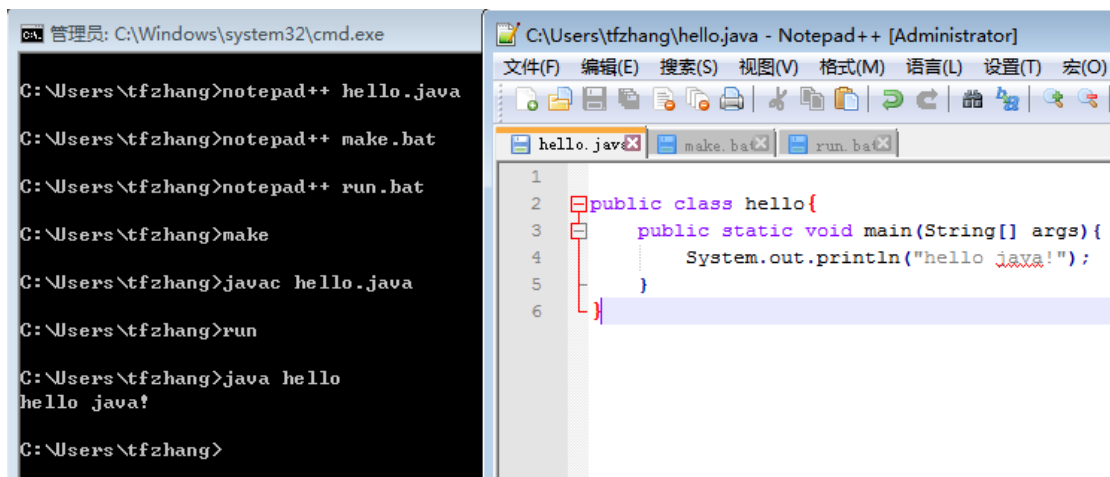
常见的 Java 开发环境的选择有两个：

1. 文本编辑器+命令行窗口；
2. Eclipse；

Eclipse 工具更加强大，对应地，更加耗费内存和计算资源；文本编辑器没有集成编译和调试功能，所以编译过程还需在命令行窗口中使用 `javac`、`java` 命令，但好处是超轻量级。选择什么工具，往往取决于目的。正像去多远的地方，采用什么样的交通工具，比如下课后要去食堂，谁也不会打车去，去北京的话，基本上没人会骑着自行车去。

根据我们的代码规模，使用 Eclipse 大多数时候是大材小用；所以不作特殊说明，本文大多数时候采用环境 1 开发。但也不直接使用 windows 自带的 notepad，本文使用 notepad++，因为一免费，二对关键词做字体颜色区别，三多标签页，切换方便；

除了 notepad++ 文本处理器，还使用批处理文件简化编译、运行命令的输入。比如将编译 `hello.java` 的命令 "`javac hello.java`" 写到 `make.bat` 批处理文件中，而运行 `hello` 的命令写入到 `run.bat` 中，编译和运行时，不需要输入一长串命令，只要输入 `make` 和 `run` 再回车即可。尤其代码源文件的数量较多时，采用批处理命令的好处更加明显，可以减少冗长重复的命令输入。如下图所示



`hello.java` 的内容不用多说，`make.bat` 批处理文件中的内容就是 "`javac hello.java`"，而 `run.bat` 中 "`java hello`"，编译时，如图中只要 "`make`" + 回车，而运行时，则 "`run`" + 回车。另外注意，`hello.java`，`make.bat`，`run.bat` 三个文件应该放置同一目录下。

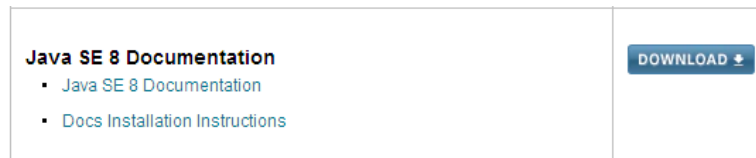
没有明确说明，本文中的开发环境是 notepad++ 和批处理文件。再提醒下读者，开发环境这么选择，由后续各个章节的源文件数量与代码量以及使用的方便性决定。如果代码量大到使用 Eclipse 更加方便，那么自然就应该使用 Eclipse。

2. 如何查阅 jdk 官方文档

1. 打开百度，输入关键词"oracle java 下载"，出现如下的网页：



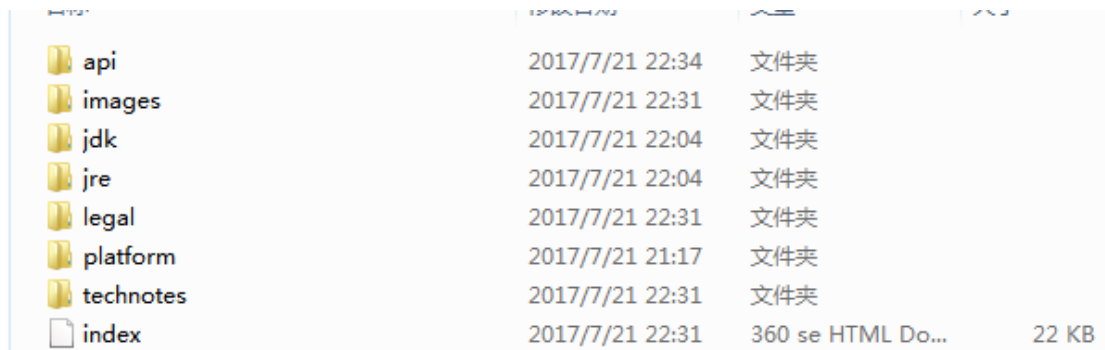
点击打开第二个链接后，往下拉直到如下的页面：



点击 Download 后，新的页面中点击第一个 accept，如下图所示：

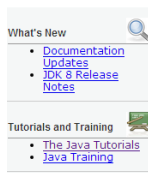


再点击下载 jdk-8u144-docs-all.zip 即可。注意该文档说明是针对 Java 8，如果要找之前版本的说明文档，既可以按照上述的文档格式直接在百度搜索，也可以在官网查找旧版本，此处不再赘述。



解压缩 jdk-8u144-docs-all.zip，目录中的 index 页面打开后，初学者可以先查看左侧的"Tutorials and Training"，其他的内容可根据自己的需要查找。

Java Platform Standard Edition 8 Documentation



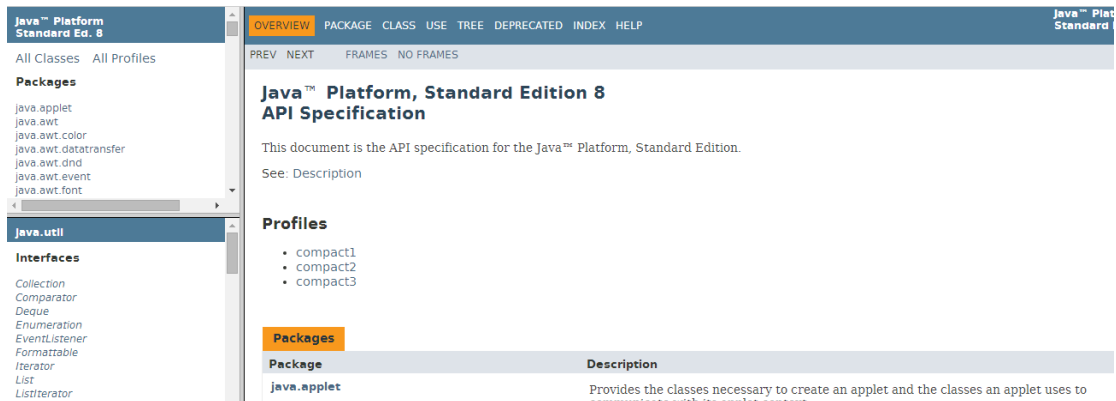
Oracle has two products that implement Java Platform Standard Edition (Java SE) 8: Java SE Development Kit (JDK) 8 and Java SE Runtime Environment (JRE) 8.

JDK 8 is a superset of JRE 8, and contains everything that is in JRE 8, plus tools such as the compilers and debuggers necessary for developing applets and applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language. Note that the JRE includes components not required by the Java SE specification, including both standard and non-standard Java components.

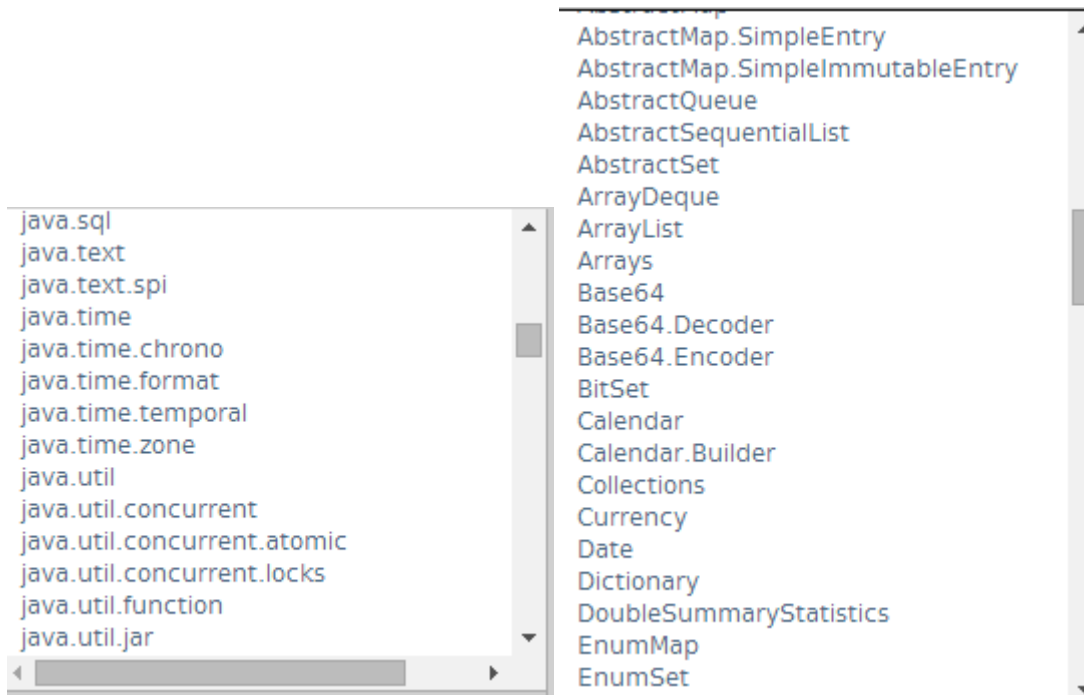
The following conceptual diagram illustrates the components of Oracle's Java SE products:

[Description of Java Conceptual Diagram](#)

如果读者要查看某个 `java` 类中的方法和变量，则打开 `docs/api/index.html` 文档，界面分为左上：`java` 包列表；左下：对应左上选定包的 `interfaces`, `classes`, `exceptions` 等；中间界面是具体的内容介绍。



不过该页面没有对类的查找功能，所以只有你知道某个类的具体所在包，才能进行查找。比如要查关于 `ArrayList` 类，并且已知 `ArrayList` 位于包 `java.util`，那么左上首先定位到 `java.util`



然后，左下再找到并且选定类 `ArrayList`，此时中间页面就呈现 `ArrayList` 类的相关信息，包含继承关系、我们特别关心的方法等；

Class ArrayList<E>

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>
```

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
boolean	add(E e)	Appends the specified element to the end of this list.
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
boolean	addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	clear()	Removes all of the elements from this list.

3. 如何查看 java 源代码

出于学习或者工作的需要，去了解 java 类的实现原理，就免不了查阅 java 的源代码。安装 jdk 时，源代码是默认安装的。



安装完毕后，根目录下有 src.zip 压缩文件，其中存放着 java 源代码。

名称	修改日期	类型	大小
bin	2017/9/7 10:14	文件夹	
db	2017/9/7 10:14	文件夹	
include	2017/9/7 10:14	文件夹	
jre	2017/9/7 10:14	文件夹	
lib	2017/9/7 10:14	文件夹	
COPYRIGHT	2017/7/21 22:21	文件	4 KB
javafx-src	2017/9/7 10:14	WinRAR ZIP 压缩...	4,979 KB
LICENSE	2017/9/7 10:14	文件	1 KB
README	2017/9/7 10:14	360 se HTML Do...	1 KB
release	2017/9/7 10:14	文件	1 KB
src	2017/7/21 22:21	WinRAR ZIP 压缩...	20,757 KB
THIRDPARTYLICENSEREADME	2017/9/7 10:14	文本文档	142 KB
THIRDPARTYLICENSEREADME-JAVAFX	2017/9/7 10:14	文本文档	63 KB

假设我们要查看 ArrayList 的源代码，那么只要打开目录/src/java/util/下的 ArrayList.java 文件即可。

软件 (D:) > Program Files > Java > jdk1.8.0_144 > src > java > util > 搜索 util				
共享 新建文件夹				
名称	修改日期	类型	大小	
<input type="checkbox"/> AbstractSet.java	2017/7/21 21:20	JAVA 文件	7 KB	
<input type="checkbox"/> ArrayDeque.java	2017/7/21 21:20	JAVA 文件	33 KB	
<input checked="" type="checkbox"/> ArrayList.java	2017/7/21 21:20	JAVA 文件	53 KB	
<input type="checkbox"/> ArrayPrefixHelpers.java	2017/7/21 21:20	JAVA 文件	31 KB	
<input type="checkbox"/> Arrays.java	2017/7/21 21:20	JAVA 文件	223 KB	
<input type="checkbox"/> ArraysParallelSortHelpers.java	2017/7/21 21:20	JAVA 文件	43 KB	
<input type="checkbox"/> Base64.java	2017/7/21 21:20	JAVA 文件	38 KB	
<input type="checkbox"/> BitSet.java	2017/7/21 21:20	JAVA 文件	41 KB	
<input type="checkbox"/> Calendar.java	2017/7/21 21:20	JAVA 文件	139 KB	

4. 如何做实证？

遇到问题时，可以有很多的解决办法，比如查书查官方文档，百度，或者自己动手实证。对提升个人能力的助益而言，应该推崇自己动手实证，古人也说，“纸上得来终觉浅，觉知此事要躬行”。另外，自己动手实证也可能是最后的办法，因为总有那么个时刻那么个场景，出现的问题，你是古今中外第一个遇到。如果真到了这个境地，又怎么能指望书籍、网络有相关的资料供参考呢？还是得自己撸起袖子，干。

下面看个例子，来看看实证是怎么回事，有了第一次，第二次就也不会那么局促畏难。问题是“有人说在 `for` 循环遍历中，尽量用其他数据结构替换 `LinkedList`，比如 `ArrayList`。”我们要验证这个说法是否有道理，及找出原因。`LinkedList` 在功能上应该没问题，之所以不建议用，估计是性能上的问题，因为常识告诉我们，遍历是程序中最耗费时间的。第一步就清晰了，分别采用 `LinkedList` 和 `ArrayList` 实现同等规模数据的遍历，再比较执行时间即可。潜在的问题是如何统计程序执行的时间？

Java 已经提供统计时间的方法 `System.currentTimeMillis()`，返回的是从 1970 年 0 点 0 分 0 秒截止到目前的时间间隔，以毫秒为单位，数据值自然比较大，所以一般用 `long` 数据类型存储。在待测代码之前运行 `currentTimeMillis()` 方法，获得一个 `long` 数据，再与待测代码之后运行 `currentTimeMillis()` 方法得到的 `long` 数据相减得到的差，就是该代码运行的时间。主体代码如下：

TestList.java:

```
public static void testLinkedList(int n){
    LinkedList<Integer> list = new LinkedList<Integer>();
    for(int i=0; i<n; i++){
        list.add(i);
    }
    long time1 = System.currentTimeMillis();
    //遍历输出;
    for(int i=0; i<n; i++){
        list.get(i);
    }
    long time2 = System.currentTimeMillis();
    System.out.println("linkedlist: "+n+" elements time="+time2-time1);
}

public static void testArrayList(int n){
    ArrayList<Integer> list = new ArrayList<Integer>();
    for(int i=0; i<n; i++){
        list.add(i);
    }
    long time1 = System.currentTimeMillis();
    //遍历输出;
    for(int i=0; i<n; i++){
        list.get(i);
    }
}
```

```

        long time2 = System.currentTimeMillis();
        System.out.println("arraylist: "+n+" elements time="+time2-time1));
    }

```

经过三次平均后的时间数值：

元素个数	LinkedList	ArrayList
1000	2	0
10000	50.6	1
100000	7204.3	1
300000	97461.7	12

明显可以看到 LinkedList 花费的时间要远大于 ArrayList 的时间。其中的数值 0，并不是不花费时间，而是因为毫秒为单位的测量精度不够。

为什么 LinkedList 比 ArrayList 要花费更多的时间呢？我们需要深入两者的源代码来找到答案（如何查找源代码见前面的章节，使用的 java 版本是 java 1.7.0_45）。先来看 ArrayList 中对应的 get 方法，因为 elementData 是个数组结构，就是简单的按地址取数据，耗时间自然很少。

```

    public E get(int index) {
        rangeCheck(index);
        return elementData(index);
    }

```

```

    private transient Object[] elementData;

```

再来看 LinkedList 中的源代码，要注意的是 node 并不是一个数组结构，而是一个方法。node 方法的输入是 index，然后通过二分方法，查找对应的节点。size>>1 相当于 size/2，所以根据 index 是否大于 size/2 来确定是从前往后查找，还是从后往前查找。因此最坏的运气是至少要查找几乎 size/2 量的节点，才能找到我们的目标节点。

```

    public E get(int index) {
        checkElementIndex(index);
        return node(index).item;
    }

```

```

    Node<E> node(int index) {
        // assert isElementIndex(index);
        if (index < (size >> 1)) {
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }

```

如果还没看懂源代码的话，请把注意力集中在 `x = x.next` 和 `x=x.prev`，可见每个节点都有

前后指针分别指向前一个元素和后一个元素，查找过程就是顺藤摸瓜的过程。举个例子，加入 $size=9$ ，我们要查找的节点序号是 4，因为 4 不小于 $9/2=4$ ，所以会从尾部往前摸，先摸 8，7,6,5，最终摸到 4，所以至少要摸近 5 次，所以最坏的情况就是要摸近 $size/2$ 次。

上述的过程很有代表性，包含实证过程的两个基本要素：

1. 针对问题的实验代码设计；
2. 追根溯源的源代码阅读；

读者遇到好的问题，也可以多使用实证的方法，过程中，忍受追根溯源，繁琐细节不知方向的沉闷；享受证得结果，云开雾散了然而然的快感。如此反复，能力在不知不觉中提升。

第二部分 Java 基本内容

1. Java 的类和对象
2. Java 图形界面
3. Java 与数据库开发
4. Java 与异常
5. Java 与正则表达式
6. Java 多线程
7. Java 与面向对象程序设计基础
8. 测试驱动(TestDriven)的开发方法
9. 基于 MVC 框架的开发方法

1. Java 的类和对象

什么是类？

学过 C 语言的同学第一眼看到 java 中的类，会认为其与结构体类似。的确，从代码角度看，类是具备方法的结构体。就概念而言，类与结构体却有着天壤之别：结构体只是数据集合，而**类则是对自己行为负责的主体**。类是对象属性和行为的封装。类的封装使得类对自己的行为负责，减少调用方对该类行为的干预。而且，类对方法的封装使其内部行为的变化对外在对象不可见。比如下的 student 类：

```
public class student{
    int weight;
    int height;

    public void speak(){
        ....
    }
    public void eat(){
        ....
    }
}
```

student 类中包含对象的身高、体重，以及说话和吃饭的行为。外界让对象 student 吃饭，那么只要调用对象的 eat()方法即可，外界不需要知道对象吃什么，怎么吃的。因此，对象的具体吃饭行为对外界不可见，当然，吃饭行为也不用外界负责。就像老师中午下课让同学们吃饭去一样，老师只需发出下课的信号，而不需要去负责每个学生对象的吃饭行为。每个学生对自己的吃饭行为负责。

类的基本定义：

定义类的主要关键词是 class，其后加上具体类的名字，比如 class student{}，表示创建一个名为 student 的类。至于 public 关键词，则表示类 student 的访问控制模式。什么是访问控制模式？与 C 语言中的作用域概念类似，限定类的可引用范围。一般有如下的四种类访问控制模式：

	同一类	同一包中的类	子类	外部包的类
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No

理解上述的类访问控制模式，需要对类包(package)和继承等概念有理解，后续会详细介绍。

类的使用：

如下是一个调用 student 对象的类，暂且命名为 TestStudent，代码如下：

```
public class TestStudent{
    public static void main(String[] args){
        Student zhang3 = new Student();
        zhang3.speak();
        zhang3.eat();
    }
}
```

在使用 Student 类时，需要通过 new Student() 获得一个类的实例，也即 zhang3 同学；然后才能针对 zhang3 调用 speak 和 eat 方法。而 zhang3 同学能够使用 speak 和 eat 方法，因为 zhang3 同学是 Student 类的一个实例。

类对内部数据的访问控制：

假设 student 类中，还有关键的分数 score 信息：

```
class student{
    int weight;
    int height;
    int score;

    public void speak(){
        ....
    }
    public void eat(){
        ....
    }
}
```

假设还有一个老师类，其设置 zhang3 的 score 的代码如下：

```
public class Teacher{
    public static void main(String[] args){
        Student zhang3 = new Student();
        zhang3.score = 101; //出现笔误.
    }
}
```

不能保证老师在设置分数时，不出现笔误，使得出现 101 这样超出 0~100 范围的分数值。要杜绝这样的问题，就要对类的敏感数据的访问进行限制。所以，需要使用 private（相应地，还有 public）关键词来修饰敏感数据。

假设 private int score; 那么教师类中直接使用 zhang3.score=101 就是违法语句。因为 score 是 student 私有的数据，外部类不能访问。在保护敏感数据的同时，通过什么方式对敏感数据进行设置呢？答案是 Student 类内部专门提供操作分数的方法，而外部类通过调用该方法完成对私有数据的设置。比如在 Student 类内部增加 public void setScore(int s) 方法，然后教师类调用 zhang3 的 setScore 方法，对成绩进行设置。

```
public class Teacher{
    public static void main(String[] args){
        Student zhang3 = new Student();
        zhang3.setScore(101);
    }
}
```

Student.java:

```
public void setScore(int s){
    if(s>=0 && s<=100)
        score = s;
    else
        System.out.println("illegal score");
}
```

虽然还是出现 101 的笔误，但是 setScore 方法中，我们增加监测代码，对超出范围的分数，直接无效化，从而审核控制外部类对敏感数据的访问。所以，private 关键字帮助我们对类内部数据进行访问控制。

类的层次划分

以类为粒度来看待 Java 程序，需要捋清以下三层关系：

1. 包(package)与源文件；
2. 源文件与类；
3. 类与类；

一个类就是一段以 class 开头的代码，一个类可以写在一个.java 源文件中，也可以多个类写在一个.java 源文件中。特别要注意，一个.java 源文件中只能有一个 public 修饰的类，并且.java 文件名必须与 public 修饰的类名相同。当源文件的数量很多时，需要采用 package 进行组织，当前暂时将 package 理解为文件夹，即将大量的源代码按文件夹(package)组织。除了创建文件夹，对源代码本身也要作必要的修改，涉及 package 和 import 两个关键词的使用。

下面举两个简单的例子，说明 package 和 import 的作用。

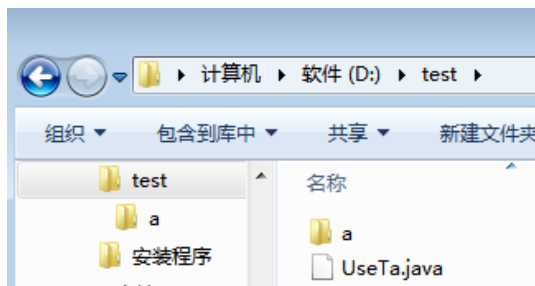


图 1

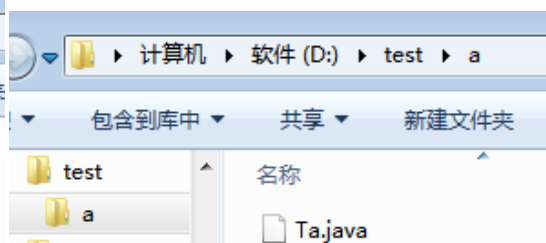


图 2

第一种情况：

UseTa.java 在当前目录 test(如图 1 所示)，Ta.java 在当前目录 test 的 a 目录下(如图 2 所示)，UseTa 类会使用到类 Ta，两者的代码如下：

UseTa.java: <pre>public class UseTa{ public static void main(String[] args){ Ta ta = new Ta(); } }</pre>	Ta.java: <pre>public class Ta{ }</pre>
---	---

表 1

现在因为两者不在同一个目录，所以直接按照表 1 的源代码编译 UseTa.java 时，会出现“错误：找不到符号 Ta。”，因为 Ta.java 与 UseTa.java 不在同一个目录，所以导致 UseTa 找不到对象 Ta。解决的办法就是要告诉 UseTa, Ta 对象的具体位置，使用的关键词就是 import 和 package。具体的修改如下：

UseTa.java: <pre>import a.*; public class UseTa{ public static void main(String[] args){ Ta ta = new Ta(); } }</pre>	Ta.java: <pre>package a; public class Ta{ }</pre>
---	--

表 2

在 Ta.java 中，增加“package a;”语句，说明当前的类 Ta 隶属于 package a（与文件夹同名）；而 UseTa.java 中，增加“import a.*”语句，说明可以使用隶属于 package a 下的类。特别地，UseTa.java 可以不使用 import a.*，但取而代之代码的书写比较麻烦：

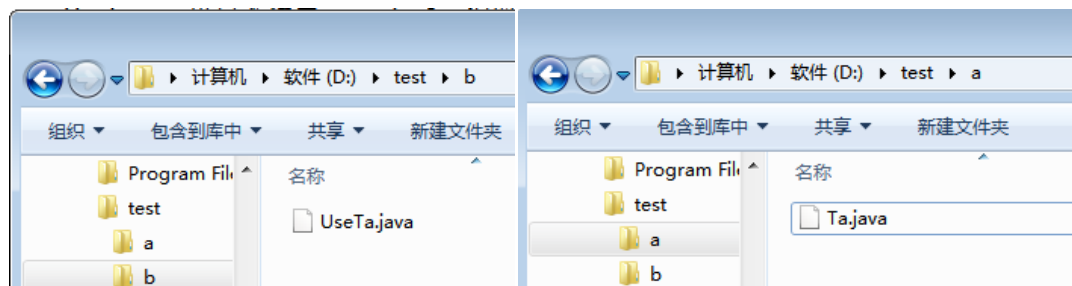
```
UseTa.java:
public class UseTa{
    public static void main(String[] args){
        a.Ta ta = new a.Ta();
    }
}
```

表 3

也即每个使用到 Ta 对应，都要注明其隶属的包 a。

第二种情况：

Ta.java 在当前目录 test 的 a 目录下，并且 UseTa.java 在当前目录 test 的 b 目录下，两者源代码同表 1。



为使得 UseTa 对象找到 Ta 对象，两者的修改如下：

UseTa.java: package b; import a.*; public class UseTa{ public static void main(String[] args){ Ta ta = new Ta(); } }	Ta.java: package a; public class Ta{ } }
---	--

表 4

不要在 b 目录中编译 UseTa.java 文件，而要退出 b 目录，到当前目录 test 中进行编译：

```
javac ./b/UseTa.java
```

编译后生成的 UseTa.class 文件，自动在 b 目录中。此时如何运行 UseTa.class 文件呢？也是在当前目录 test 下执行命令：java b.UseTa。上述两个简单案例的嵌套使用，比如目录 a 里面再有目录 c 等等，对应的包名就是 package a.c，同样地，import 的时候，就是 import a.c.*；按照上述思路，就可以构建多层次的包结构。

同一个源代码文件中，一般只能存在一个 **public** 类，而且该类必须与文件同名。比如 **Student** 类，必须保存在 **Student.java** 源文件中。如果源文件的名字是 student.java，则编译时会出现错误，因为即使大小写不一致也不行。

非 public 的类不能用 private 修饰，默认不采用关键词修饰，即 default 模式。default 模式允许同一个 package 中的类相互访问。比如 Student.java 文件中，再创建一个 Score 类，那么 Score 类的访问控制关键词既不能是 public，也不能是 private，只能是如下的形式：

Student.java:

```
public class Student{  
...  
}  
class Score{  
...  
}
```

另外，类中还可以包含其他类，即类中的数据还可以是类，主要的好处是提升代码的模块性，因为被包含的类也只被当前类使用。比如将 Score 类作为 Student 类的一个内部类。

Student.java:

```
public class Student{  
    class Score{  
        ...  
    }  
...  
}
```

类的继承

继承是类与类之间依赖性最强的一种关系。子类通过继承，分享父类中定义的数据和方法。一个父类可以被多个子类继承，而 Java 中的一个子类只能继承一个父类。同时，子类也可以定义自己的方法，拥有自己的数据。如下的 Father 类与 Son 类，Son 类继承自 Father。

Father.java:

```
public class Father{
    public String skincolor = "yellow";
    public String eyes = "double eye";

    public Father(){
        System.out.println("father constructed");
    }
    public void speak(){
        System.out.printf("Speak Chinese");
    }

    public void eat(){
        System.out.println("Eat rice");
    }
}
```

Son.java:

```
public class Son extends Father{
    public Son(){
        System.out.println("son constructed");
    }

    public static void main(String[] args){
        Son s = new Son();
        System.out.println("color of son:"+s.skincolor+ " eyes of son:"+s.eyes);
        s.speak(); s.eat();
    }
}
```

子类除了继承父类的方法和属性，也可以拥有自己的方法。另外，子类如果觉得父类的方法不够完善，还可以重写父类的方法。比如 `son` 除了学习汉语，还学习英语，因此子类的 `speak` 方法需要更新。

```
public void speak(){
    Sytem.out.println("Speak Chinese and English");
}
```

重新运行代码后，`s.speak()`方法执行的就是子类自身的版本，也即对父类的 `speak` 方法进行覆盖。

观察上述源代码，读者可以发现一个比较特殊的方法：与类名相同的方法，且没有返回类型。比如 `Father` 类中的 `public Father()`方法，`Son` 类中的 `public Son()`方法。这种特殊的方法称为类的构造函数，类实例创建时必须调用的方法，如果程序员没有显示地写构造函数，系统也会调用默认的构造函数。所以，程序员即使不写也没有关系，但如果要对类作一些初始化操作的话，都会将初始化代码写在显示的构造函数中。

上述示例代码中，还需要注意的一点就是关于构造函数的调用。子类在调用自身构造函

数之前，首先会自动调用父类的无参构造函数，再调用自身的构造函数。某个类被声明为 `final` 时，则该类不能被继承。

什么是类的多态性？

多态指的是以相同的方法名调用方法，但得到不同的行为，实现的方式包括方法的重载和方法的重写。方法的重载，指在同一个类中的以同一个函数名命名的多个方法，方法之间通过参数的个数和顺序进行区分。方法的重写指在继承类之间，子类继承且重写父类的方法。下面是一个重载的列子。

Area.java:

```
public class Area{
    public void getArea(int length, int width){
        int area = length*width;
        System.out.println("area of rectangular is "+area);
    }
    public void getArea(int r){
        float area = (float)3.14*r*r;
        System.out.println("area of circle is "+area);
    }
    public void getArea(float width, float height){
        float area = (float)0.5*width*height;
        System.out.println("area of triangular is "+area);
    }
}
```

TestArea.java:

```
public class TestArea{
    public static void main(String[] args){
        Area a = new Area();
        a.getArea(3);
        a.getArea(2,3);
    }
}
```

Area 类中有三个 `getArea` 方法，虽然方法名相同，但分别求方形，圆形和三角形的面积，之间的不同通过参数来区分。

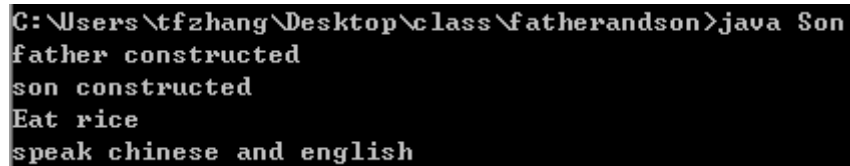
基于 Son.java 类的方法重写的例子：

Son.java:

```
public class Son extends Father{
    public Son(){
        System.out.println("son constructed");
    }
    public void speak(){
        System.out.println("Speak chinese and english");
    }
    public static void main(String[] args){
```

```
        Father s = new Son();
        s.eat();
        s.speak();
    }
}
```

运行后的结果:



```
C:\Users\tfzhang\Desktop\class\fatherandson>java Son
father constructed
son constructed
Eat rice
speak chinese and english
```

s 虽然是父类对象，但不是本身的类实例化，还是 Son 子类的实例，因此，调用的还是子类的 speak()方法。

抽象类?

什么是抽象类? 存在抽象方法的类属于抽象类，抽象方法是一种只有方法的声明，但没有方法实现的方法。以求面积为例，增加一个 shape 抽象类:

```
abstract class Shape{
    int r;
    int length, width;
    float height;
    String name;
    abstract void getArea();

    public Shape(int r){
        this.r = r;
    }

    public Shape(int length, int width){
        this.length = length;
        this.width = width;
    }

    public Shape(int width, float height){
        this.width = width;
        this.height = height;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

```
public class Circle extends Shape{
```

```

    public Circle(int r){
        super(r);
        setName("circle");
    }

    public void getArea(){
        float area = (float)3.14*r*r;
        System.out.println("area of "+name+" is "+area);
    }
}

public class Rectangular extends Shape{
    public Rectangular(int length, int width){
        super(length, width);
        setName("rectangular");
    }

    public void getArea(){
        int area = length*width;
        System.out.println("the area of "+name+" is "+area);
    }
}

```

Circle 类和 Rectangular 类作为子类继承自 Shape 类，并且实现各自的 getArea 方法。抽象方法提取了各种 Shape 中共性的部分。

接口是什么？

什么是接口？接口是抽象类的变体，接口也包含数据和未被实现的方法，但不会包含实现了的方法。抽象类是对事物的抽象，而接口更多是对动作和行为的抽象。抽象类表示这个对象是什么，而接口表示这个对象能做什么。Java 只能继承一个类，但可以实现多个接口，因此接口是一种 Java 实现多继承机制的方法。两者的异同总结如下：

1. 抽象类和接口都是用来抽象具体对象的，但是接口的抽象级别最高；
2. 抽象类可以有具体的方法和属性，接口只能有抽象方法和不可变常量；
3. 抽象类主要用来抽象类别，接口主要用来抽象功能；
4. 抽象类中且不包含任何实现，派生类必须覆盖它们；接口中所有方法都必须是未实现的；

再以求面积(area)为例说明接口，示例代码见 interface 文件夹。

Shape.java:

```

interface Shape{
    public void getArea();
    public void setName(String name);
}

```

Circle.java:

```

public class Circle implements Shape{

```

```

    private int r;
    private String name;
    public Circle(int r){
        this.r = r;
    }
    public void setName(String s){
        name = s;
    }
    public void getArea(){
        float area = (float)3.14*r*r;
        System.out.println("area of "+name+" is "+area);
    }
}

```

接口是面向对象设计的基础，需要仔细领会。

课后练习

问题 1：分别针对 `Int` 数据类型，`double` 数据类型和 `String` 数据类型，撰写三个冒泡排序（从小到大排序）的类：`IntBubbleSorter`，`DoubleBubbleSorter`，`StringBubbleSorter`。每个类至少包含排序方法 `bubbleSort()` 和将数据打印到终端的方法 `showData()`。测试类如下：

TestBubble.java:

```

public class TestBubble{
    public static void main(String[] args){
        IntBubbleSorter is = new IntBubbleSorter();
        int a[] = {3, 2, 1, 4};
        is.bubbleSort(a);
        is.showData(a);

        DoubleBubbleSorter bs = new DoubleBubbleSorter();
        double b[] = {4.1, 1.1, 2.1, 3.1, 5.3};
        bs.bubbleSort(b);
        bs.showData(b);

        StringBubbleSorter ss = new StringBubbleSorter();
        String strs[] = {"zhang3", "li4", "Wang5", "qian6", "liu7"};
        ss.bubbleSort(strs);
        ss.showData(strs);
    }
}

```

要求：完成上述的三个类，并且通过测试，即完成排序并且将排序后的打印到终端。注意：`string` 自带 `compare` 方法，如何使用，具体查看官方 API 文档。

2. Java 图形界面

组件和容器？

要掌握 Java 图形界面开发，首先需理解组件和容量两个概念。组件放置在容器中，容器包含组件。日常生活中，书包就是一个容器，而书包里的一本书、一支笔都是组件。当你把书包放到车里时，车就变成容器，而书包就成为组件。所以，组件和容器是相对的概念。容器可以被放置到更大的容器中，因此容器可以包含组件，也可以包含其他的容器。容器之间的包含关系也反映容器之间层级关系，上一级的容器自然可以包含下一级的容器。

组件如何放置到容器中？

Swing 中的各种布局方式 Layout 来决定。Java 提供现成的用户界面程序开发工具包 Swing。开发人员只需调用 Swing 中的组件和容器，就可以完成界面开发，十分便利。前面已经提过容器的层级问题，JFrame 是 Swing 中常用的最高层级。图 1 是 JFrame 的例子，Frame 内部放置一菜单栏和 contentPane，contentPane 中放置一黄色的标签。

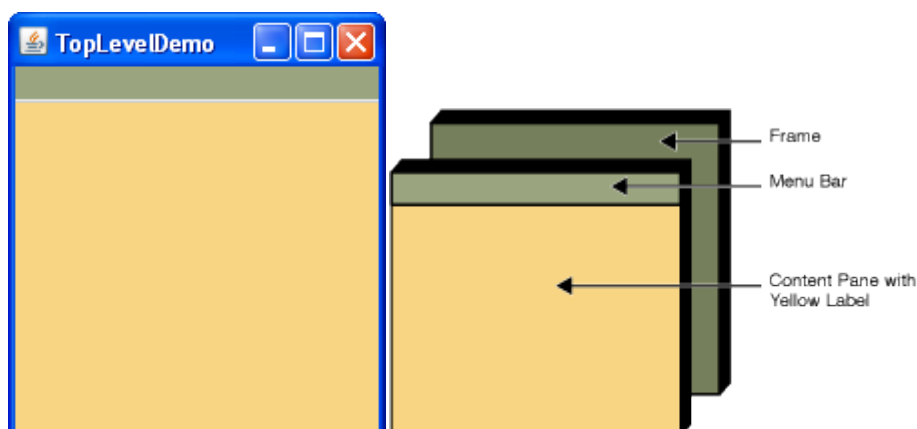
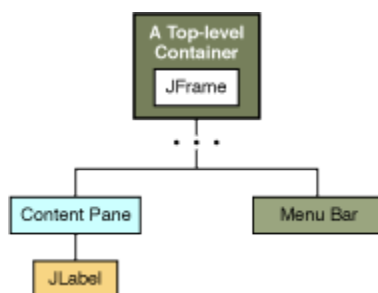


图 1

上述各对象的层级关系如下：



注意 JLabel 没有直接放置，而是通过间接地放置在 Content Pane 实现放置在 Frame 中。具体的代码见文件夹 MyFrame.java:

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;
```

```

public class MyFrame{
    public MyFrame() {
        JFrame frame = new JFrame("MyFrame");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

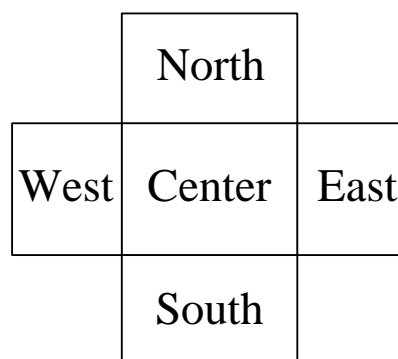
        JMenuBar greenMenuBar = new JMenuBar();
        greenMenuBar.setBackground(new Color(154, 165, 127)); //三原色调配出目标颜色
        greenMenuBar.setPreferredSize(new Dimension(200, 20)); //设定 MenuBar 的大小

        JLabel yellowLabel = new JLabel();
        yellowLabel.setBackground(new Color(248, 213, 131));
        yellowLabel.setPreferredSize(new Dimension(200, 180));

        frame.setJMenuBar(greenMenuBar); //将 menuBar 添加到 JFrame 容器中
        frame.getContentPane().add(yellowLabel, BorderLayout.CENTER); //将 yellowLabel 添
加到 ContentPane 中。
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        new MyFrame();
    }
}

```

注意放置 JLabel 部分代码：frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);
 add 方法中有 BorderLayout.CENTER 参数，就是设置 Label 在 ContentPane 中放置的方式
 与位置。还有很多其他的 Layout 方式，读者可以在应用时查阅官方 API 文档或者其他相关
 资料，此处不再赘述。BorderLayout 布局表示将容器分割如下：



参数 BorderLayout.CENTER 表示 yellowLabel 放置在容器的中心位置。

另外值得留意的代码如下：

```

JMenuBar greenMenuBar = new JMenuBar();
greenMenuBar.setBackground(new Color(154, 165, 127)); //三原色调配出目标颜色
greenMenuBar.setPreferredSize(new Dimension(200, 20)); //设定 MenuBar 的大小

```

其中的 new Color 部分的三个数值是设定红、黄、蓝三原色的纯度，每个值的范围是 0~255，
 值越大说明纯度越高。读者如果尝试 new Color(255, 0, 0)，则显示纯粹的红色。另外，new

Dimension(200, 20)用来设置二维体的长和宽，并且通过 `setPreferredSize` 将二维体的大小设定给 `greenMenuBar`。

关于 `JFrame` 还有两个方法：`pack()`和`setVisible(true)`。`setVisible`方法顾名思义，表示 `JFrame` 是否处于可见状态。**`pack()`的功能？**不能直接从方法名猜测的话，可以尝试注释掉再运行程序，观察前后的效果来推测 `pack` 方法的功能，请读者自己实践。另外，还可以通过查询官方 API 文档确认 `pack` 方法的功能。

尝试对 `JFrame.java` 源代码做简单的修改，比如在 `JFrame` 中增加一个按钮组件，显示在 `yellowLabel` 下，并且按钮上显示"press me"。主要的步骤包括：1、创建按钮对象；2、将按钮对象放置在对应的位置。步骤 1 的代码如下：

```
JBUTTON jb = new JBUTTON("press me");
jb.setPreferredSize(new Dimension(50, 20));
```

步骤 2 对应的代码如下：

```
frame.getContentPane().add(yellowLabel, BorderLayout.SOUTH);
```

请读者将上述代码添加到 `JFrame.java`，编译运行程序，实现按钮的正确添加。

让鼠标响应事件

点击鼠标后，界面一般都会有所反应，无论是关闭界面，还是弹出窗口。要实现界面对于用户操作的变化，需要引入事件的概念。什么是事件？用户对程序界面的每个操作，就对应一个事件。比如 `Word` 文档区，每输入一个字符就是一个事件；鼠标点击按钮，也是一个事件；敲击键盘上的任意一个按键，也是一个事件。

`Java` 是面向对象的编程语言，所以每个事件也被封装为一个个对象。日常经验显示当按钮被点击后，能看到界面对应的反馈，比如弹出新的对话框。整个过程可由如下的链条描述：点击动作->触发事件->捕捉事件->进行对应的处理。当鼠标点击按钮时，产生按钮点击事件，如何区分不同按钮的点击事件？对按钮而言，可以给每个按钮设置一个身份标签，不同的按钮拥有自己独一无二的标签区分，通过方法 `setActionCommand` 实现。对按钮事件的处理，则通过 `actionPerformed` 方法。

例 1：以 `MyFrame` 中的按钮为例，假设要实现如下的功能，每点击一次按钮，按钮上显示当前的点击次数。

按钮上的文字要求能够更新，这种需求不是现在才有，以前的程序员也有，因此可确定 `Java` 肯定会提供对应的 API。打开 `Java` API 文档，定位到 `JButton` 类后，可在 `Method Summary` 下面看到 8 个方法，但没有一个符合我们的需求。再往下看，可发现 `JButton` 继承大量来自 `javax.swing.AbstractButton`，`javax.swing.JComponent`，`java.awt.Container` 和 `java.awt.Component` 的方法。即使只是从字面意思去揣测哪个是目标方法，面对这么多方法，人也有点不知所措。

Methods inherited from class `java.awt.Component`

```
action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener,
addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains,
createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods,
firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground,
getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor,
getFocusListeners, getFocusTraversalKeysEnabled, getFont, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners,
getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale,
getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent,
getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, inside,
isBackgroundSet, isCursorSet, isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet,
isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list,
location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage,
prepareImage, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent,
processMouseEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener,
removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener,
removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds,
setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation,
setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusBackward, transferFocusUpCycle
```

此时应该停下来想一想，目标是在按钮上设置文字，自然和关键词 `text` 有关（为什么

不是 `String`？这依赖于经验，一般是 `text` 优先；如果 `text` 不行，再考虑 `String` 也不迟。)，因此可以基于关键词 `text` 进行查找。`AbstractButton` 中有大量的 `Text` 相关的方法。因为是要设置按钮文字，所以凡是“`get`”前缀的方法，可以排除。

Methods inherited from class `javax.swing.AbstractButton`

`actionPropertyChange`, `addActionListener`, `addChangeListener`, `addImpl`, `addItemListener`, `checkHorizontalKey`, `checkVerticalKey`, `configurePropertiesFromAction`, `createActionListener`, `createActionPropertyChangeListener`, `createChangeListener`, `createItemListener`, `doClick`, `doClick`, `fireActionPerformed`, `fireItemStateChanged`, `fireStateChanged`, `getAction`, `getActionCommand`, `getActionListeners`, `getChangeListeners`, `getDisabledIcon`, `getDisabledSelectedIcon`, `getDisplayedMnemonicIndex`, `getHideActionText`, `getHorizontalAlignment`, `getHorizontalTextPosition`, `getIcon`, `getIconTextGap`, `getItemListeners`, `getLabel`, `getMargin`, `getMnemonic`, `getModel`, `getMultiClickThreshold`, `getPressedIcon`, `getRolloverIcon`, `getRolloverSelectedIcon`, `getSelectedIcon`, `getSelectedObjects`, `getText`, `getUI`, `getVerticalAlignment`, `getVerticalTextPosition`, `imageUpdate`, `init`, `isBorderPainted`, `isContentAreaFilled`, `isFocusPainted`, `isRolloverEnabled`, `isSelected`, `paintBorder`, `removeActionListener`, `removeChangeListener`, `removeItemListener`, `setAction`, `setActionCommand`, `setBorderPainted`, `setContentAreaFilled`, `setDisabledIcon`, `setDisabledSelectedIcon`, `setDisplayedMnemonicIndex`, `setEnabled`, `setFocusPainted`, `setHideActionText`, `setHorizontalAlignment`, `setHorizontalTextPosition`, `setIcon`, `setIconTextGap`, `setLabel`, `setLayout`, `setMargin`, `setMnemonic`, `setModel`, `setMultiClickThreshold`, `setPressedIcon`, `setRolloverEnabled`, `setRolloverIcon`, `setRolloverSelectedIcon`, `setSelected`, `setSelectedIcon`, `setText`, `setUI`, `setVerticalAlignment`, `setVerticalTextPosition`

对比剩下的方法，`setText` 的可能性最大。点击查看 `setText` 方法的说明，可确定就是我们需要的办法。

```
setText

public void setText(String text)

Sets the button's text.

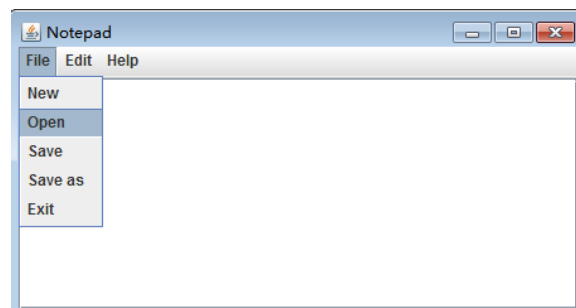
Parameters:
text - the string used to set the text

See Also:
getText()
```

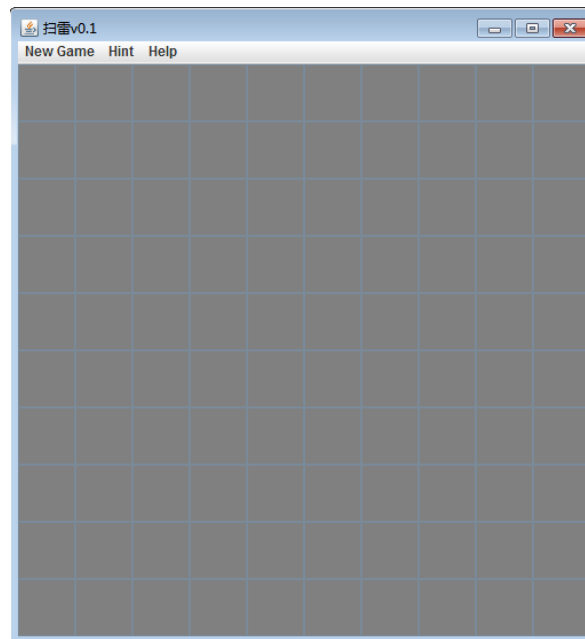
另外要注意的一点，`actionPerformed` 方法由谁来实现是个很重要的问题。根据面向对象设计中“自己的事情自己负责”原则，每个按钮应该有权利指定：发生在自身的点击事件由谁来处理。具体则由方法 `addActionListener` 实现（作者怎么知道要用该方法？因为作者看过示例代码，反推得出这样的结论呀。），查阅该方法，发现一个输入参数是一个接口 `ActionListener`，其中有唯一的方法：`void actionPerformed(ActionEvent e)`。只要有一个实现 `ActionListener` 接口以及其中的 `actionPerformed` 方法的类，再将该类与按钮绑定，就可以让发生在按钮上的事件由该类中的 `actionPerformed` 方法来实现。

请读者参考官方 `Java API` 文档或者查找书籍实现例 1；如果还不明白上述的文字，请查看官方文档中的示例代码。案例 1 的示例代码见文件夹 `MyFrame2`。其他组件的事件处理与按钮的处理类似，应用时请参照官方文档或者其他书籍即可。学以致用，我们来看两个 `UI` 程序界面的案例。

案例 2：第一个是简单的记事本界面，具体代码见 `Notepad` 文件夹，使用到的组件有 `JMenuBar`, `JMenu`, `JMenuItem`, `JScrollPane`, `JText` 等。



案例 3：如下的扫雷游戏的界面，主要是 `JButton` 的二维排列，并且每个按钮点击后改变颜色，具体代码见 `MineView` 文件夹。



Java 图形界面的设计开发介绍到此，主要的方法是参考示例代码和查阅官方 API 文档。实际的开发过程中，采用可视化的开发工具，比如 **windowbuilder**，将界面的开发过程转化为画图的过程，提高开发速率。

参考资料:

官方文档: <http://docs.oracle.com/javase/tutorial/uiswing/components/button.html>

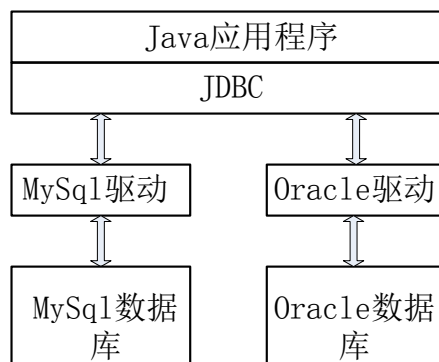
练习

问题 1: 进一步完善扫雷游戏，每个灰色小方块鼠标点击后变换为白色.

3. Java 与数据库开发

Java 如何访问数据库？

一般的应用程序都会有存储、读取、更新与删除数据的需求，要满足这些需求，最好就是应用程序连接数据库。Java 基于如下的方式访问数据库：



图中显示两种常用的数据库 MySQL 和 Oracle，前者小巧免费，后者功能强大但收费。一般我们使用 MySQL。如果使用 MySQL 数据库，那就要安装对应版本的 MySQL 驱动（MySQL 数据库厂商会提供，我们只负责安装）。程序开发人员需要负责的是 Java 应用程序与 JDBC 部分的开发；JDBC 的全称是 Java Database Connectivity，Java 数据库连接。JDBC 提供的是一组与平台无关，可用于连接数据库及执行 SQL 语句的 API。

什么是 SQL 语句？SQL，全称是 Structured Query Language，结构化查询语言，是关系数据库的标准语言。既然是语言，就离不开语句，常见的 SQL 的语句和例句如下：

	例句	含义
创建数据库	Create Database test;	创建一个名为 test 的数据库；
创建数据表	Create table student (id VARCHAR(10) not null, Name VARCHAR(16) not null, Phone VARCHAR(16) not null)	创建一个名为 student 的数据表，其中包含 id, name 和 phone 三个值；
插入数据	Insert into student (id, name, phone) values ('12345', 'zhang3', '1371234567');	向数据表 student 中插入一个叫 zhang3 的学生信息；
查询数据	Select id, name, phone from student where id=12345	从数据表 student 中查询 id 为 12345 的学生的学号，姓名与电话；
删除数据	Delete from student where id=12345	从数据表 student 中删除 id 为 12345 的学生信息；

sql 还有很多其他语句，此处不一一赘述，后续介绍 mysql 时，还会以上述例句为例说明。

MySQL 是常用的一种免费的数据库，本文使用的 mysql-5.6.0 版本，具体的安装过程请见附录 1。安装完后，可能在 configuration 遇到的问题，不用担心，可借助网络搜索解决。使用时，在命令行输入 mysql，启动客户端，界面如下：

```
管理员: C:\Windows\system32\cmd.exe - mysql -u root -p

C:\Users\tfzhang>mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.6.0-m4 MySQL Community Server (GPL)
```

mysql 后面的参数-u, 用户名; -p, 提示密码输入。

输入正确的密码后, 则正确登陆。登陆后, 一般会出于安全的考虑创建一个新的用户, 因为 root 用户的权限太高。

创建新用户之前, 先通过命令 show database 查看当前已有的数据库:

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| test |
+-----+
4 rows in set (0.00 sec)
```

可以发现已经存在 test 数据库。

例 1: 创建新用户 user1, 只对 test 数据库享有所有权限。

创建用户 user1 并且指定密码为 12345 的命令:

```
mysql> create user "user1@localhost" identified by "12345"
-> ;
Query OK, 0 rows affected (0.00 sec)
```

注意每条 sql 语句须以分号结尾, 如果以回车结尾, 系统默认为命令还没结束, 会以符号"->"换行。赋予 user1 对 test 数据库的所有权限的命令:

```
mysql> grant all privileges on test.* to user1@localhost identified by "12345";
Query OK, 0 rows affected (0.00 sec)
```

退出当前的 root 用户, 再以 user1 登陆, 运行 show databases 后, 果然可以看到其中有 test;

例 2: 删除 test 数据库后再重新 create, 创建 student 表, 插入 zhang3 和 li4, 查询 zhang3 的信息, 再删除 li4。

删除 test 数据库和重建 test 的命令如下:

```
mysql> drop database test;
Query OK, 0 rows affected (1.18 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
+-----+
1 row in set (0.00 sec)

mysql> create database test;
Query OK, 1 row affected (0.00 sec)
```

test 数据库中创建 student 表的命令如下:

```
mysql> create table student(  
-> id varchar(10) not null,  
-> name varchar(16) not null,  
-> phone varchar(16) not null);  
Query OK, 0 rows affected (0.34 sec)
```

向数据表中插入 zhang3 和 li4:

```
mysql> insert into student(id, name, phone) values('1', 'zhang3', '1234567');  
Query OK, 1 row affected (0.09 sec)  
  
mysql> insert into student(id, name, phone) values(  
-> '2', 'li4', '2345');  
Query OK, 1 row affected (0.04 sec)
```

现在要查询 zhang3 的信息:

```
mysql> select * from student where name="zhang3";  
+-----+-----+-----+  
| id | name | phone |  
+-----+-----+-----+  
| 1 | zhang3 | 1234567 |  
+-----+-----+-----+  
1 row in set (0.00 sec)
```

删除 zhang3 的信息:

```
mysql> select * from student;  
+-----+-----+-----+  
| id | name | phone |  
+-----+-----+-----+  
| 1 | zhang3 | 1234567 |  
| 2 | li4 | 2345 |  
+-----+-----+-----+  
2 rows in set (0.00 sec)  
  
mysql> delete from student where name="zhang3";  
Query OK, 1 row affected (0.11 sec)  
  
mysql> select * from student;  
+-----+-----+-----+  
| id | name | phone |  
+-----+-----+-----+  
| 2 | li4 | 2345 |  
+-----+-----+-----+  
1 row in set (0.00 sec)
```

以上在 mysql 客户端展现如何操作数据库, 如何直接在 Java 应用程序中操作数据库呢? 此时就需要利用 JDBC。JDBC 提供一系列接口, 使得直接在 java 应用程序中, 就能调用执行 sql 语句。

如何使用 JDBC?

使用 JDBC 的主要步骤如下:

1. 加载 JDBC 驱动;

Java 语句: `Class.forName("com.mysql.jdbc.Driver");`

2. 建立与数据库的连接;

Java 语句: `Connection con = DriverManager.getConnection(url, userName, password);`

其中, url 指定要连接的数据库名, userName 表示数据库用户名, password 与当前用户相对应的密码;

3. 使用 SQL 语句进行数据库操作; 对数据库的操作主要分为数据更新和数据查询;

4. 释放资源;

数据库操作完毕后，关闭数据库连接以及释放相关资源;

```
conn.close();
```

例 3: 通过 JDBC 为用户 user1 创建名为 test 的数据库，并在其中创建 student 数据表，再插入 zhang3 和 li4 两位学生的信息，然后顺序查询两个学生的信息，最后删除学生 zhang3 的条目。

具体使用的 java.sql.* 的有关语句，可以先查询官方 API 文档中有关 java.sql 部分内容或者参考有关书籍。本文使用的 jdbc 版本为 jdbc-5.1.22，文件是 jar 包，与 JUnit 的安装配置过程一样。请读者先自行完成案例 3，再对照 database 文件中的 DataTest.java 示例代码。下面对示例代码中部分内容予以说明：

```
public static final String user = "user1";  
public static final String userpass = "12345";
```

此处的 user1 及对应的 password 是以 root 用户登录 mysql 后创建的。preparedStatement 的下列用法，有助于批量化的插入数据，只要将 set 有关语句放在循环中执行。

```
String cmdStr = "insert into student (id, name, phone) values (?, ?, ?)";  
try{  
    cmd = con.prepareStatement(cmdStr);  
    cmd.setString(1, "1");  
    cmd.setString(2, "zhang3");  
    cmd.setString(3, "1234");  
    cmd.executeUpdate();  
}....
```

下面的语句表示访问本地数据库，如果要访问远程，就要填写相应的 url 地址。

```
public static final String dburl = "jdbc:mysql://localhost:3306";
```

JDBC 的使用基本参照手册即可，多是约定的操作，不是难度所在，真正的难点在于结合特定业务的数据库设计，以及如何生成统计数据。

练习题:

问题 1:

如下是某公司的两个 SQL 数据表描述:

1. 部门表[department]: 包含两个字段: id, name

编号 (主键) 名称

1	财务
2	销售
3	客服

2. 员工表[employee] : 包含三个字段: id, name, id_department

编号 (主键) 姓名 所属部门编号 (外键)

1	张三	1
2	李四	1
3	王五	3
4	赵六	3

使用一个 SQL 语句输出如下的统计表：

编号（主键）	名称	员工数目
--------	----	------

1	财务	2
---	----	---

2	销售	0
---	----	---

3	客服	2
---	----	---

参考答案：

```
select d.*, e.isnull(num, 0) num from department d
```

```
left join
```

```
(
```

```
    select id_department, count(*) num from employee group by id_department
```

```
)e on e.id_department=d.id
```

问题 2：

请打开教务网站，思考学生信息，课程信息，教师信息这些数据表如何设计以及数据表之间如何关联。

问题 3：

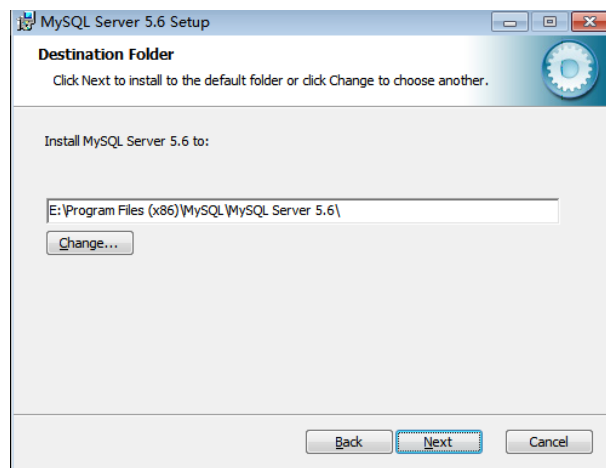
请打开淘宝网页，思考商品信息，我的订单，我的购物车等数据表如何设计？

附录 1：

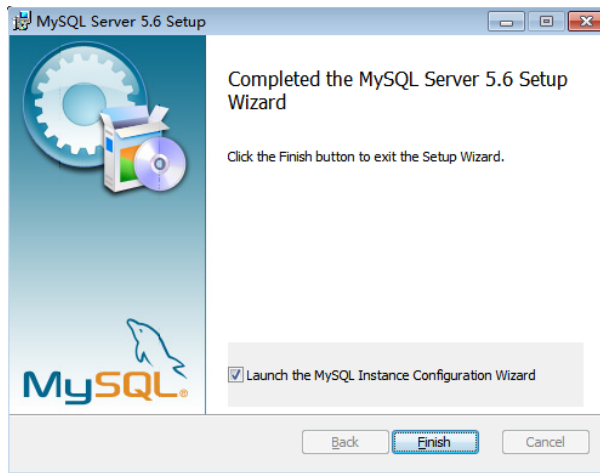
我们安装的 mysql 版本是 mysql-essential-5.6.0-m4-win32，就安装的每个步骤，说明每个对应选项的内容。

step1: License Agreement 打钩，next；

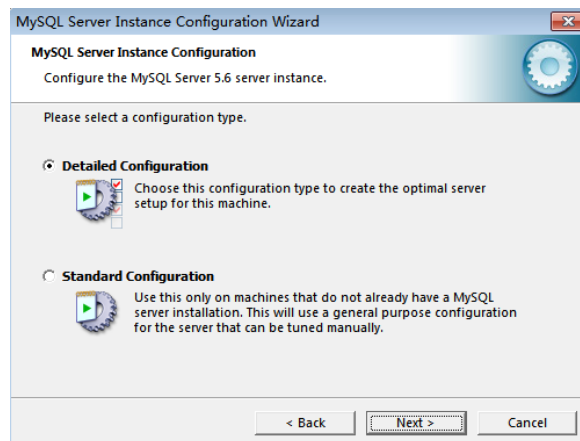
step2: 选择安装目录，可以点击 change 自主修改，比如 E:\Program Files (x86)\MySQL\MySQL Server 5.6\



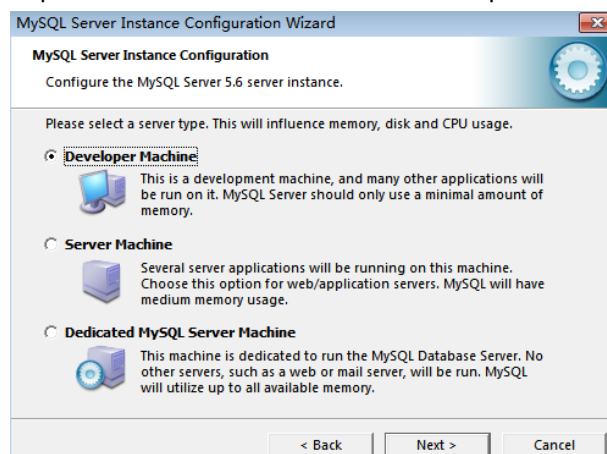
step3: 点击 install，安装结束后，打钩 Configuration Wizard，开始配置数据库；



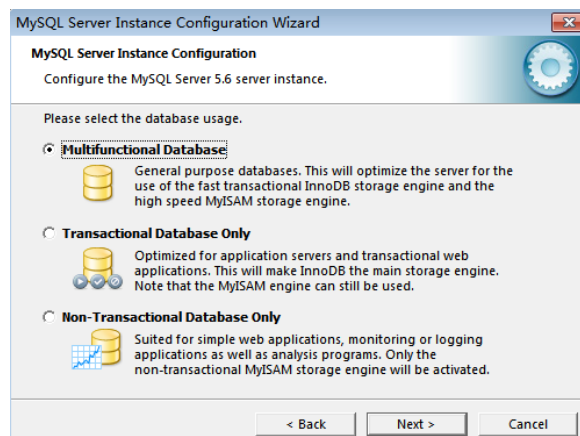
step4: 配置类型选择"Detailed Configuration"选项;



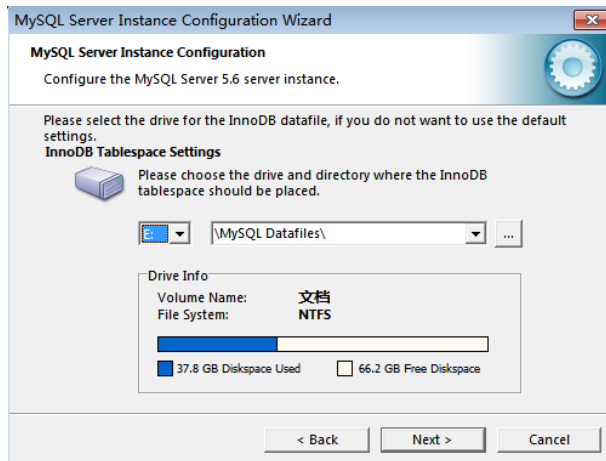
step5: 机器类别选择, 开发模式"Developer Machine";



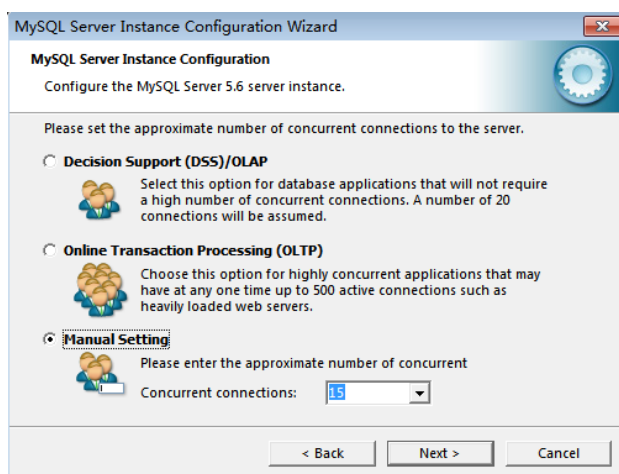
step6: 选择"multifunctional database", next;



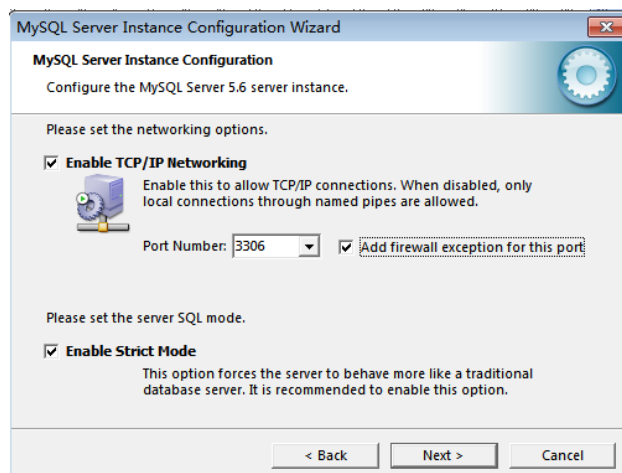
step7: 设置 tablespace 的目录到 E 盘的\MySQL Datafiles\;



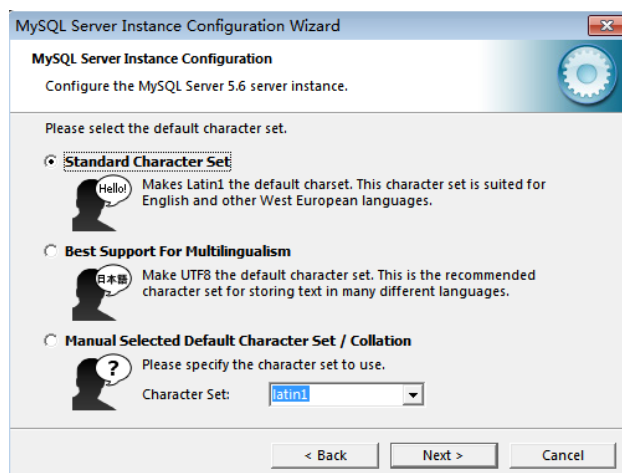
step8: Server Instance 设置为 Manual Setting 默认数值 15;



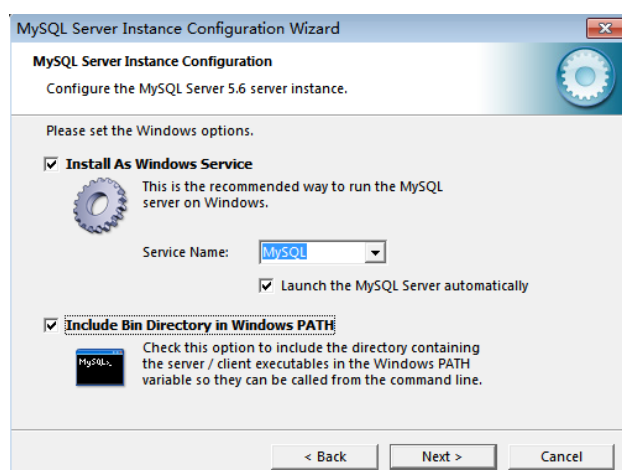
step9: 数据库的端口默认为 3306;



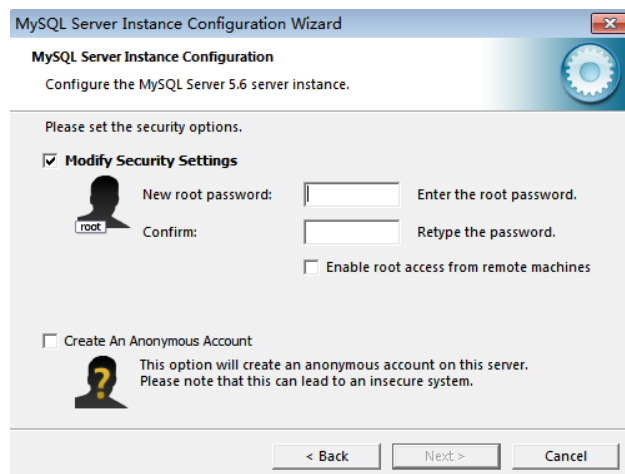
step10: character set 默认设置为 “Standard Character Set”; g



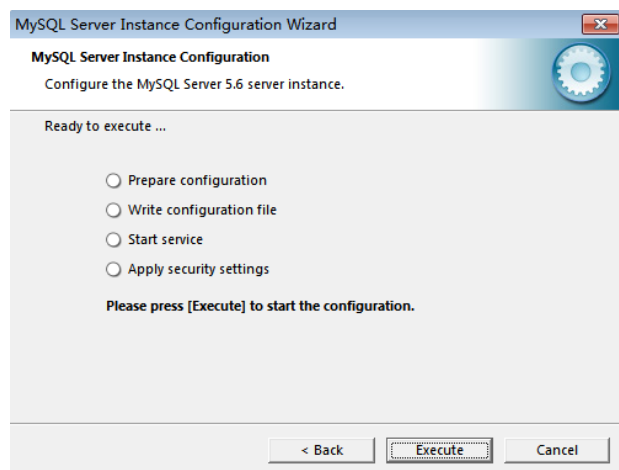
step11: Service 名称默认为 MySQL，并且将可运行程序的目录包含在 PATH 目录中;



Step12: 数据库根用户的密码设置；也即安装数据库后的第一个用户的密码设定，类似于 windows 安装时的 root 用户密码设定；



Step13: 数据库配置设定完毕后，点击 Execute 后，进行配置。



参考文献：

1. Java 编程手记-从实践中学习 Java。欧二强等编著。 清华大学出版社。

4. Java 与异常

什么是异常？

异常发生的原因是 Java 执行过程中发生干扰程序正常执行流的事件，程序通过生成并且抛出 `Exception` 对象告知系统出现意外事件。对象中的方法生成并且抛出异常，而系统则负责捕捉并且将其交与特定的 `Exception handler` 处理。如果没有合理的 `Exception handler` 来处理该异常，则程序终止，如下带有除零操作的类：

`ZeroDiv.java`:

```
public class ZeroDiv{
    public void div(int a, int b){
        int c = a/b;
        System.out.println("the result = "+c);
    }
    public static void main(String[] args){
        ZeroDiv zd = new ZeroDiv();
        zd.div(10, 0);
    }
}
```

`ZeroDiv.java` 编译正常通过，但执行时由于异常没有得到处理而终止：

```
C:\Users\tfzhang\Desktop\Exception>java ZeroDiv
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ZeroDiv.div<ZeroDiv.java:5>
    at ZeroDiv.main<ZeroDiv.java:11>
```

Java 异常机制的两个要素：

1. 将可能发生异常的操作放置在 `try{}语句`中，并且提供对应的 `Exception handler`（异常处理方法）；
2. 将可能产生异常的方法用关键词 `throw` 声明，明确抛出何种类型的异常；

`Exception Handler` 由三个代码块构成：`try`、`catch` 和 `finally`。`try` 块之后一定要跟上 `catch` 块。`try` 块限定异常捕捉的范围，`catch` 则是捕捉并且处理异常。

```
try {
    ...
} catch (ExceptionType1 name) {
    ...
} catch (ExceptionType2 name) {
    ...
}
```

`try` 之后可以跟多个的 `catch` 块，并且一般而言，`ExceptionType2` 包含 `ExceptionType1`，也即 `ExceptionType1` 定位更具体更精确。上述两个 `catch` 块也可以使用或符号合并：

```
try {
} catch (ExceptionType1 name / ExceptionType2 name) {
```

```
}
```

finally 块跟在 **catch** 块之后，而且 **try** 块执行时发生异常不能完全执行时，**finally** 块一定会被执行。所以，回收资源的事务都放在 **finally** 块中被执行。

finally 不被执行的例外情况有两种：

1. java 虚拟机在执行 **try** 或者 **catch** 块时退出，则 **finally** 块不定被执行；
2. 当线程在执行 **try** 或者 **catch** 块时被挂起或者杀死，则 **finally** 块不一定被执行；

我们将上述的 **try-catch-finally** 语句应用于 **ZeroDiv** 案例：

```
public class ZeroDiv{
    public void div(int a, int b){
        int c = 0;
        try{
            c = a/b;
        }catch(ArithmeticException e){
            System.out.println("can not divide zero");
        }catch(Exception e){
            System.out.println("some other errors than dividing zero");
        }finally{
            System.out.println("please try again");
        }
        System.out.println("the result = "+c);
    }

    public static void main(String[] args){
        ZeroDiv zd = new ZeroDiv();
        zd.div(10, 0);
        System.out.println("-----");
        zd.div(10, 1);
    }
}
```

如何让方法抛出异常？

方法声明异常的格式如下：

```
public void 函数名 throws ExceptionType1, ExceptionType2{
    ...
}
```

方法执行过程中，扔出异常的代码如下：

```
throw someThrowableObject;
```

任何一个被抛出的异常类都继承自 **Throwable** class。

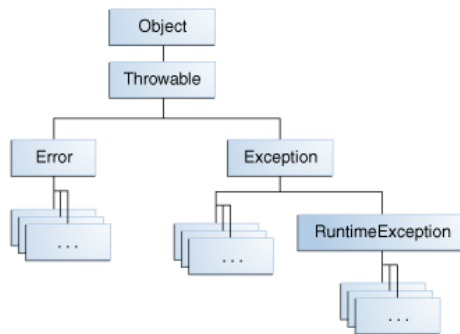


图 1 异常类层次

如下是摘自 `java.util.Stack` 中的示例代码：

```
public synchronized E pop() {
    E      obj;
    int    len = size();
    obj = peek();
    removeElementAt(len - 1);
    return obj;
}

public synchronized E peek() {
    int    len = size();
    if (len == 0)
        throw new EmptyStackException();
    return elementAt(len - 1);
}
```

语句"`throw new EmptyStackException();`"，创建一个 `EmptyStackException` 的异常，并且抛出；其中 `EmptyStackException` 是 `java.util` 包中的一个类。

如何自定义异常？

根据 `Throwable` 的类层次图，所有异常类的祖先是 `Exception`，所以，自定义的类也需要继承自 `Exception`。示例程序要求用户，从终端输入学生的成绩 0~100，如果出现超越范围的数据，抛出异常。

Score.java:

```
import java.util.Scanner;
```

```
public class Score{
```

```
    public static void main(String[] args){
        int scores[] = new int[10];
        Scanner scan = new Scanner(System.in);

        for(int i=0; i<scores.length; i++)
        {
            System.out.print("the score of " + (i+1) + "th student:");
            scores[i] = scan.nextInt();
        }
    }
}
```

```
    }  
}
```

当用户输入学生的分值出错，比如大于 100，程序就应该抛出异常，而异常只要说明自身即可，所以只要携带一条信息内容。

自定义的 Exception 类如下：

```
class MyException extends Exception{  
    public MyException(String s){  
        super(s);  
    }  
}
```

修改后的 Score.java 如下：

```
import java.util.Scanner;
```

```
public class Score{  
    public static void main(String[] args){  
        int scores[] = new int[10];  
        Scanner scan = new Scanner(System.in);  
  
        try{  
            for(int i=0; i<scores.length; i++){  
                {  
                    System.out.print("the score of "+(i+1)+"th student:");  
                    scores[i] = scan.nextInt();  
                    if(scores[i] < 0 || scores[i]>100){  
                        throw new MyException("score <0 or score > 100");  
                    }  
                }  
            }  
        }catch(MyException e){  
            System.out.println(e);  
        }  
    }  
}
```

使用异常带来的好处

为什么要使用 Exception？一、保持代码的整洁：如下关于 readFile 的伪代码：

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```

上述的 1~5 步，每步都容易出现错误，如果使用传统的 C 语言来定位错误，则代码如下：

```

errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
    }
    close the file;
    if (theFileDidntClose && errorCode == 0) {
        errorCode = -4;
    } else {
        errorCode = errorCode and -4;
    }
} else {
    errorCode = -5;
}
return errorCode;
}

```

代码看上去十分的繁复，使用异常得到的代码如下，既能定位错误的出处，也能保持代码整洁：

```

readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    }
}

```

```

    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}

```

异常的好处二：方便地将错误码上传到目标方法。如下三个方法的例子中：

```

method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call readFile;
}

```

Method1 调用 Method2，Method2 调用 Method3，Method3 中调用方法 `readFile`；Method1 对 `readFile` 中发生的错误感兴趣。C 语言的处理方式：

```

method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}

```

Method2 和 Method3 虽然对 `readFile` 可能发生的错误不感兴趣，但还是要将 `errorCode` 往上一级方法传递，十分麻烦。而采用异常的处理方式：

```

method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

```

```

method2 throws exception {
    call method3;
}

```

```

method3 throws exception {
    call readFile;
}

```

method2 和 method3 只需要申明会抛出异常即可。

异常的好处三、以不同的粒度灵活处理错误。就文件操作而言，只想处理文件打开错误的话，那么对应的语句如下：

```

catch (FileNotFoundException e) {
    ...
}

```

因为 `FileNotFoundException` 类没有被其他类继承，所以此 `Exception Handler` 只能处理文件没找到的异常。而如果想处理文件有关的错误，则只要捕捉的是 `IOException` 即可。

```

catch (IOException e) {
    ...
}

```

通过捕捉同一类错误的不同对象，达到灵活处理错误的目的。

练习：

问题 1：考察下面两段代码，确定每段代码中的 `finally` 语句是否执行？

代码 1：

```

public class ExceptionTest01{
    public static void main(String[] args){
        try{
            System.out.println("in try");
            int i = 1/0;
        }catch(ArithmeticException e){
            System.exit(0);
            e.printStackTrace();
        }finally{
            System.out.println("in finally");
        }
    }
}

```

代码 2:

```
public class ExceptionTest02{
    public static void main(String[] args){
        int n = fun(2);
        System.out.println(n);
    }
    public static int fun(int i){
        try{
            int m = i/0;
            return (i+1);
        }catch(ArithmeticException e){
            return (i+2);
        }finally{
            return (i+4);
        }
    }
}
```

问题 2:

如下代码中有三个 **print** 语句，请确定哪几个会被打印以及打印的先后顺序？

```
public class ThrowTest{
    public static void main(String[] args){
        try{
            System.out.println("1: before throw_do");
            throw_do();
            System.out.println("2: after throw_do");
        }finally{
            System.out.println("3:finally executing");
        }
    }
    public static void throw_do(){
        throw new ArithmeticException();
    }
}
```

5. Java 与正则表达式

什么是正则表达式?

正则表达式是描述和处理字符串的工具，可以对字符串进行查找、替换、分割等操作。很多文本编辑器与编程语言都支持正则表达式，java 也不例外。虽然正则表达式并不复杂，但要灵活应用还需要基本知识与案例的操练。首先，来看两个应用正则表达式的简单案例。

案例 1: String s="132ffdsaf3242fdsfsadfsadf232",使用正则表达式从中找出所有数字 1323242232。

案例 1 主要目的是将字符串 s 中的英文字符删除，只剩下数字。一般的做法是依次处理 String s 中的每个字符，如果不是英文字符，则删除，最后剩下的就是数字。相比而言，采用正则表达式的方式就要简单很多，代码如下：

```
public class Demo1{
    static String s="132ffdsaf3242fdsfsadfsadf232";
    public static void main(String[] args){
        String str = s.replaceAll("[^0-9]", "");
        System.out.println(str);
    }
}
```

replaceAll()是一个替换操作，方法的第二个参数表示空，也就把第一个参数表示的字符用空字符替换，即删除。第一个参数"[^0-9]"是一个正则表达式，什么意思呢？符号 0-9 表示 0 到 9 的数字，[]表示出现在括号中的任意字符，[^]表示括号中的任意字符的取反，最终"[^0-9]"表示出现的任意非数字字符。所以，语句 replaceAll("[^0-9]", "")表示将任意非数字的字符删除。

案例 2: String str="2008-03-23 12:30:01",使用正则表达式将字符串变成 20080323123001。

案例 2 的目的是将其中任意非数字的字符删除，当然可用案例 1 中的正则表达式解决。我们尝试使用正则表达式中其他的表达方式来解决问题，代码如下：

```
public class Demo2{
    public static void main(String[] args){
        String str = "2008-03-23 12:30:01";
        String str2 = "";
        String[] res = str.split("\\D");
        for(int i=0; i<res.length; i++){
            str2 += res[i];
        }
        System.out.println(str2);
    }
}
```

语句 str.split("\\D")中，表示 str 按给定的参数符号为界，分割(split)为多个子字符串。以字符串 s="2008-03-23"，如果调用 s.split("-")，字符串 s 就以 "-" 为界，将 s 分割为三个子字符串 "2008", "03" 和 "23"。但代码中的 str，难以以具体的字符为界，因为除了 "-" 间隔，还有空格和 ":" 等。此时就需要使用正则表达式 "\\D"，"\\D" 表示非数字字符，而 "\\" 表示下一个字符，比如已知 "\\n" 表示匹配字符 n，而 "\\n" 表示匹配换行符 "\n"；所以 "\\D" 表示匹配非数字字符。

表 1 是简单的一类表达式，左侧是正则表达式，右侧是功能描述。

Construct	Description
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z, or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

表 1 字符正则符

Construct	Description
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\r\f]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

表 2 字符正则符

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly n times
X{n,}	X{n,}?	X{n,}+	X, at least n times
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times

表 3 数量正则符

Boundary Construct	Description
^	The beginning of a line
\$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input

\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

表 4 边界正则符

案例 3: 假设我们要搜索美国的社会安全号，格式是 999-99-9999，对应的正则表达式：

`[0-9]{3}\-[0-9]{2}\-[0-9]{4}`

注：

`[0-9]`：0 到 9 的数字；

`{3}`：出现 3 次；

`\-`：匹配-；

如果“-”也可以不出现，比如 999999999，只需要增加“?”符，表示或者没有：

`[0-9]{3}\-?[0-9]{2}\-?[0-9]{4}`

案例 4: 匹配生日格式“June 26, 1951”，对应的正则表达式：

`([a-z]+) \s+[0-9]{1,2},\s*[0-9]{4}`

Java 中使用正则表达式：

Java 关于正则表达式的包 `java.util.regex`，主要包含三个类：`Pattern`，`Matcher` 和 `PatternSyntaxException`。`Pattern` 类用于创建一个正则表达式，创建一个匹配模式，它的构造方法私有，不可以直接创建，需要通过 `Pattern.compile` 方法创建一个正则表达式。比如案例 1 为例：

```
import java.util.regex.*;
public class T1{
    public static void main(String[] args){
        Pattern p = Pattern.compile("\\D");
        System.out.println(p.pattern()); //输出为"\D";
    }
}
```

`Pattern` 类中主要的方法有 `matcher`，并且需要搭配 `Matcher` 类一起使用，因为 `matcher` 方法返回 `Matcher` 类。而 `Matcher` 类中常用的方法是 `find()`，`group()`，`start()`和 `end()`，对应的说明读者可自行查阅 Java API 文档。下面是官方文档中的例子：

```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest{

    public static void main(String[] args){
        Console console = System.console();
        if (console == null) {
            System.err.println("No console.");
            System.exit(1);
        }
    }
}
```

```

while (true) {
    Pattern pattern =
        Pattern.compile(console.readLine("%nEnter your regex: "));
    //System.out.println(pattern.pattern());

    Matcher matcher =
        pattern.matcher(console.readLine("Enter input string to search: "));

    boolean found = false;
    while (matcher.find()) {
        console.format("I found the text" +
            "\"%s\" starting at " +
            "index %d and ending at index %d.%n",
            matcher.group(),
            matcher.start(),
            matcher.end());
        found = true;
    }
    if(!found){
        console.format("No match found.%n");
    }
}
}

```

主要可以划分为两个主要步骤:

1. 通过终端输入正则表达式:

```
Pattern pattern = Pattern.compile(console.readLine("%nEnter your regex: "));
```

2. 基于输入正则表达式的模式, 查找输入字符串中是否出现对应的模式, 如果出现则输出其对应的起始点和终点位置; 如果多个匹配, 则输出多次;

```
Matcher matcher = pattern.matcher(console.readLine("Enter input string to search: "));
```

`matcher.find()`用于查找下个匹配的字符串, `matcher.group()`用于输出匹配的字符串。以案例 2 为例, 输出其中每个数字:

```

Enter your regex: \d+
Enter input string to search: 2008-03-23 12:30:01
I found the text "2008" starting at index 0 and ending at index 4.
I found the text "03" starting at index 5 and ending at index 7.
I found the text "23" starting at index 8 and ending at index 10.
I found the text "12" starting at index 11 and ending at index 13.
I found the text "30" starting at index 14 and ending at index 16.
I found the text "01" starting at index 17 and ending at index 19.

```

下面的数量正则表达式:

```

Enter your regex: a{2}
Enter input string to search: afdsadfafdaafd
I found the text "aa" starting at index 10 and ending at index 12.

```

因此上述程序可以作为正则表达式的测试工具。

案例 5: 编写一 java 程序, 将 C 语言的常数声明语句修改为对应的 java 常数声明语句, 比如:

C 语言: `#define PI 3.141592654`

转化为:

Java: `private static final double PI = 3.141592654`

假设 cl.c 是包含常数的 C 语言文件, 转化为对应的 java 版本 jl.java。

cl.c:

```
#define PI 3.141592654
#define EPOCH 85
#define EPSILONg 279.611371 /*solar ecliptic long at EPOCH */
#define RHOG 282.680403 /* solar ecliptic long of perigee at EPOCH */
#define ECCEN 0.01671542 /* solar orbiteccentricity */
#define lzero 18.251907 /* lunar meanlong at EPOCH */
#define Pzero 192.917585 /* lunar mean long of perigee at EPOCH */
#define Nzero 55.204723 /* lunar meanlong of node at EPOCH */
```

以下是供参考的代码行, 请读者自己想出其他的正则表达式方案; 另外, 请读者补上打开文件 cl.c, 逐行处理, 并且写文件 jl.java 的辅助代码。

C2Java.java:

```
public class C2Java{
    public static void main(String[] args){
        String str = "#define Pzero 192.917585 /* lunar mean long of perigee at EPOCH */";
        String jstr = "";

        String[] s = str.split("[\\s\\t]+");
        s[0] = "private static final double "; //overwrite #define;

        jstr = s[0] + s[1] + "=" + s[2] + "; //";
        for(int i=3; i<s.length; i++)
        {
            jstr = jstr + " " + s[i];
        }
        System.out.println(jstr);
    }
}
```

练习:

问题 1: 读取一个文件, 输出该文件中所有的 11 位手机号码。测试文件见附件。

<http://www.cnblogs.com/0201zcr/p/4994724.html>

问题 2: 读取一个文件, 输出该文件中所有的电子邮箱地址。测试文件见附件。

练习 3: 完成案例 5 的完整代码。

6. Java 多线程

什么是线程？

程序的执行本质上是占用处理器的运行时间，线程可以理解为能够申请并且占用处理器时间的基本单元。因为每个 Java 程序都要占用处理器运行自身，所以每个 Java 程序至少是一个线程，绝大多数情况一个 Java 程序包含大量线程。举例而言，一个 Java 游戏在运行时，假如画面持续更新由一个线程负责，那么音乐的播放就肯定是另外一个线程负责。所以，当你在游戏设置界面中开启音乐键后，就会产生一个新的线程并且执行音乐播放的相关代码。平时我们写的带有 main 方法的类基本上是一个线程的程序。

如何创建线程？

线程创建的两种方法：

基于继承：

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

基于接口：

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

两者都要实现 run() 方法，要求线程完成的工作一般都实现于此方法中。调用 start 方法可以触发线程的 run 方法。上述两种实现线程的方法不同，也自带不同的优缺点。实际编程开发中，多以 Runnable 接口来实现线程，主要原因是实现接口相比继承有如下的优点：

1. 可以避免由于 Java 的单继承性带来的限制；
2. 代码共享，多线程可以共享同一资源，比如数据；

下面买火车票的为例，说明两者间的差别。假设要购买 15 张火车票，如果要求三个人 A，B，C 人，对应三个线程来实施购买行为，如果采用继承的方式来实现，只能如下：

```
public class Ticket extends Thread{
    private int num = 5;
```

```

    public Ticket(String name){
        this.setName(name);
    }

    public void run(){
        int i = 1;

        while(num>0){
            System.out.println(this.getName()+" buy ticket "+i++);
            num--;
        }
    }
}

public class TestTic{
    public static void main(String[] args){
        Thread a = new Ticket("a");
        Thread b = new Ticket("b");
        Thread c = new Ticket("c");
        a.start();
        b.start();
        c.start();
    }
}

```

现在我们要考虑总体购买时间最短，上述的这种实现就存在问题。因为可能 **a** 购买票的窗口发生意外，比如售票机器发生故障，**a** 只能漫长地等待；而 **b** 和 **c** 不知道发生该情况，早早地买好自己份内的 5 张车票，那么购买 15 张车票这一任务的时间就被 **a** 拖得很迟。这一情况发生的原因：各方对当前已经购票数这一信息不能共享。采用接口实现购买线程就可以解决这一问题：

```

public class Ticket implements Runnable{
    private int num = 15;
    public void run(){
        while(num >= 0){
            num--;
            System.out.println("buy ticket "+(15-num));
        }
    }
}

public class TestTic{
    public static void main(String[] args){
        Ticket tc = new Ticket();

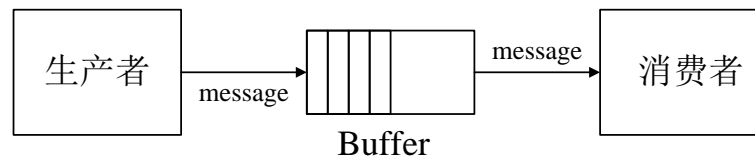
        new Thread(tc).start();
        new Thread(tc).start();
    }
}

```

```
        new Thread(tc).start();
    }
}
```

显而易见，3 个线程共享对象 `tc`，也即共享数据 `num=15`，只要还没有买到 15 张，那么三个线程就会一直忙碌，不会出现继承实现中出现的一个线程还在忙碌，另外两个线程提前完成的情况。采用接口实现的多线程自带共享资源的特点。上述接口实现的代码还存在一小问题，会出现买 16 张票的情况。因为一线程在判断 `num=1` 大等于 0 时，另外一个线程已经将 `num` 减去 1 变为 0，那么当前面线程再减 1 时，`num` 的值变为-1。请读者思考如何解决这一问题。

线程之间有时候需要同步合作，需要使用 `wait`，`notify` 以及 `synchronized` 等关键词。下面以另一个经典案例生产者与消费者来说明线程之间的同步。



生产者产生 `message`，并且一次一条地放入到 `buffer` 中；同时，生产者唤醒消费者来处理 `buffer` 中刚被放入的信息。因为 `buffer` 对生产者和消费者都可见（**java 程序设计中，“可见”这个词要好好理解，什么是可见？什么是不可见？**），所以两者也就可以通过 `buffer` 对象同步。只要 `buffer` 非空，生产者就可往其中填入字符；反之，`buffer` 空的时候，应该处于等待状态。DIY(do it yourself)代码如下：

```
while(!empty){
}
```

但是循环内部什么都不干，浪费 `cpu` 时间；常见的处理方式是挂起当前线程，让出 `cpu` 时间，代码如下：

```
while(!empty){
    wait();
}
```

代码调用 `wait` 方法后，挂起当前线程；只有当其他线程调用同一对象的 `notify()` 或者 `notifyAll()` 方法后，挂起的线程才有机会重新启动。自然地，只有 `consumer` 处理完数据后，`buffer` 重新处于 `empty` 状态，才可能调用 `notify()` 方法，让 `producer` 线程有机会重新启动。`buffer` 的具体代码如下：

```
public class Buffer{
    private boolean empty = true;
    private String message;

    public void put(String m){
        try{
            while(!empty){
                wait();
            }
        }catch(InterruptedException e){
        }
        message = m;
        empty = false;
    }
}
```

```

        notifyAll();
    }

    public String take(){
        try{
            while (empty){
                wait();
            }
        }catch(InterruptedException e){
        }
        empty = true;
        notifyAll();
        return message;
    }
}

```

`put()`方法主要被生产者使用，表示放入信息；而 `take` 方法则由消费者调用，取出信息，并进行处理。两个方法的处理流程基本一致，其中的逻辑并不复杂，请读者自己跟踪理解。下面是生产者的代码：

```

import java.util.Random;
public class Producer implements Runnable{
    private Buffer bf = null;
    public static String[] messages = {
        "I am here", "where are you?", "could you find me?", "Done";
    }

    public Producer(Buffer bf){
        this.bf = bf;
    }

    public void run(){
        Random rd = new Random();
        for(int i=0; i<messages.length; i++)
        {
            bf.put(messages[i]);
            try{
                Thread.sleep(rd.nextInt(3000));
            }catch(InterruptedException e){
            }
        }
    }
}

```

生产者中一共有 4 条信息，每隔一个 0~3 秒的随机时间间隔，尝试向 `buffer` 中添加信息。下面是消费者的代码：

```

import java.util.Random;
public class Consumer implements Runnable{

```

```

private Buffer bf = null;

public Consumer(Buffer bf){
    this.bf = bf;
}

public void run(){
    for(String s = bf.take(); !s.equals("Done"); s = bf.take()){
        System.out.println("message received: "+s);
    }
}
}

```

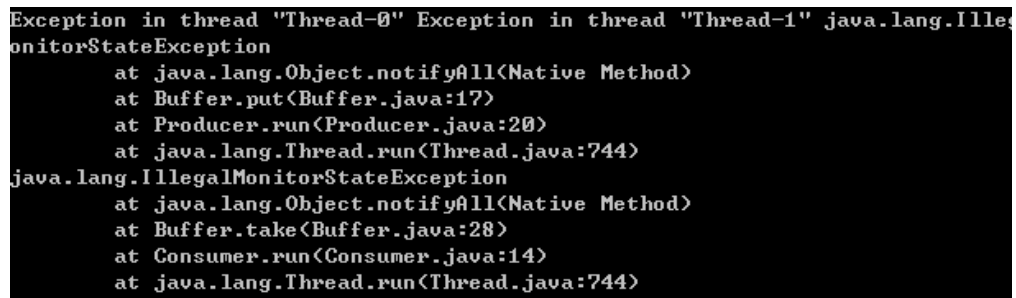
消费者同生产者一样，因为要作为线程运行，都实现 `Runnable` 接口。消费者的功能不复杂，将 `buffer` 中的数据取出并打印。最后是测试代码：

```

public class Test{
    public static void main(String[] args){
        Buffer bf = new Buffer();
        new Thread(new Producer(bf)).start();
        new Thread(new Consumer(bf)).start();
    }
}

```

但是运行过程中出现如下的错误：



```

Exception in thread "Thread-0" java.lang.IllegalMonitorStateException
    at java.lang.Object.notifyAll(Native Method)
    at Buffer.put(Buffer.java:17)
    at Producer.run(Producer.java:20)
    at java.lang.Thread.run(Thread.java:744)
Exception in thread "Thread-1" java.lang.IllegalMonitorStateException
    at java.lang.Object.notifyAll(Native Method)
    at Buffer.take(Buffer.java:28)
    at Consumer.run(Consumer.java:14)
    at java.lang.Thread.run(Thread.java:744)

```

出现这种错误，我们该如何处理？首先，根据提示信息，发现错误发生在 `notifyAll()` 方法的调用。因此，我们查询 Java API 文档，有如下的说明：

This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor. Throws: `IllegalMonitorStateException` - if the current thread is not the owner of this object's monitor.

说明只要线程不是当前对象的控制权持有者，那么调用 `notifyAll()` 方法时，就会抛出 `IllegalMonitorStateException`。问题转化为我们的代码哪里书写有误，使得当前线程不是对象控制权的持有者？继续细看，发现如下文字：

A thread becomes the owner of the object's monitor in one of three ways:

1. By executing a synchronized instance method of that object.
2. By executing the body of a synchronized statement that synchronizes on the object.
3. For objects of type `Class`, by executing a synchronized static method of that class.

Only one thread at a time can own an object's monitor.

暂时忽略关于 Class 类的第 3 条，因此当前线程获得对象的控制权，主要有两种方式：执行对象的某个同步实例方法和执行对该对象加同步锁的同步块。我们可以采用第一种方法，只要在方法前加上 `synchronized` 关键词，就可以将 Buffer 的 `put()`和 `take()`方法转化为同步方法，再由 `Producer` 和 `Consumer` 调用，自然成为同步实例方法。读者可以自己尝试下，编译运行正常。如果采用第 2 种做法，又该怎么做呢？只要对 `this` 引用，增加对应的 `synchronized statement` 即可，具体的修改如下：

```
public void put(String m){
    synchronized(this){
        try{
            while(!empty){
                wait();
            }
        }catch(InterruptedException e){
        }
        message = m;
        empty = false;
        notifyAll();
    }
}

public String take(){
    synchronized(this){
        try{
            while (empty){
                wait();
            }
        }catch(InterruptedException e){
        }
        empty = true;
        notifyAll();
    }
    return message;
}
```

综上，我们可以发现 `synchronized statement` 中的代码才有权利竞争一个对象的控制权，否则连参与竞争的资格都没有，即使你代码里出现了 `wait()`和 `notify()`这些语句。下面在看一个经典的线程同步案例：线程交替打印 ABC。

问题描述：建立三个线程，A 线程打印 10 次 A，B 线程打印 10 次 B,C 线程打印 10 次 C，要求线程同时运行，交替打印 10 次 ABC。采用 `wait` 和 `notify` 方法实现。我们需要有三个同步用的对象比如 `m,n,o`，为完成交替打印功能，每个线程占用同步对象的顺序如下：

A	B	C
m	o	N
n	m	O

当 A 占据 `m` 和 `n` 时，B 只占据 `o` 而没有 `m`，所以此时 B、C 挂起，只有 A 能运行；

A 释放 m 后, B 获得机会执行; 而此时 C 还没有机会运行。按照这个顺序, 代码依次获得执行的机会。代码主要的操作步骤:

- 1、线程占据两个对象的控制权, 完成本线程的工作;
- 2、线程释放对象的控制权, 唤醒其他等待的线程, 并且让自己处于等待被唤醒的状态;

第一个步骤要求使用 `synchronized` 的语句, 使得当前线程可以竞争对象控制权; 第二个步骤, 则要求线程同时使用 `notify` 和 `wait` 方法; `notify` 方法唤醒其他等待的线程, 而 `wait` 方法则让当前线程挂起。代码的大体框架如下:

```
while(num > 0){
    synchronized(a){
        synchronized(b){
            System.out.printf(message);
            num--;
        }
    }
}
```

剩下的问题就是 `notify()` 和 `wait()` 针对哪个对象进行, 并且决定代码中的位置。可以肯定的一点, `notify` 方法在 `wait` 之前, 如果 `wait` 在前, 则 `notify` 没有机会执行, 必然造成死锁。那么 `notify` 方法由对象 a 调用还是由对象 b 调用呢? 我们来看看 `a.notify()` 和 `b.wait()` 的组合:

```
while(num > 0){
    synchronized(a){
        synchronized(b){
            System.out.printf(message);
            num--;

            a.notify();
            try{
                b.wait();
            }catch(Exception e){}
        }
    }
}
```

当代码运行到 `a.notify()` 时, 线程唤醒其他竞争 a 对象的控制权的线程, 但是由于紧接着 `a.notify()` 后面是 `b.wait()`, 当前线程挂起停止运行, 但还在 `synchronized(a)` 的作用范围之内, 所以当前线程还是没有释放对象 a 的控制权, 导致后续的线程无法执行。所以, `notify` 方法只能由对象 b 来调用。具体的代码如下:

```
while(num > 0){
    synchronized(a){
        synchronized(b){
            System.out.printf(message);
            num--;
            b.notify();
        }
        try{
```

```

        a.wait();
    }catch(Exception e){}
}
}

```

这段代码有趣的两个地方：1. `notify()`方法是对象 `a` 调用还是对象 `b` 调用？另一个问题就是：两个 `synchronized` 方法的作用范围？如果我们将代码修改如下，即两个 `synchronized` 语句的作用范围相同，那么会带来什么结果？对象 `a` 的控制权释放，但是对象 `b` 的控制权没有释放，程序无法正常地运行。

```

while(num > 0){
    synchronized(a){
        synchronized(b){
            System.out.printf(message);
            num--;
            b.notify();
            try{
                a.wait();
            }catch(Exception e){}
        }
    }
}

```

由此，我们是否可以推论：当前线程调用 `notify()`方法时，表示唤醒竞争当前对象控制权的线程，但唤醒后是否有权利运行，依赖于是否能获得控制权，如果当前线程因某种原因停顿，没有完成 `synchronized` 语句，那当前线程就一直霸占控制权，唤醒的线程依然无法执行。当前线程调用 `wait()`方法则不同，不管当前线程是否执行完 `synchronized` 语句，都立即出让控制权，供其他线程竞争使用。请读者查阅官方文档，以及设计相应实验验证该推论的正确性。完成上述代码主体后，我们继续完成测试代码部分：

```

public class TestRound{
    public static void main(String[] args){
        Object a = new Object();
        Object b = new Object();
        Object c = new Object();

        new Thread(new RoundPrint("a", a, b)).start();
        new Thread(new RoundPrint("b", c, a)).start();
        new Thread(new RoundPrint("c", b, c)).start();
    }
}

```

运行后，结果虽然符合预期，但存在一问题：代码没有结束运行，请读者思考这是什么原因？



```

C:\Users\tfzhang\Desktop\threads\rr>java TestRound
abcabcacbacbacbacbacbacbacb_

```

上述做法明显的不足之处是引入额外的可见的对象，即 `Object a, b, c`，来实现线程的同步。可以想见，如果当前不只有 3 个线程而是要求 10 个线程的依次交替执行，额外的对象

操作会成为十分沉重的负担，请读者思考有没有更理想的做法，可就线程数量方便地扩展？提示：此处可采用信号量 **Semaphor** 来实现，请读者自己尝试。

线程池技术：

线程运行的时间分为初始化线程 **T1**、执行任务 **T2**、销毁线程 **T3**。如果任务数量很多，但是每个任务工作量很小，使得 $T1+T3>T2$ ，也就是说线程运行时大于 50%的时间在做与任务无关的辅助工作。能不能将 **T1** 和 **T3** 的时间降下来呢，或者说线程是否可以事先创建好，当有任务时，线程就像一种公共资源，直接拿来用即可？

线程池技术正好是解决该问题。线程池，顾名思义，事先生成一定数量的线程，集中由特定数据结构管理，当新任务到来时，分配线程执行任务；任务完成后，收回线程资源；当所有任务完成后，再统一销毁所有线程。线程池中的线程只有一次生成和销毁过程，但可以多次加载不同任务执行，每个任务分摊到的 **T1** 和 **T3** 就很小，从而提升效率。一个线程池要正常运作，需要如下的四个部分：

1. 线程池管理器(**ThreadPool**)：管理线程，创建和销毁线程；
2. 工作线程(**WorkThread**)：线程池中的线程，有任务时，加载任务执行；没任务则休眠；
3. 任务接口：每个任务必须实现的接口，以供工作线程调度任务的执行，主要规定任务的入口，任务的收尾工作以及任务的执行状态等；
4. 任务队列(**taskQueue**)：没有空闲的线程，任务需要在任务队列中等待；

需要考虑的技术点：

1. 任务队列的角色类似于生产者和消费者的 **Buffer**，可以推论，往任务队列中放入和拿出任务时需要同步；
2. 已知工作线程在 **taskQueue** 为空时，处于休眠状态；而任务队列非空时，则处于工作状态；那么当任务队列非空时，谁来唤醒线程？每次加入新任务时，调用 **notify()**，唤醒当前休眠的线程；另外，也可以在线程调用 **wait** 方法进入休眠时给定时间参数，休眠时间到，线程也会醒来；
3. 任务如何由工作线程来执行？任务要实现 **Runnable** 接口，而线程中包含 **Runnable** 变量 **r**，将任务赋值给 **r**，再执行 **r.run()**，即工作线程开始执行任务的 **run** 方法，也即任务加载到工作线程上；
4. 如何结束工作线程？当所有任务执行完毕，通过改变工作线程的状态位，跳出 **run()** 方法中的循环；

首先，工作线程需要如下的要素：

1. 根据技术点 4 的要求，每个工作线程是一个线程，包含状态位 **isRunning**，初始状态位 **true**；
2. 执行任务之前，工作线程启动，执行 **run()** 方法；**run** 方法的主要逻辑包括：

只要当前 **isRunning** 为 **true**，

- a. 当前的 **isRunning** 为 **true** 且任务队列为空，线程调用 **wait** 方法，休眠；
- b. 任务队列非空，从任务队列中取任务，执行任务；

工作线程 **WorkThread** 的代码如下：

```
class WorkThread extends Thread {  
    private boolean isRunning = true; //true 表示工作状态;  
  
    public void run() {
```

```

        Runnable r = null;
        while (isRunning) {
            synchronized (taskQueue) {
                while (isRunning && taskQueue.isEmpty()) { // 队列为空
                    try {
                        taskQueue.wait(20);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                if (!taskQueue.isEmpty())
                    r = taskQueue.remove(0); // 取出任务
            }
            if (r != null) {
                r.run(); // 执行任务
            }
            finished_task++;
            r = null;
        }
    }

    public void stopWorker() {
        isRunning = false;
    }
}

```

代码中的关键的语句:

```

        r = taskQueue.remove(0);
        r.run();

```

对应的正是技术点 3，任务从任务队列中取出并赋值于 `r`，再启动任务中的 `run` 方法执行。注意 `synchronized` 语句包含的范围。线程池管理器(`ThreadPool`)首要包含生成和销毁线程两个部分。线程池根据指定数值生成对应数量的工作线程，并且启动每个工作线程，具体代码如下:

```

private ThreadPool(int worker_num) {
    ThreadPool.worker_num = worker_num;

    workThrad = new WorkThread[worker_num];
    for (int i = 0; i < worker_num; i++){
        workThrad[i] = new WorkThread();
        workThrad[i].start(); // 开启线程池中的线程
    }
}

```

根据 `WorkThread` 的代码，只要 `isRunning` 由 `true` 变为 `false`，则 `WorkThread` 消亡，所以对应的销毁线程池的代码如下:

```

public void destroy() {

```

```

        while (!taskQueue.isEmpty()) { // 如果还有任务没执行完成，就先睡会吧
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 工作线程停止工作，且置为 null
        for (int i = 0; i < worker_num; i++) {
            workThrad[i].stopWorker();
            workThrad[i] = null;
        }
        threadPool=null;
        taskQueue.clear();// 清空任务队列
    }

```

需要注意线程池的 **destroy** 方法可一直处于启动状态，只不过 **taskQueue** 处于非空状态时，**destroy** 方法会周期性地处于 **sleep** 状态。

线程管理池的另一个重要任务是将任务添加到任务队列中，添加过程使用方法的重载，根据输入参数的不同，调用不同方法。另外，添加任务后，需要调用 **notify** 方法，唤醒可能处于休眠的工作线程。

```

    public void execute(Runnable task) {
        synchronized (taskQueue) {
            taskQueue.add(task);
            taskQueue.notify();
        }
    }

    public void execute(Runnable[] task) {
        synchronized (taskQueue) {
            for (Runnable t : task)
                taskQueue.add(t);
            taskQueue.notify();
        }
    }

```

execute 方法将输入的单个任务或者任务队列，添加到任务队列中，并且调用 **notify()** 方法唤醒工作线程（此处是否该使用 **notifyAll()** 方法？）。根据 **WorkThread** 中的代码，每个具体的任务只要实现 **Runnable** 接口即可，代码结构如下：

```

class Task implements Runnable {
    public void run() {
        ....//具体任务的代码写在此处;
    }
}

```

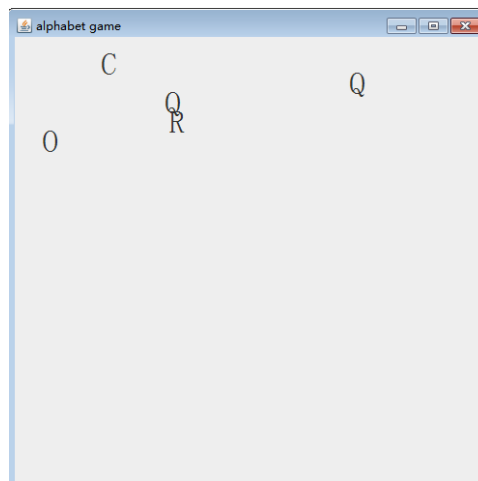
完整的代码见文件夹 **threadpool**。

因为只是从原理层面介绍，所以上述的例子相对较简单。读者可以进一步去阅读、应用、

理解 Java 自带的线程池技术，对应的类主要是 `java.util.concurrent` 中的 `ThreadPoolExecutor` 类。建议先使用，然后根据原理层面的理解，再去阅读源代码；如果源代码阅读有困难，可基于关键字“深入理解 Java 之线程池”搜索，自然能找到相应的技术文章。

线程的应用：

线程在实际编程工作中十分常用，比如贪吃蛇游戏，此处我们通过一个简单的打字游戏来看看线程在实际场景中的应用。游戏的界面如下：



游戏界面中英文字符随机落下，当敲击对应字符的按键后，字符消失；如果某字符下落到界面最下方还没有被敲击，则自动消失。字符通过将 `text` 附加到 `JLabel` 组件上实现，具体的代码如下：

```
JLabel show = new JLabel();
String pst;
show.setFont(new java.awt.Font("宋体",Font.PLAIN, 33));
pst = "A"; //如果要显示其他的字母，则改变 pst 的值;
show.setText(pst);
```

字符的下落主要依靠移动 `JLabel` 的坐标值实现，具体的代码：

```
while(true){
    if(y >= 350){ //落到一定高度，逃出边界则退出;
        show.setVisible(false);
        break;
    }
    show.setBounds(new Rectangle(x, y, 33, 33));
    try{
        Thread.sleep(400); //每隔 400ms 的时间;
    }catch(InterruptedException e){
        e.printStackTrace();
    }
    y += 4; //JLabel 每次移动的距离为 4 个单位;
}
}
```

请根据上述的已知信息，使用线程技术，修改附录中的 `typeGame.java` 代码，使得十个不同的字母同时从随机的初始位置下落。

练习:

问题 1: 当前 Buffer 容器的大小只有 1, 如果某个程序中容器的大小可变由用户指定, 那么又该如何实现呢? 提示: 可以使用 java 自带 `java.util.concurrent.BlockingQueue` interface。请在使用 `BlockingQueue` 接口时考虑一个问题: 消费者是在 `blockingqueue` 满的时候一次性消费数据, 还是只要有数据填入即使 `blockingqueue` 不满也照样消费数据?

<http://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/answers.html> 实例 1: 生成者和消费者模型;

<http://blog.csdn.net/liuxingtianshi9570/article/details/39135043>

<http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>

问题 2: 阅读 Java 版的贪吃蛇源代码, 理解为什么需要使用多线程?

<https://blog.csdn.net/u010412719/article/details/51804360>

参考文献:

三个线程轮流输出 ABC:

<http://blog.csdn.net/u012110719/article/details/47161789>

Java 中自带的线程池技术:

<http://www.cnblogs.com/dolphin0520/p/3932921.html>

<http://www.importnew.com/19011.html?replyto=560513#respond>

7. Java 与面向对象程序设计基础

面向对象设计的一个基本原则是高内聚、低耦合。高内聚要求：类尽量对自己负责，完成自己要负责的功能，并且保持对外界透明。而低耦合更强调类与类之间的关系，比如模块之间尽量减少依赖，仅透露需要对方知道的信息即可。很多读者听到的编程原则，比如依赖导致原则、面向接口编程、面向抽象编程等，目的都是做到类之间的低耦合。

现实中，也有类似的低耦合的例子。比如我们笔记本中的处理器、内存、主板和硬盘，一般是不同厂商独立生产的，没有一个内存厂商会对客户说：告诉我，你要使用的主板，我再给你量身定做内存。内存厂商都是独立生产内存条。各个器件厂商不管别家的生产，而专注于自己的工作，就是低耦合原则的体现。

我们再看一个编程的例子。房子中的有一个中央控制器，要去管理每个电器的开关。这个场景如何做到低耦合呢？首先，中央控制器不需要去知道每个电器具体是什么，因为不管是电饭煲，还是热水器；中央控制器只需要知道它是一个带开关的电器，也即每个具体的电器需要被抽象为一个只带有开关功能的虚拟电器。另外，每个具体电器一定是自带开关功能的。从 Java 语言的角度而言，中央控制器只要去操作一个接口，而每个具体电器则实现了这个接口，结构图如下：

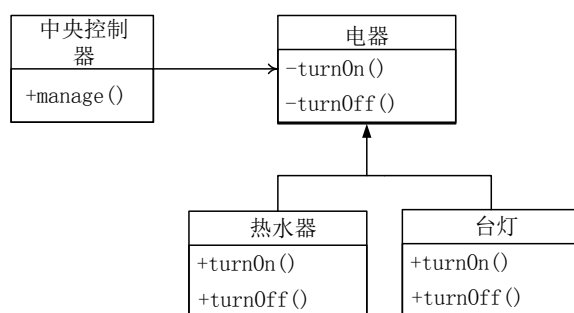


图 1

图中可见，中央控制器只需要知道电器这个接口的存在即可，至于有哪些具体的电器，中央控制器不需要知道。而热水器和台灯等只是实现开和关的功能，他们也不知道中央控制器的存在。两者之间只是通过电器这个接口想联系。类似地，笔记本电脑中各个器件的生产厂商也仅通过产品标准相联系，就可相互独立生产，而不用担心最后的组装问题。

基于 java 语言，我们来实现图 1 中的设计。中央控制器控制电器的方法是 `manage` 方法，其主要的功能是根据当前电器的状态，如果处于开状态，就关闭电器；处于关的话，那就打开电器。因此，图 1 中的接口还需要增加查询当前电器状态的方法：

```
public interface Appliance{
    public void turnOn();
    public void turnOff();
    public boolean isOn();
}
```

Central.java:

```
public class Central{
    private Appliance app = null;
    public void manage(){
        if(app.isOn()){
            app.turnOff();
        }
    }
}
```

```

        }else{
            app.turnOn();
        }
    }
    public void setApp(Appliance a){
        app = a;
    }
}
Test.java:
public class Test{

```

```

    public static void main(String[] args){
        Central ct = new Central();
        Lamp lp = new Lamp();
        ct.setApp(lp);
        ct.manage();

        Waterheater wh = new Waterheater();
        ct.setApp(wh);
        ct.manage();
    }
}

```

注意看，`Central.java` 中只出现 `Appliance` 接口，而没有出现 `Lamp` 和 `Waterheater` 对象，也就是说通过 `Appliance` 接口，`Lamp` 和 `Waterheater` 对 `Central` 不可见。初学的读者可能会问，在 `Test.java` 中，`Lamp` 的实例 `lp` 不是通过 `setApp` 方法让 `ct` 知道其存在了么？这里要注意，“可见不可见”是对 `Central.java` 对象里的代码而言，并不是针对外在的测试代码。`Central` 类内部的代码，的确不知道 `Lamp` 对象的存在。后续如果再有电视对象要控制开关，同样地，`Central` 类内部的代码完全不需要修改，就可直接使用。

假设，我们不采用上述的设计，而是让 `Central` 知道 `Lamp` 和 `Waterheater` 的存在，那对应的代码就成为如下：

```

public class Central{
    private Lamp lp = null;
    private Waterheater wh = null;

    public void setLamp(){...}
    public void setWaterheater{...}

}

```

这样的写法，`Lamp` 对象和 `Waterheater` 对象都对 `Central` 可见，不仅造成的冗余（比如 `set` 要写多个版本），而且如果有新的电器要控制，`Central.java` 中的代码又要增加新的代码，实现对新增电器的可见与控制。之所以，后续的扩展会这么繁琐，源头就在于 `Central` 被改为对具体的电器可见，吃力又不讨好。而像原来只对虚拟的电器 `Appliance` 可见就不存在类似的问题。

基于接口版本的 `Central.java` 可自由扩展而不用修改代码，即对新电器的控制不需要修

改 `Central.java` 中的任何代码，这种特性十分重要，是程序开发人员梦寐以求的状态。所以，这种特性值得专有名词描述“开放封闭原则”，对扩展（功能）开放，对修改（代码）封闭。按开放封闭原则来解释中央控制器，那基于接口的实现就是对新电器的控制是开放的，你要控制新的电器，比如 `Television`，那只要增加实现 `Appliance` 接口的 `Television` 类即可（对功能开放），而 `Central` 类中的代码则不需要修改，即对修改封闭。

下面再看个说明开放封闭原则的案例。一个画图程序中不同的图形，要依次绘制出来。每个图形是一个类，绘制方法中有一集合，凡是集合中的图形依次被绘制。程序后续会有新的图形加入，因此绘制程序要对新图形绘制功能扩展开放，同样不需要修改代码。绘制程序对每个具体的图形进行抽象，即每个图形只是能绘制自己的 `Shape`，因此得到接口：

```
public interface Shape{
    public void draw();
}
```

剩下的具体图形 `Circle` 和 `Square` 只要实现该接口即可。

`Circle.java`:

```
public class Circle implements Shape{
    public void draw(){
        System.out.println("draw a circle");
    }
}
```

再看看绘制方法的代码：

```
public class TestDraw{
    static void drawAllShapes(Shape[] shapeList, int n){
        int i;
        for(i=0; i<n; i++){
            Shape sp = shapeList[i];
            sp.draw();
        }
    }

    public static void main(String args[])
    {
        Circle c = new Circle();
        Square s = new Square();

        Shape[] shapelist = {c, s};
        drawAllShapes(shapelist, 2);
    }
}
```

只有抽象的 `Shape` 接口对 `drawAllShapes` 方法可见，有多少图形以及具体是哪些，`drawAllShapes` 方法不需要知道。图形绘制时，`drawAllShapes` 方法主要调用 `sp.draw()` 方法即可。因此，后续如果有三角形 `Triangle` 需要绘制，只要实现 `Triangle` 类，并且修改下测试代码即可：

```
public static void main(String args[])
{
```

```

        Circle c = new Circle();
        Square s = new Square();
        Triangle t = new Triangle();

        Shape[] shapelist = {c, s, t};
        drawAllShapes(shapelist, 3);
    }

```

`drawAllShapes` 方法中的代码不需要改动，做到了开放封闭原则。

要注意的是开放封闭原则是针对特定需求而言的。如果考虑不同的需求，同样的代码，可能就做不到开放封闭。比如 `drawAllShape` 方法对新图形绘制的需求，可以做到开放封闭原则；但如果有了新的需求，比如要求图形绘制保持某种顺序，那么原来 `drawAllShape` 中的代码就不能做到开放封闭了。以两个图形 `Circle` 和 `Square` 为例，假设要求 `drawAllShape` 方法不管参数的输入顺序，`Circle` 图形要先于 `Square` 绘制(注意也可要求 `Square` 图形要先于 `Circle` 绘制)。对于这种两个图形有先后顺序的绘制需求，`drawAllShape` 应该如何修改呢？

可以肯定的是 `drawAllShape` 方法中不应该出现 `Circle`、`Square` 这种具体的类名，因为不是每一次都会给定这两个类，也许下次是 `Circle` 和 `Triangle` 类，你是没有办法在 `drawAllShape` 中穷尽所有情况的。所以，`drawAllShape` 方法中要有对输入类进行排序的操作，但排序操作又不应该涉及到具体的类。先后的顺序最好由类自己来负责。就 `Circle` 和 `Triangle` 两个类而言，`Circle` 类遇到 `Triangle` 类，就会抢先；同样地，`Triangle` 类遇到 `Circle` 类就该退让。抢先还是退让，由具体的图形类本身负责，不应该放到 `drawAllShape` 中来处理。按照上述的思路，每个类除了实现 `draw` 接口，应该再实现一个是否抢先的接口。`drawAllShape` 方法绘制之前，首先对图形排序。

```

static void sortShape(Shape[] shapeList, int n){
    for(int i=0; i<n; i++)
        for(int j=0; j<i; j++)
        {
            if(shapeList[j].Precedes(shapeList[j+1]))
            {
                Shape temp = shapeList[j];
                shapeList[j] = shapeList[j+1];
                shapeList[j+1] = temp;
            }
        }
}

```

排序过程中，关键是调用类的 `Precedes` 方法比较 `Shape[j]` 和 `Shape[j+1]` 的先后顺序，从而确定前后两个图形的先后顺序。

```

class Circle implements Shape{
    public void draw(){
        System.out.println("draw a circle");
    }

    public boolean Precedes(Shape s){
        if(s.getClass().equals(Square.class))
            return true;
    }
}

```

```
        else
            return false;
    }
}
```

Circle 类中,如果遇到下一个 Shape 是 Square 类型,则返回 false,说明不需要调换顺序。而对 Square 类而言,如果遇到下一个是 Circle 类型,则返回 true,说明要调换顺序。

Circle.java:

```
class Circle implements Shape{
    public void draw(){
        System.out.println("draw a circle");
    }

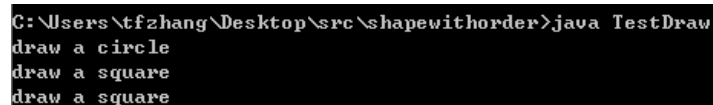
    public boolean Precedes(Shape s){
        if(s.getClass().equals(Square.class))
            return false;
        else
            return true;
    }
}
```

测试代码如下:

```
public static void main(String args[])
{
    Circle c = new Circle();
    Square s = new Square();
    Square t = new Square();

    Shape[] shapelist = {s, t, c};
    drawAllShapes(shapelist, 3);
}
```

给定的输入是 square, square, circle, 而输出的内容中保证 circle 的先输出。



```
C:\Users\tfzhang\Desktop\src\shapewithorder>java TestDraw
draw a circle
draw a square
draw a square
```

上述实现主要的不足在于:Circle 类需要知道 Square 类的存在。相互之间的耦合太严重,另外,当需求扩展为多余两个的图形的任意顺序输出,那么上述的代码就不能满足开放封闭原则。如何再修改 drawAllShape 方法,使得代码对任意多个图形的任意顺序绘制这一需求开放封闭呢?

有没有办法让图形的 Precedes 函数做到封闭呢?即增加新的图形,新的排序,Shape 的 Precedes 函数可以不用修改?简单地来说,就是派生类之间不用做到相互知晓,消除耦合。此处我们采用一种数据表驱动的方法:把各个图形类的名称和排名先后集中到一张表格中,根据这张表格来进行排序操作。将所有涉及到的类名放置在 OrderTable 中:

```
public class OrderTable{
    final static Object[] orderTable = {
        Square.class,
```

```

        Circle.class,
        Triangle.class,
        0
    };
}

```

测试代码中：

```

public static void main(String args[])
{
    Circle c = new Circle();
    Square s = new Square();
    Triangle t = new Triangle();

    Shape[] shapelist = {s, t, c};
    drawAllShapes(shapelist, 3);
}

```

最终的输出：

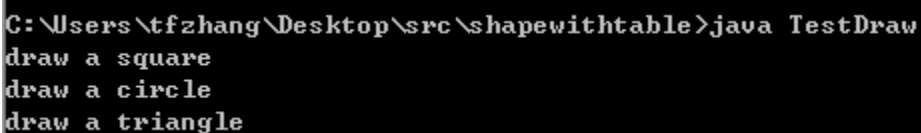
```

public static void main(String args[])
{
    Circle c = new Circle();
    Square s = new Square();
    Triangle t = new Triangle();

    Shape[] shapelist = {s, t, c};
    drawAllShapes(shapelist, 3);
}

```

最终的输出根据 OrderTable 中的顺序。



```

C:\Users\tfzhang\Desktop\src\shapewithtable>java TestDraw
draw a square
draw a circle
draw a triangle

```

表驱动编码后，每个具体的类功能又恢复到最基本的绘制方法：

```

class Circle extends Shape{
    public void draw(){
        System.out.println("draw a circle");
    }
}

```

关于顺序判定的代码都集中到虚类 Shape 中（因为要包含 Precedes 方法的实现，所以从 interface 修改为 abstract 类）：

Shape.java:

```

public abstract class Shape{
    public abstract void draw();

    public boolean Precedes(Shape s){
        int thisOrder = -1;
    }
}

```

```

        int argOrder = -1;
        boolean done = false;

        int i=0;
        while(!done){
            Object item = OrderTable.orderTable[i++];
            if(item != null){
                if(this.getClass().equals(item))
                    thisOrder = i;
                if(s.getClass().equals(item))
                    argOrder = i;
                if(thisOrder >= 0 && argOrder >= 0)
                    done = true;
            }else
                done = true;
        }
        return thisOrder > argOrder;
    }
}

```

Precedes 代码中，注意 **thisOrder** 是当前类在数据表中的序号，而 **argOrder** 则是下一个类（当前类在 **ShapeList** 中的下一个）在数据表中的序号，根据大小判定两者之间是否需要调换位置，从而完成排序。

采用数据表的方法，后续要有新的图形加入，只需要修改两处：1、书写继承自虚拟类 **Shape** 的新图形类；2、修改 **orderTable** 数据表。而 **drawAllShapes** 中的代码完全不需要修改，做到了开放封闭原则。

通过上述的两个例子，读者可以体会到开放封闭原则的巨大威力，遵循开放封闭原则可以使程序更加灵活，更有可扩展性和重用性。做到开放封闭原则的关键是对程序中可能会出现频繁变化的那些部分进行抽象。当然，能不用抽象，尽量也不要滥用。

结合具体的应用场景和面向对象程序设计的基本原则，人们开发出很多应用广泛的设计模式。设计模式可以理解为程序开发中的定式，前人将特定场景下的最佳处理方式整理出来，这就有了设计模式。当然，遇到新的场景，你能结合面向对象程序设计的基本原则，也能开发出新的模式。设计模式可以看做招式，而像封闭开放这样的设计原则就是内功。设计模式有很多，我们只介绍最常用的 **template** 和 **strategy** 模式，以及后续会使用的观察者模式。下面我们通过冒泡排序这个方法来讲解 Java 程序设计模式中的 **template** 模式和 **policy** 模式。首先，我们来看一个已经写好的 **int** 类型的冒泡排序程序。

```

public class IntBubble{
    private int[] intArray;
    private int arrayLen;

    public void sortArray(int[] a)
    {
        intArray = a;
        arrayLen = a.length;
    }
}

```

```

        for(int i=arrayLen-2; i>0; i--)
            for(int j=0; j<i; j++)
            {
                if(compare(j, j+1))
                    swap(j, j+1);
            }
    }

    public boolean compare(int i, int j)
    {
        if(intArray[i] > intArray[j])
            return true;
        else
            return false;
    }

    public void swap(int i, int j)
    {
        int temp = intArray[i];
        intArray[i] = intArray[j];
        intArray[j] = temp;
    }

    public static void main(String args[])
    {
        int a[] = {3, 4, 5, 1};
        IntBubble ib = new IntBubble();
        ib.sortArray(a, a.length);
        for(int i=0; i<a.length; i++)
            System.out.print(a[i]+" ");
    }
}

```

由于冒泡排序十分常用，现在我们要对 `double` 数据类型或者对 `String` 类型写个冒泡排序程序，那是否得写个 `DoubleBubble.java` 或者 `StringBubble.java` 文件呢？你可能不愿意了，因为你觉得只是换了个数据类型，代码的大致逻辑相同，有必要全部重写么？所以，要解决的问题是如何重用已有的 `IntBubble` 代码，完成对 `Double` 或者 `String` 数据类型的冒泡排序。解决方案 1，引入 `template` 模式，本人更喜欢称之“骨架模式”，即将 `Int`，`Double`，`String` 三种数据类型冒泡排序的共有逻辑抽取出来成为一父类，对应三个版本的冒泡排序就是三个子类。

冒泡排序的共有逻辑就是 `sortArray` 方法，其中 `sortArray` 只有前两句的方法与具体数据类型 `int a[]` 有关，其他部分与具体数据类型无关，所以可以被抽取。

```

public void doSort( )
{
    for(int i=arrayLen-2; i>0; i--)

```

```

        for(int j=0; j<i; j++)
        {
            if(compare(j, j+1))
                swap(j, j+1);
        }
    }

```

swap 方法和 compare 方法，与具体的类型有关，也应该被抽象，最后抽象得到的 BubbleSort.java:

```

public abstract class BubbleSort{
    protected int arrayLen = 0;
    abstract boolean compare(int i, int j);
    abstract void swap(int i, int j);

    protected void doSort( ){
        for(int i=arrayLen-1; i>0; i--){
            for(int j=0; j<i; j++){
                if(compare(j, j+1))
                    swap(j, j+1);
            }
        }
    }
}

```

三种数据类型冒泡排序只要继承 BubbleSort，并且实现 compare 方法和 swap 方法即可。如下是 Int 类型的 IntBubble.java:

```

public class IntBubble extends BubbleSort{
    private int[] array = null;
    public void swap(int i, int j){
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    public boolean compare(int i, int j){
        if(array[i] > array[j])
            return true;
        else
            return false;
    }
    public void bubbleSort(int [] a){
        array = a;
        arrayLen = a.length;
        doSort();
    }
}

```

请读者自行完成 Double 类型和 String 类型的冒泡排序版本。

与 `template` 模式不同，`Strategy` 模式不是将共有的算法放进一个抽象基类中，而是将其放进一个具体类 `BubbleSorter`。对于个性的部分，比如 `swap` 和 `compare` 方法，先抽取出来集中为接 `SortHandle`，再由不同数据类型的类去实现该接口，而 `BubbleSorter` 只要去操作 `SortHandle`，不需要知道具体是什么数据类型。我们来看看具体的实现：

`BubbleSorter.java`:

```
public class BubbleSorter{
    private int arrayLen = 0;
    private SortHandle bh = null;

    public void setHandle(SortHandle bh){
        this.bh = bh;
    }
    public void doSort(Object a){
        bh.setArray(a);
        arrayLen = bh.getLength();

        for(int i=arrayLen-1; i>0; i--){
            for(int j=0; j<i; j++){
                {
                    if(bh.compare(j, j+1))
                        bh.swap(j, j+1);
                }
            }
        }
    }
}
```

接口：

```
public interface SortHandle{
    public void setArray(Object a);
    public int getLength();
    public boolean compare(int i, int j);
    public void swap(int i, int j);
}
```

`Int` 数据类：

```
public class IntHandle implements SortHandle{
    private int[] array = null;
    public void setArray(Object a){
        array = (int[])a;
    }
    public int getLength(){
        return array.length;
    }
    public boolean compare(int i, int j){
        if(array[i]>array[j])
            return true;
        else
```

```

        return false;
    }
    public void swap(int i, int j){
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

```

测试文件：

```

public class TestBubble{
    public static void main(String[] args){
        int a[] = {1, 3, 2, 4};
        IntHandle ih = new IntHandle();

        BubbleSorter bs = new BubbleSorter();
        bs.setHandle(ih);
        bs.doSort(a);
        for(int i=0; i<a.length; i++)
            System.out.print(a[i]+" ");
    }
}

```

代码中值得注意的两点：

1. `public void setArray(Object a)`，使用 `Object` 的原因在于具体的数据类型是未知的，但是所有的类都继承自 `Object`，所以 `Object` 可以覆盖所有的数据类型；
2. 测试代码中，`IntHandle` 通过 `setHandle` 方法才与 `BubbleSorter` 产生联系，而实际上 `IntHandle` 类并不知道 `BubbleSorter` 的存在，而 `BubbleSorter` 也不需要知道当前的 `SortHandle` 具体是什么类型，最小化相互之间的信息暴露，降低类之间的耦合，提升了设计的灵活性，增加 `IntHandle` 具体类的重用性。为什么 `Strategy` 模式能增加 `IntHandle` 具体的重用性呢？我们以选择排序 `SelectionSorter` 为例说明：

```

public class SelectionSorter{
    private int arrayLen = 0;
    private SortHandle bh = null;

    public void setHandle(SortHandle bh){
        this.bh = bh;
    }
    public void doSort(Object a){
        bh.setArray(a);
        arrayLen = bh.getLength();

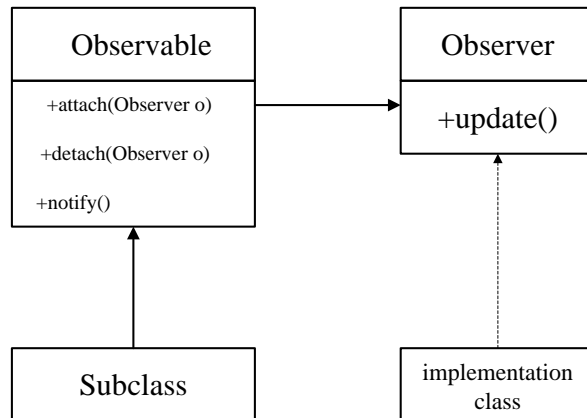
        for(int i=0; i<arrayLen-1; i++){
            int k=i;
            for(int j=k+1; j<arrayLen; j++){
                if(!bh.compare(j, k)){

```

测试代码:

对比测试代码，读者可以发现 `IntHandle` 完全不需要修改，可以原封不动供 `SelectionSorter` 使用，其中的原因：1. 选择排序使用的基本操作（比如 `Compare, Swap`）与冒泡排序没有差异，故 `IntHandle` 可以重用；2. `BubbleSorter` 与 `IntHandle` 相互之间松耦合。相对地，如果是继承的 `template` 模式，那选择排序的 `Int` 版本就需要大费周章了，首先要书写选择排序的抽象基类，然后再去继承实现对应的 `Int` 数据版本，代码基本重写，可重用度低。`Strategy` 模式虽然具有更强的灵活性、可维护性，但是以相应的设计复杂性，内存以及运行时的时间开销为代价；是否使用，视具体情况而定。

观察者定义对象间的一对多的依赖关系，当一个对象的状态发生变化时，所有观察它的对象将得到通知并执行对应的操作。一般而言，存在一个被观察者和多个观察者，且观察者的数量可以增加或者减少。



Observable 是被观察对象，Observer 是观察者，observable 通过 attach 和 detach 方法增加或者删减被观察者对象。其中，Observable 设置为抽象类，而 Observer 设置为接口，其中只有 update 方法。如何使用呢？具体的被观察者类继承 Observable 类，并且通过 attach 方法将实现了 Observer 接口的观察者，与被观察者相关联。

Observable 类：

```
import java.util.Vector;
```

```
abstract class Observable{
    private Vector<Observer> obs = new Vector<Observer>();
    public void addObserver(Observer ob){
        obs.add(ob);
    }
    public void delObserver(Observer ob){
        obs.remove(ob);
    }
    protected void notifyObserver(){
        for(Observer o: obs){
            o.update();
        }
    }
    public abstract void doSomething();
}
```

Observer.java:

```
public interface Observer{
    public void update();
}
```

Observable 类中的 doSomething()方法主要是发生事件后，调用 notifyObservers 方法，通知观察者。上述的一个抽象类和接口，就实现观察者模式。具体如何使用呢？下面有一个被观察者，两个观察者。

被观察者代码：

```
public class MyObservable extends Observable{
    public void doSomething(){
```

```
        System.out.println("something you care occur");
        this.notifyObserver();
    }
}
```

两个观察者的代码:

```
public class MyObserver1 implements Observer{
    public void update(){
        System.out.println("Observer1 gets your message");
    }
}

public class MyObserver2 implements Observer{
    public void update(){
        System.out.println("Observer2 gets your message");
    }
}
```

测试代码:

```
public class TestObserver{
    public static void main(String[] args){
        MyObservable obva = new MyObservable();
        obva.addObserver(new MyObserver1()); //添加观察者 1
        obva.addObserver(new MyObserver2()); //添加观察者 2
        obva.doSomething();
    }
}
```

请读者再尝试 `delObserver` 方法的测试。上述我们自己动手构建的观察者模式与 Java 自带的原理一致，细节和使用稍有不同，相信读者只要看下示例就能使用。Java 自带的被观察者抽象基类是 `java.util.Observable`，观察者接口是 `java.util.Observer`。后续的贪吃蛇游戏中会使用观察者模式。

关于设计模式，很多经典的书籍都有介绍，那我们该如何看待设计模式呢？一般的观点：设计模式是好药，但要对症下药；如果没病，就尽量不要吃药。

8. 测试驱动(TestDriven)的开发方法

我们知道软件中出现 bug 不可避免，不可能百分百杜绝其的出现，能做的只能是优化开发流程，尽量少地在开发过程中引入 bug；同时，尽量早地发现和消除 bug，因为随着开发的过程的进行，初期的 bug 造成的破坏越大，而发现初期的 bug 的成本越来越高。TestDriven 开发，即测试驱动的开发技术，是一种减少软件 bug，增强软件质量的有效技术。顾名思义，TD 技术要求软件代码书写之前或者过程中，书写测试代码并且进行测试。我们试着来开发一个小程序，来经历和体验 TD 开发技术。

首先，安装配置第三方提供的测试框架，我们一般使用的免费的 Junit 框架（具体的安装配置过程请参见文档 Junit4 的安装和配置。）通过一简单的例子，我们介绍下 Junit 的使用。

图 1 展示一简单的加减乘除运算器 Cal.java:

```
public class Cal{
    public int add(int a, int b){
        return a+b;
    }
    public int sub(int a, int b){
        return a-b;
    }
    public int mul(int a, int b){
        return a*b;
    }
    public int div(int a, int b){
        return a/b;
    }
}
```

图 1

程序本身很简单，虽然正确性一目了然，但作为例子，假设我们对程序的正确性不甚明了。那就要对程序正确与否进行验证。验证的基本步骤：给定输入值，然后对照其输出值与预设正确值是否有出入，如果两者一致，表示程序给出正确值，我们有理由相信代码没有错误；反之，可以肯定代码有误。所以，验证程序对错的验证代码要完成两个任务：1、比较程序值与预设正确值；2、如果出错，则发出报警。以 Cal 中的 sub 方法为例，如果要验证该方法中的代码是否正确，我们可以亲手 diy 打造如下的验证代码：

```
public class TestSub{
    public static void main(String[] args){
        Cal c = new Cal();
        if( c.sub(24, 1) != 23){
            System.out.println("there is something wrong in sub method of class Cal");
        }
    }
}
```

24 减去 1 应该等于 23，所以 if 语句对应验证代码的比较过程；而后面的 println 语句则在比较出错的时候，发出警告告诉开发人员，具体代码的哪个位置发生错误。上述的 diy 代码虽然很好地完成验证的两个步骤，但也有明显不足。首先，代码的冗余十分明显，一旦某个类

有成千上百的方法时，那要写的 `println` 语句数量可想而知；不能精准定位错误且工作量大，`println` 中错误发生在何处，需要程序员来负责定位，加大程序员工作量；一旦程序规模变大，重名导致错误定位也是可能发生的。测试框架，比如 Junit，正是为解决上述的弊端而应时出现。以具体为例，Junit 是如何来验证类 `Cal` 中的 `sub` 方法的呢？

```
import junit.framework.*;

public class TestSub2 extends TestCase{
    public static void main(String[] args){
        Cal c = new Cal();
        assertEquals(23, c.sub(24, 1));
    }
}
```

代码中两个地方值得留意。首先是 `import` 和 `extends` 语句，要使用 Junit 框架中的功能，这是免不了的步骤。另外，可以发现 `if` 和 `println` 语句都看不到了，取而代之的是简单的一句 `assertEquals`。根据前后的对比，`assertEquals` 的作用显而易见：当预设值与方法返回值相同时，验证通过；如果两者不符，程序会自动发出警报，读者可通过将 `23` 改为 `22` 确认。`assertEquals` 只用一条语句即完成我们 `diy` 的验证代码的工作，并且不需要开发人员来指定发生错误的位置，这就规避两个主要的弊端，给我们开发人员带来极大便利。`assertEquals` 还只是 Junit 框架中的一个简单常用的功能，Junit 还有很多其他强大的功能，其流行也就不奇怪了。下面是采用 Junit 验证类 `Cal` 完整的代码：

```
public class TestCal extends TestCase{
    private Cal t;

    public TestCal(){
        t = new Cal();
    }
    public void testAdd(){
        assertEquals(6, t.add(2, 4));
    }
    public void testSub(){
        assertEquals(23, t.sub(24, 1));
    }
    public void testMul(){
        assertEquals(24, t.mul(8, 3));
    }
    public void testDiv(){
        assertEquals(3, t.div(24, 8));
    }
    public static void main(String[] args){
        TestCal tc = new TestCal();
        tc.testAdd();
        tc.testSub();
        tc.testMul();
        tc.testDiv();
    }
}
```

}

对 Cal 类的测试，只用来说明 Junit 存在的价值，以及基本的使用流程。该案例不能说明测试这一工作的难度以及必要性。测试代码的编写常常要求完备性和正交性。完备性，即测试代码要涵盖所有可能出现的情况；正交性，则要求测试用例间相互独立，彼此不涵盖。测试完备性要求程序员对业务逻辑清晰的认识，才能够穷尽所有的用例情况，正确完备地进行功能测试。而正交性，就看测试过程中对案例的选择取舍。下面我们以一简单的保龄球计分程序为例，再次展现测试驱动的开发过程。整个开发记录主要分成以下几个部分，问题与业务逻辑的梳理，其次测试的简单讨论，之后是代码撰写和测试过程交替进行的开发过程记录。

问题与业务逻辑：

1	4	4	5	6	5		0	1	7	6			2	6
5		14		29	49	60	61		77	97	117		133	

图 2 保龄球计分卡

保龄球比赛按局进行，每局 10 轮，前 9 轮每轮有两次投掷机会，最后一轮有 3 次投掷机会。每轮的第一次投掷，击倒全部 10 个瓶子...
具体的比赛和计分规则请自行百度理解。

测试的简单讨论：

先明确将开发程序的输入与输出。保龄球计分程序的输入是每局比赛中每一次投掷的分数，对应图 1 的话，输入就是：1,4,4,5,6,4,5,5,10,0,1,7,3,6,4,10,2,8,6。
对应的输出是什么呢？考虑输出，只要从开发者视角切换到用户视角，即考虑用户在使用我们开发的小程序时，希望小程序能够告诉用户哪些信息，这些信息就是输出。对照图 1，用户希望的输出包括两个方面：截止当前轮的计分以及每局的最终计分。
测试过程的具体实现基本就是输入部分数据，然后对比程序的输出是否与期望的计分值一致，如果一致，认为测试通过；否则，测试不通过，程序有问题，我们根据提示信息修正。

开发过程实录：

我们先作一些约定：每局游戏 game，每一轮投掷 frame，每局 game 包含 10 个 frame。每一次投掷称为 ball；如果第一次投掷就将全部 10 个瓶击倒，称为 strike；如果两次投掷才把所有瓶子击倒，则是 spare。

好，现在开始写代码，从哪里入手呢？我们把 frame 作为一个独立的对象开始。图 1 是代码的大致轮廓，frame 对象的职责：add 方法记录本轮投掷的分数，getscore 方法返回截止目前轮的得分。

```
public class Frame{
    public void add(int score){
    }
    public int getScore(){
    }
}
```

图 2 Frame 代码

问题出现：因为 `getscore` 方法返回的并不是本轮投掷的总分，而要返回截止目前轮的得分。导致当前 `frame` 对象需要使用其他 `frame` 对象的数据，`game` 对象包含总统的 10 个 `frame` 对象，因此当前 `frame` 对象要通过 `game` 对象去访问其他 `frame` 对象的数据。`game` 对象依赖 `frame` 对象，而 `frame` 对象又反过来依赖 `game` 对象，这种双向依赖的关系是我们面向对象开发过程中要避免的。所以从 `frame` 对象切入的做法有误，我们返回，回到上一层，从 `game` 对象入手。

<pre> public class Game{ private int[] scores = new int[21]; public void add(int score){ } public void setFrameScore(int frame){ } public int getAllScore(){ } } </pre>	<pre> import junit.framework.*; public class TestGame extends TestCase { private Game g; public void testCase1(){ g = new Game(); } public static void main(){ TestGame tg = new TestGame(); tg.testCase1(); } } </pre>
--	--

图 3 Game 框架

图 3 是从 `Game` 入手后的 `game` 代码框架以及相关的测试代码框架。`Game` 的主要职责包括：`add()` 添加某轮投掷的分数，`getFrameScore(int frame)` 返回截止到 `frame` 轮的计分值，`getAllScore()` 返回整轮的计分值。`Game` 中，我们没有罗列出 10 个 `frame` 对象，`frame` 状态的主要作用来存储分值数据，所以先用 `int` 数组替代。（后期我们再看采用 `frame` 对象是否比 `int` 数组要好。）

测试代码我们采用独立的对象 `TestGame` 来实现，其中的每个测试用例，由独立的方法来实现。注意 `TestGame` 需要两个例行操作：1. `import junit` 的 `package`；2. `extends TestCase`。具体化，第一个测试用例：第一轮扔两次，分别计分为 1 和 4，对应的测试代码：

```

public void tesOneThrows(){ //将 testCase1 方法名修改为 testOneThrow，更具可读性。
    g = new Game();
    g.add(1);
    g.add(4);
    assertEquals(5, g.getScoreFrame(1));
}

```

`assertEquals` 是 `TestCase` 提供的方法，如果 5 和 `g.getScoreFrame(1)` 两者的值不等，程序会自动报错，也即提醒我们代码有问题（因为代码如果没问题的话，5 个 `getScoreFrame(1)` 返回的值相等。）

为完成上述的测试，我们要修改 `Game` 代码，增加 `ball` 私有变量，作为 `scores` 数组的下标，指示第几个球的计分。同时，增加 `currentFrame` 私有变量，用来表示当前是第几轮。这两个变量的更新如下：

```

public void add(int score){
    scores[ball++] = score;
}

```

```

        if(score == 10) //当前分数为 10 分，就算作一轮；
            currentFrame++;
        else if(mark == false){ //mark 保证投掷两次才更新轮数；
            currentFrame++;
            mark = true;
        }else if(mark == true){
            mark = false;
        }
    }
}

```

getFrameScore 对应的代码如下：

```

public int getFrameScore(int frame){
    int ball;
    int frameIndex;
    int score = 0;

    for(frameindex =0, ball=0; frameindex<frame;)
    {
        if(scores[ball] == 10){
            frameIndex++;
            score += scores[ball++];
            score += scores[ball] + scores[ball+1];
        }else{
            if(scores[ball]+scores[ball+1] ==10)
                score += scores[ball] + scores[ball+1] + scores[ball+2];
            else
                score += scores[ball] + scores[ball+1];

            ball += 2;
            frameIndex++;
        }
    }
}

```

需要注意的是分数为 10 分的情况。getAllScore()函数返回比赛到目前为止的计分，可以划归到 getFrameScore 的情况，只要保证输入的参数为 currentFrame 即可。

```

public int getAllScore(){
    return getFrameScore(currentFrame);
}

```

运行测试程序后，测试通过。（标注：截止目前的源代码放置在阶段 1 文件夹中。）我们再测试下如下的代码：

```

public void testTwoThrows(){
    g = new Game();
    g.add(1);
    g.add(4);
}

```

```
        g.add(4);
        g.add(5);
        assertEquals(5, g.getFrameScore(1));
        assertEquals(14, g.getFrameScore(2));
        assertEquals(14, g.getAllScore());
    }
```

通过测试。

目前还没有考虑 strike 和 spare 的情况，下面分别考虑这两种情况：

```
public void testSimpleStrike(){
    g = new Game();
    g.add(10);
    g.add(3);
    g.add(5);
    assertEquals(18, g.getFrameScore(1));
    assertEquals(26, g.getFrameScore(2));
    assertEquals(26, g.getAllScore());
}
```

```
public void testSimpleSpare(){
    g = new Game();
    g.add(3);
    g.add(7);
    g.add(5);
    g.add(1);
    assertEquals(15, g.getFrameScore(1));
    assertEquals(21, g.getFrameScore(2));
}
```

上述两种情况都通过测试，我们再测试 strike 和 spare 结合的情况。

```
public void testStrikeSpare(){
    g = new Game();
    g.add(3);
    g.add(7);
    g.add(10);
    g.add(8);
    g.add(1);
    assertEquals(20, g.getFrameScore(1));
    assertEquals(39, g.getFrameScore(2));
}
```

测试通过。（标注：截止目前的源代码放置在阶段 2 文件夹中。）

以上的游戏只是整个游戏中的局部，下面我们测试两种简单的全局情况。第一种全局情况是每一次的投掷都是 strike。那么一共要投掷 12 次：前 9 次/轮+最后 3 次/轮。前 9 轮计分 270，最后 3 次投掷算在同一局，最高 30。所以 12 次投掷均为 strike 的话，总分 300。测

试代码如下：

```
public void allStrike(){
    g = new Game();
    int i;

    for(i=0; i<13; i++)
        g.add(10);

    assertEquals(270, g.getFrameScore(9));
    assertEquals(300, g.getAllScore());
}
```

运行测试，代码不通过，具体如下：

```
Exception in thread "main" junit.framework.AssertionFailedError: expected:<300>
but was:<360>
    at junit.framework.Assert.fail(Assert.java:47)
    at junit.framework.Assert.failNotEquals(Assert.java:277)
    at junit.framework.Assert.assertEquals(Assert.java:64)
    at junit.framework.Assert.assertEquals(Assert.java:195)
    at junit.framework.Assert.assertEquals(Assert.java:201)
    at TestGame.allStrike(TestGame.java:65)
    at TestGame.main(TestGame.java:75)
```

说明如下的测试语句没有通过，原因是输出应该是 300，而程序给出的是 360。

```
assertEquals(300, g.getAllScore());
```

该如何解释多出来的 60 呢？一个原因，最后一轮的第一次投掷（ball 10）因为投掷 10 分，所以又被误认为是新一轮，总分会再加上后续两次的 10 分，此处多加 30 分。而 ball 11 也被认为是新一轮，多加 20 分；最后 ball 12 又被认为是新一轮。因此，整个一局被记录为 12 轮，正常情况下应该是被记为 10 轮。要修改两处：一是更新 currentFrame 的代码；其次是 getFrameScore；

```
private void adjustFrame(){
    currentFrame = Math.min(currentFrame+1, 10);
}

public void add(int score){
    scores[ball++] = score;
    if(score == 10)
        adjustFrame();//currentFrame++;
    else if(mark == false){
        adjustFrame();//currentFrame++;
        mark = true;
    }else if(mark == true){
        mark = false;
    }
}
```

采用 adjustFrame()方法来限定 frame 的数量，保证不超过 10 次。

```
public int getFrameScore(int frame){
    int ball;
```

```

    int frameIndex;
    int score = 0;

    for(frameIndex =0, ball=0; frameIndex<frame;)
    {
        if(scores[ball] == 10){
            frameIndex++;
            score += scores[ball++];
            score += scores[ball] + scores[ball+1];
        }else{
            if(scores[ball]+scores[ball+1] ==10)
                score += scores[ball] + scores[ball+1] + scores[ball+2];
            else
                score += scores[ball] + scores[ball+1];

            ball += 2;
            frameIndex++;
        }
        if(frameIndex == 10 - 1)
            break;
    }
    if(frame == 10)
        score += scores[ball] + scores[ball+1] + scores[ball+2];
    return score;
}

```

因为第 10 轮投掷的情况与前 9 轮不同，所以使用下列两个代码片段进行单独处理：

```

    if(frameIndex == 10 - 1)
        break;
    if(frame == 10)
        score += scores[ball] + scores[ball+1] + scores[ball+2];

```

按照上述的修改，测试通过。我们再测试前 9 局的每次投掷都为 0，而第 10 局的 3 次投掷都是 10 的情况，代码如下：

```

public void butLastStrike(){
    g = new Game();
    int i;

    for(i=0; i<9; i++){
        g.add(0); g.add(0);
    }
    g.add(10);
    g.add(10);
    g.add(10);
    assertEquals(30, g.getFrameScore(10));
    assertEquals(0, g.getFrameScore(9));
}

```

}

测试通过。我们再用下面的全例测试：

1	4	4	5	6		5			0	1	7		6			2		6
5		14		29		49		60		61		77		97		117		133

```
public void fullExample(){
    g = new Game();

    g.add(1); g.add(4);
    g.add(4); g.add(5);
    g.add(6); g.add(4);
    g.add(5); g.add(5);
    g.add(10);
    g.add(0); g.add(1);
    g.add(7); g.add(3);
    g.add(6); g.add(4);
    g.add(10);
    g.add(2); g.add(8);g.add(6);
    assertEquals(49, g.getFrameScore(4));
    assertEquals(77, g.getFrameScore(7));
    assertEquals(133, g.getAllScore());
}
```

测试通过。我们再测试如下的情况：

```
public void tenthSpare(){
    g = new Game();
    for(int i=0; i<9; i++)
        g.add(10);

    g.add(9);
    g.add(1);
    g.add(1);

    assertEquals(270, g.getAllScore());
}
```

测试也通过。截止目前为止，我们尝试了所有能够想到的测试用例，程序都顺利通过测试，所以基本上可以认为开发完成。检视 `Game.java`，其中使用到的常数 10 最好使用更具可读性的常量符号代替：`public static final int NumOfFrames = 10;`

另外，我们发现 `Game` 类中并不需要包含 `Frame` 类，只要使用整数数组即可完成我们的任务，这有点超过当初的设想，但也不奇怪，因为好的设计往往不是一开始的拍脑袋设想，而是随着开发深入一步步演化出来的。另外，还可以发现测试驱动开发的好处：开发结束，我们自然而然有了整套的测试案例；对于后期的维护十分有利，任何一处代码的改动只要运行下测试案例，就能确定正确性，避免修改代码后引入新的问题。

练习:

问题 1: 采用测试驱动的方式, 开发一伙伴内存管理器。所谓的伙伴内存管理器, 用户提出申请时, 分配一块恰当的内存区域给用户; 反之, 在用户释放内存区域的时候收回。假设当前管理的内存总大小为 32 个整型数据, 当申请 5 个数据容量的存储空间时, 内存管理器先把 32 分割为 16+16 大小的两块, 再将其中第一个 16 分割为 8+8; 因为 $8 > 5$ 且 $4 < 5$, 所以将第 1 个 8 容量分配给 5 字节容量的请求。如果第 2 个存储空间申请大小为 3, 那么第 2 个 8 就会分割为 4+4; 然后将第 1 个 4 分配给容量为 3 的请求; 如果再来一个容量 3 的请求, 那么第 2 个 4 就被分配; 此时, 如果有一个 17 容量的请求, 由于剩下的最大内存块为 16, 所以不能满足请求; 当第 1 个 3 和第 2 个 3 的存储空间被释放时, 则两者又合并为一个 1 个容量为 8 的内存块。其他的操作依次类推。

32 个整型数据的下标为 0~31, 其中测试代码的大致形式如下:

```
assertEquals(0, test_alloc(4));
```

其中 `test_alloc` 方法输入的是申请空间的大小, 返回空间的第一个下标值; 如果没有足够的空间, 那么返回 -1, 所以有如下的语句:

```
assertEquals(0, test_alloc(4));
```

```
assertEquals(4, test_alloc(4));
```

```
assertEquals(8, test_alloc(4));
```

```
assertEquals(12, test_alloc(4));
```

```
assertEquals(16, test_alloc(16));
```

```
assertEquals(-1, test_alloc(1));
```

当空间被释放后, 如果相邻的兄弟块处于空闲状态, 那么导致融合, 可重新分配。

```
int m1 = test_alloc(4);
```

```
assertEquals(0, m1);
```

```
test_free(m1);
```

```
int m2 = test_alloc(8);
```

```
assertEquals(0, m2);
```

把整个空间比作树的根节点的话, 那么分配空间的过程相当于从根节点分出子节点的过程, 而释放空间则对应子节点合并删除复归到父节点的过程。还有, 从上到下的分支, 从下而上的合并, 都是递归的过程。比如对于 `test_alloc(4)`, 就是图 4 的分支过程, 没有标记的长条表示闲置空间, 如果此时再有 `test_alloc(7)`, 那么图中长条的 8 的兄弟整条会被分配。`test_free` 的过程与 `test_alloc` 相反可逆。

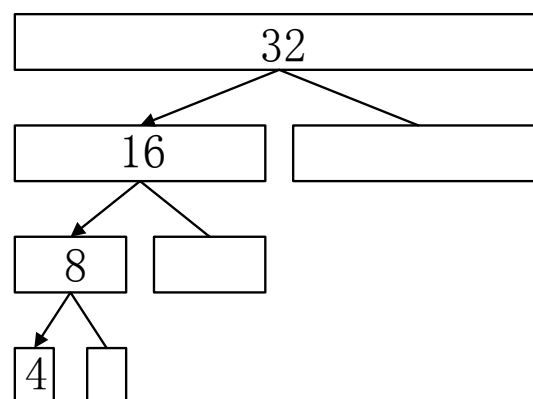


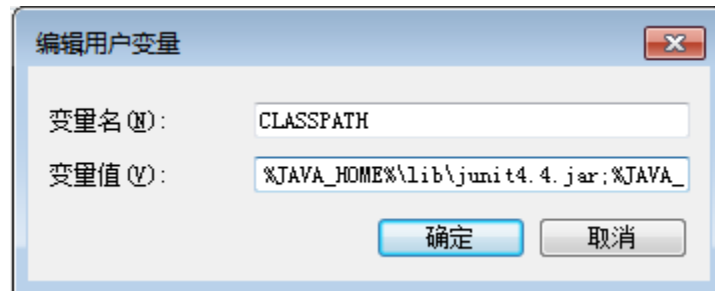
图 4 伙伴内存系统

附录：外来类包的安装

具体的开发过程中，需要安装来自第三方类包，一般是.jar 后缀。安装过程主要分为两步：

1. 下载对应版本的类包；
2. 设定路径，将第三方类包的路径添加到 classpath 中，需要注意：路径设定需包含.jar 后缀；

以 Junit4.4 第三方类包的安装为例，首先，下载 Junit4.4 类包（见附录），比如我们将其放置到 java 安装目录的 lib 文件夹中，并且将整个路径添加到 classpath 中即可。注意，路径要细化到 junit4.4.jar 文件名。



如果采用的是 Eclipse 开发软件，添加第三方开发包的过程是规范化操作，网上有很多介绍，请自行搜索。

参考文献：

- 1、《敏捷软件开发：原则、模式与实践》

9. 基于 MVC 框架的开发方法

什么是 MVC?

MVC 全称 Model View Controller，是模型界面控制的缩写。MVC 是什么呢？软件开发人员头脑中的开发框架，也可以说是一种看待软件的视角。以小程序贪吃蛇为例，由 MVC 的视角来看待如下：

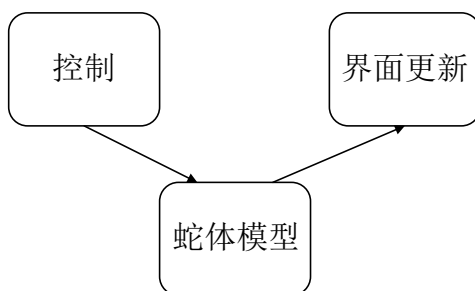


图 1 贪吃蛇的 MVC

其中的 control 部分，可以认为是玩家的按键输入，而 Model 则对应描述贪吃蛇体长度与位置的数据结构，而 view 则是玩家看到的蛇移动的界面。MVC 的视角一下子就将贪吃蛇游戏的构成分割清晰，让程序员易于动手开发。后文可见，贪吃蛇 MVC 的每个部分基本对应一个 java 文件。

MVC 视角看待扫雷：

实际的开发过程中，Model 涉及业务逻辑，往往是最关键的。以扫雷游戏为例，对应的 Model 就是一记录地雷位置信息及周边地雷数量的二维数组。模型的构建过程分为两个步骤：

- 1、二维数组中确定地雷的位置，即选定特定数量的二维数组元素为地雷，假设采用值-1 表示元素为地雷；

- 2、对于一般元素，统计其周围的地雷数量，也即统计周边非-1 的元素数量；

以 8*8 的地雷阵为例，即对应 8*8 的二维数组。第一步骤就是随机地生成 10 个地雷，即在数组 a[8][8]中随机地选择 10 个元素赋值为-1。主要的难点不要重复在一个位置放置地雷，即放置地雷时，不要在之前已经放置地雷的位置重复放置；而对剩余其他的元素，统计其周围地雷数，并且和赋值该元素。赋值的情况如下三种：

情况 1:

处于角落的元素，只要查看邻接的三个元素。以 a[0][0]为例，假设该角落没有随机分配到地雷，查看其邻居 a[0][1], a[1][0], a[1][1]三处的值，如果三处都没有分配到雷，那么 a[0][0]处赋值 0；如果三者之中有一个为-1，那么 a[0][0]处赋值为 1；两者是-1 的话，a[0][0]赋值为 2；三者都是-1，则是 3。

目标 节点	

情况 2:

目标 节点	

处于边界的元素（但不在角落），查看邻接的五个元素。邻接三个元素中-1 的个数之和就是目标节点的值。

情况 3:

除情况 1 和 2 外的内部节点，需要查看其周围 8 个节点中-1 的个数，将个数之和赋值给目标节点。

	目标 节点	

下图是按照上述两个步骤生成的地雷阵（-1 表示地雷）:

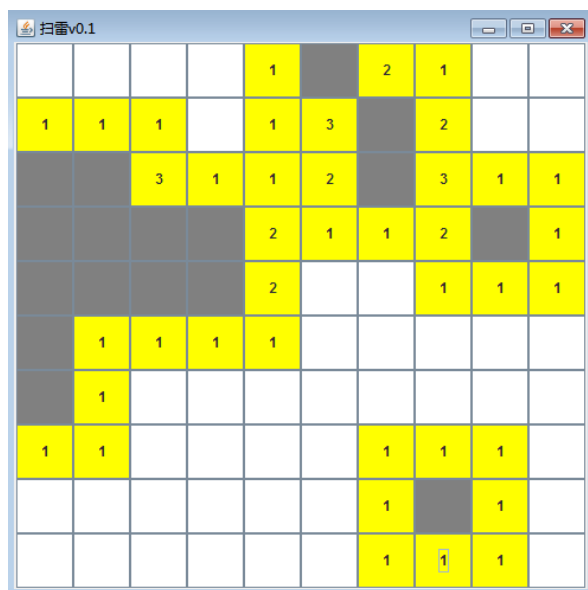
0	1	-1	2	2	-1	-1	2
0	1	1	2	-1	4	-1	2
1	1	0	1	1	2	1	1
-1	1	0	0	0	0	0	0
2	2	1	0	1	1	1	0
2	-1	2	0	2	-1	2	0
2	-1	2	0	2	-1	2	0
1	1	1	0	1	1	1	0

读者可以自行检验上述生成的地雷阵是否合规，请读者编写生成二维地雷阵的类 **Block**，对应的接口为：**Block(int x, int y, int numofmine)**; **x** 表示地雷阵的宽度，**y** 表示地雷阵的高度，而 **numofmine** 则是地雷的数量。

完成 **Model** 的设计与实现，再看界面的设计。玩过扫雷游戏就是用鼠标点击格子。点击的操作自然让人想到按钮，没错！地雷阵的每一个格子就可以使用按钮组件来实现，每个按钮有一个数值标记其属性，该数值正好与 **Model** 中的二维数组值一一对应，并且每个按钮初始化为同样的底色-灰色（**windows** 的扫雷游戏每个格子蓝色底色）。点击不同类型的按钮，带来不同界面变化。

- 1、点击地雷所在按钮，则终止游戏并且弹出对话框告知玩家；
- 2、如果点击到数值非 0 且非雷的按钮，则按钮褪去背景色，显示对应的数字，让玩家知道；
- 3、如果点击到值为 0 的按钮，按钮褪去背景色，变为白色，并且触发周边值同为 0 的按钮作同样的反应；

具体的效果如下图：



三中界面变化中，前两种较简单。较复杂的是第 3 种，主要原因是点击一个 0 值按钮，会触发近邻的其他的 0 值按钮，从而产生连锁反应，要怎么样实现这种连锁反应呢？直接的想法是采用递归的方式，当然也可以使用队列方式。下面是采用队列方式的算法步骤：

1. 点击到 0 值按钮时，当前按钮褪色为白色，并标记被点击；
2. 当前 0 值按钮周围的未被点击的按钮对象添加到队列；
3. 队列非空，则从队列弹出一个未被点击的 0 值按钮，重复步骤 1 和 2；
4. 重复步骤 3，直到队列为空；

MVC 的第三部分是 Control，扫雷的 control 相对简单，只要对捕捉玩家的点击即可。对每个按钮对象，增加相应的点击事件监听器，并且对不同类型的按钮增加对应的处理流程。

MVC 框架看待贪吃蛇：

下面再对贪吃蛇游戏应用 MVC 开发框架。同样地，先考虑贪吃蛇的 Model 部分，对应蛇体的描述。玩过贪吃蛇的都知道，贪吃蛇体长度是可变的，每次吃到新的豆子，蛇体长度会增加 1；并且蛇体中某个小块的位置都在时刻变化。因此，描述蛇体的数据结构要具备如下功能：

1. 数据结构包含对应于蛇体长度数量的节点，并且每个节点包含自身的位置信息；
2. 该数据结构可动态增长，蛇体每吃到新豆子，对应的数据结构就增加一个节点；

综合上述两点，蛇体应该是一种可动态增加的队列结构。每次新的数据节点从头部加入，同时，对应蛇体的每次移动，队列结构中的每个数据节点所包含的位置要周期性的更新。经过查找，LinkedList 数据结构符合要求。贪吃蛇身体的每个节点是个小方块，可采用方块左上方像素的坐标描述其所在位置：

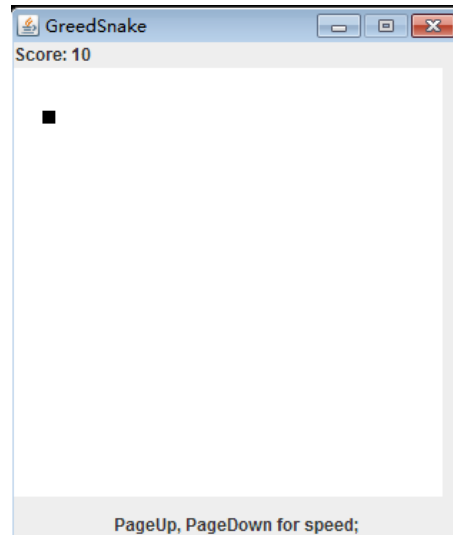
```
class node{
    int x ; int y;
}
```

LinkedList 主要负责对贪吃蛇如下行为的建模：

1. 贪吃蛇吃到新豆子，相当于往 LinkedList 头部加入新的 node 实例；如何判断贪吃蛇碰到豆子呢？只要将蛇头节点弹出，比较蛇头和豆子的 x、y 值是否相等，判断贪吃蛇是否吃到豆子；
2. 贪吃蛇是否咬到自己？取蛇头节点的 x 和 y 值，是否与蛇身体某个节点的 x 和 y 值相等；

3. 贪吃蛇是否撞墙？取蛇头的 x 和 y 值，判断其是否图形边界碰撞；
4. 贪吃蛇的移动相当于 `LinkedList` 中每个节点的 x 和 y 值得到更新，从尾部到头部，靠近尾部节点的 x 和 y 值采用相邻的靠近头部节点的 x 和 y 覆盖，依次类推，剩下的头部节点则根据贪吃蛇移动的方向更新值；

贪吃蛇的 `Model` 部分基本完成。考虑实现的切入点，与扫雷不同，不考虑从 `Model` 入手。因为 `Model` 的实现，很大层面依赖于界面，贪吃蛇需要从 `View` 入手。



`View` 采用最简单的实现，以白为底色，蛇体每一节和豆子均采用黑色的小方块实现，如图所示，就是如下代码实现：

```
Graphics g = paintCanvas.getGraphics();
g.setColor(Color.WHITE);
g.fillRect(0, 0, canvasWidth, canvasHeight);
g.setColor(Color.BLACK);
g.fillRect(2*nodeWidth, 3*nodeHeight, nodeWidth-1, nodeHeight-1);
```

蛇体的绘制：遍历 `LinkedList` 中的每个节点，获得每个节点的 x 和 y 值，然后由上述代码依次绘制每个节点即可完成蛇体绘制。蛇体移动的效果：`LinkedList` 中所有节点的 x 和 y 值更新后，清楚屏幕上原有的方块，按新的位置再绘制蛇体，呈现动态地效果；豆子的出现：发现豆子被吃后，随机产生豆子的 x 和 y 值（要保证该位置不落在现有的蛇体身上），并且完成绘制；

已知 `Model` 每隔相同时间段更新，而 `View` 根据 `Model` 的当前值进行重新绘制，实现时只要 `Model` 一更新完成，就通知 `View` 重新绘图即可。将两者抽象地比作为对象 `A(Model)` 和对象 `B(View)`，对象 `B` 需要紧盯对象 `A`，只要对象 `A` 发生变化，对象 `B` 就采取某种动作。在设计模式中，观察者模式恰好对应此种场景。读者请查阅对应资料，并且应用到当前的贪吃蛇场景中。

剩下的 `Control` 部分主要处理玩家的输入，监听来自键盘的输入，确定移动方向并且设置贪吃蛇的运动方向变量，供 `Model` 更新蛇头节点的 x 和 y 值使用。

贪吃蛇开发的实操记录：

以初学者的眼光看上述文字，虽然明白个大概，但如何入手进行实际开发？估计还是茫然不知所措。因此，本节主要采用实录地写法，详细记录开发贪吃蛇游戏的每一步，供初学者按部就班学习。前文已经说过，贪吃蛇的开发应从 `View` 入手。实际动手时，目标不要一下子定得太大，每次一个小目标，一步一个脚印地向大目标靠近。

小目标 1: 实现在白底背景色中绘制一个黑色小方块, 先不要求运动。这个小目标又该如何启动呢? 可参考的方式: 去网上查找完整的贪吃蛇游戏, 抽取相关的界面代码。假设第一步的代码框架如下:

```
import javax.swing.*;
import java.awt.*;

public class SnakeView
{
    JFrame mainFrame;
    Canvas paintCanvas;
    JLabel labelScore;

    public static final int canvasWidth = 200;
    public static final int canvasHeight = 300;

    public static final int nodeWidth = 10;
    public static final int nodeHeight = 10;

    public SnakeView(){

        mainFrame = new JFrame("GreedSnake");
        Container cp = mainFrame.getContentPane();

        labelScore = new JLabel("Score:");
        cp.add(labelScore, BorderLayout.NORTH);

        //game area;
        paintCanvas = new Canvas();
        paintCanvas.setSize(canvasWidth+1, canvasHeight+1);
        cp.add(paintCanvas, BorderLayout.CENTER);

        mainFrame.pack();
        mainFrame.setResizable(false);
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setVisible(true);

        repaint();
    }

    void repaint(){
        Graphics g = paintCanvas.getGraphics();

        g.setColor(Color.WHITE);
        g.fillRect(0, 0, canvasWidth, canvasHeight);
    }
}
```

```

        g.setColor(Color.BLACK);
        g.fillRect(2*nodeWidth, 3*nodeHeight, nodeWidth-1, nodeHeight-1);
    }

    public static void main(String args[])
    {
        new SnakeView();
    }
}

```

代码中的 `repaint()` 方法实现我们的白底黑块绘制：

```

        g.setColor(Color.WHITE);
        g.fillRect(0, 0, canvasWidth, canvasHeight);

```

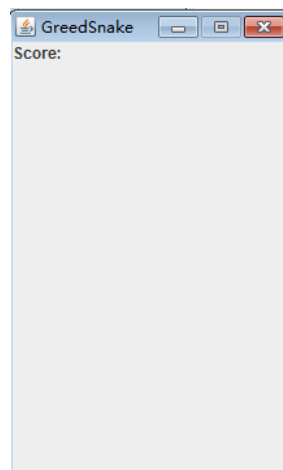
将整个的游戏界面绘制为白色；

```

        g.setColor(Color.BLACK);
        g.fillRect(2*nodeWidth, 3*nodeHeight, nodeWidth-1, nodeHeight-1);

```

以坐标 `x=20, y=30` 为起点，绘制一个 `10*10` 大小的小黑块。实际运行代码的结果：



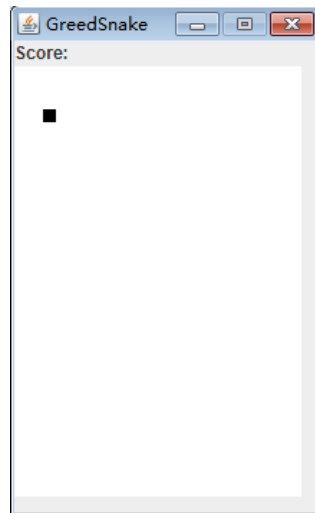
图中并没有出现白色底和黑色的方块，问题在哪呢？现实开发中，总会遇到出乎意料的问题，这也是最考验人耐心和毅力的时候，所以读者遇到问题要将其视为学习和磨练自己的机会，要认真地坚持下来。经过一番查找和对比后，可发现主要的问题在 `repaint()` 方法，上述代码的 `repaint()` 只被执行一次是不对的，正确的做法是周期性地运行。主要的修改如下：

```

    public void update(){
        while(true){
            repaint();
            try{
                Thread.sleep(500);
            }catch(InterruptedException e){
            }
        }
    }
}

```

将 `repaint()` 方法放置到 `while` 循环中，周期性地更新，运行代码符合小目标 1。



完整的代码见 SnakeView1 文件夹。

增加类 `node` 描述蛇体节点，增加 `LinkedList` 数据结构，并且将 `x=2` 和 `y=3` 的数据节点加入到 `LinkedList` 中，重构 `SnakeView.java` 代码，实现与小目标 1 同样的功能。代码的重构主要方便后续的开发。

小目标 2：让小方块沿着界面边沿顺时针移动。

要实现顺时针的运动，需要周期性地按如下方式更新方块的 `x` 和 `y` 值，具体的代码见 `SnakeView2` 文件夹。

```
public void updateSnake(){
    Node nd = Snake.get(0);
    int x = nd.getX();
    int y = nd.getY();

    if(y==0 && x<widthCount-1){
        x++;
    }else if(x==widthCount-1 && y<heightCount-1){
        y++;
    }else if(x>0 && y==heightCount-1){
        x--;
    }else if(x==0 && y>0){
        y--;
    }
    nd.setX(x);
    nd.setY(y);
}
```

小目标 3：增加键盘控制部分，让小方块根据上下左右方向按键移动。

增加对应的方向常量：

```
public final static int UP = 0;
public final static int DOWN = 3;
```

```
public final static int LEFT = 1;
public final static int RIGHT = 2;
private int currentDir;
```

变量 `currentDir` 存储最近一次按键的方向。需要留意一种特殊的情况：玩家按键的方向与当前蛇头运动方向相反。假设 `dir` 变量存储当前一次按键变量，根据方向变量常数值，只要 `dir+currentDir` 的值等于 3，就可以捕捉到特殊情况。方向更新的主体代码如下：

```
public void updateSnake(){
    Node nd = Snake.get(0);

    if(curDir + preDir == 3){ //按键与蛇头运动方向矛盾；
        nd = dirUpdate(nd, preDir);
    }else{
        nd = dirUpdate(nd, curDir);
        preDir = curDir;
    }
}
```

```
public Node dirUpdate(Node nd, int dir){

    int x = nd.getX();
    int y = nd.getY();

    if(dir == UP){
        y--;
    }else if(dir == DOWN){
        y++;
    }else if(dir == LEFT){
        x--;
    }else if(dir == RIGHT){
        x++;
    }

    nd.setX(x);
    nd.setY(y);

    return nd;
}
```

完整的代码见 `SnakeView3` 文件夹。

小目标 4：设定贪吃蛇长度 5 个节点，并且可通过上下左右按键控制蛇头运动方向。

```
public void snakeInit(){
    int i;
    for(i=3; i<3+5; i++)
    {
```

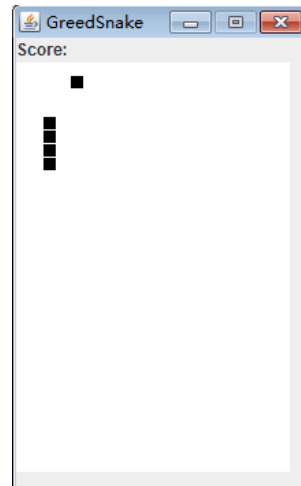


```

        Snake.addLast(new Node(2, i));
    }
}

```

上述代码设置 5 个节点长度的蛇体，但显示存在问题，只显示一个节点，并没有出现预期的 5 个节点。仔细检查后，可发现问题在 `drawNode()` 方法中。读者可自行修改下。修改后，又出现如下问题：



蛇身不能做到与蛇头同步运动，问题可能出在蛇身位置更新部分的代码。仔细检查后，问题出现如下：

```

public void updateBody(){

    int size = Snake.size();

    Node temp = new Node(0, 0);
    Node head = Snake.get(0);

    temp.setX(head.getX());
    temp.setY(head.getY());

    Node pre = new Node(0, 0);
    Node next = null;

    for(int i=1; i<size; i++){
        pre=temp;    //主要的错误在此处;

        next = Snake.get(i);
        temp.setX(next.getX());
        temp.setY(next.getY());

        next.setX(pre.getX());
        next.setY(pre.getY());
    }
}

```

注释处的代码修改为如下即可：

```
pre.setX(temp.getX());
pre.setY(temp.getY());
```

可运行的代码见 `snakeview4` 文件夹。目前为止，贪吃蛇雏形已经有了，后续还有很多细节要处理，比如吃豆子，如何判断咬到自身以及撞墙等。深入细节之前，我们要停一下（为什么这时候要停一下？靠经验，没有什么为什么。），将现有的代码重构下，方便后续的开发。现有代码的框架体现在下列代码中：

```
public void update(){
    while(true){
        updateSnake();
        drawSnake();

        try{
            Thread.sleep(1000);
        }catch(InterruptedException e){
        }
    }
}
```

也即每隔 1s，贪吃蛇的 `Model` 会更新，然后 `View` 基于更新后的 `Model` 数值，重新绘制蛇体，产生动画的效果。这样的实现将 `Model` 部分与 `View` 部分耦合在一起（因为 `updateSnake()` 方法属于 `Model` 而 `drawSnake` 部分属于 `View`），违反 MVC 应该将 `Model` 与 `View` 分离的开发原则。对当前代码进行重构的一个主要目的就是 `Model` 部分代码与 `View` 分离。我们已经提过：`Model` 对 `View` 可见，并且 `Model` 是 `View` 的观察对象，每次 `Model` 一发生改变，则触发 `View` 也对应地更新。按照上述原则，将当前 `SnakeView` 中的 `Model` 有关的代码抽离，集中到 `SnakeModel` 新类中，并且采用观察者模式整合两者。如果读者对观察者模式不熟悉，暂停下手头的开发工作，去查阅 Java 下观察者模式的基本使用，玩几个小例子，熟悉后再应用到当前的开发工作中。

`SnakeModel` 更新通知 `SnakeView` 后进入休眠状态，而 `SnakeView` 则开始绘制工作，因此 `SnakeModel` 应该与 `SnakeView` 隶属两个不同的线程，否则会产生干扰。我们选择将 `SnakeModel` 线程化。

完成开发后运行后，出现屏幕没有出现更新的情况，原始代码见 `snakeview5` 文件夹。请读者自己阅读源代码，找出问题并且修正。修正后，又出现键盘无法控制蛇头运动的问题，估计 `SnakeView` 对键盘响应后，没有把 `curDir` 的值传递给 `SnakeModel` 所致。请读者尝试自己修正问题。重构后可运行的完整代码见 `snakeview6` 文件夹。

小目标 5：增加豆子，贪吃蛇吃到豆子后增长。

豆子是 `Node` 对象的一个实例，需要考虑以下问题：

1. 豆子的 `x` 和 `y` 值不能与蛇体中任意一节的 `x` 和 `y` 值相同，否则出现某个小方块重复绘制的情况，界面中会看不到豆子；
2. 如何判定贪吃蛇是否吃到豆子？当豆子的坐标与更新后的贪吃蛇的蛇头坐标相同，则可以确定豆子会被贪吃蛇吃到。此时，当前轮蛇体的各个坐标值就不要更新，只要将豆子添加到 `LinkedList` 的头部即可，也就是说豆子变成头部，相当于蛇往前进一步；
3. 当豆子被贪吃蛇吃掉以后，立即生成新的豆子，出现在界面中的任一位置，当然前提是不能与蛇体重叠；

根据上述思路：**SnakeView** 只要增加关于豆子的绘制动作即可，而 **SnakeModel** 则要负责豆子的生成与被吃等操作。生成豆子的基本思路：

1. 生成一个 0~600 之间的整数随机数，除整和取模后，获得 x 和 y 值；
2. 遍历当前蛇体的每个节点，判断是否有节点与 1 中的 x 和 y 值相等；如果不存在这样的节点，步骤 1 中生成的 x 和 y 值可作为新豆子的坐标；否则重复步骤 1 和 2；

代码中需要遍历 **LinkedList** 中的每个节点，我们使用迭代器 **Iterator** 实现遍历而不是 **get**。这个问题值得读者耐下心来好好研究清楚，因为遍历是日常编程中经常使用的，十分重要；另外，问题的求解过程也涉及到验证实验的设计与源代码的阅读，是个很好的学习案例。

完成豆子的生成后，考虑如何实现吃豆子？主要考虑两种情况：

1. 豆子在蛇头下一步移动方向；
2. 豆子不在蛇头下一步移动方向；

对于第二种情况，按之前的逻辑处理，不需要改动；对于第一种情况，需要将豆子加入到 **LinkedList** 的头部，而原来的 **LinkedList** 不需要更新（这么实现的话，视觉上不流畅，所以最终实现时，将豆子添加到头部后，再次更新整个蛇体。）；实现时，只要先判断下豆子是否在蛇头下一格方向，然后再分上述两种情况处理即可。

完整的实现代码见文件夹 **snakeview7**。运行测试代码后，可以发现如下的两个问题：

1. 贪吃蛇偶尔对按键反应迟钝；
2. 可能出现豆子与蛇体重叠的情况，尤其是当蛇体长度比较长的时候；

对问题 1，发生按键时，**SnakeModel** 可能正处于休眠状态，故不会对按键反应。所以每次按键后，就应该触发 **SnakeView** 的绘制操作。请读者自行完成代码的修改。

问题 2 很容易定位到方法 **genBean** 中，请读者仔细检查该方法，找到问题原因，并修正。

修正问题后的代码见文件夹 **snakeview8**。目前为止，贪吃蛇游戏的大致框架已经完成，后续要完成的还有：

1. 记分：界面上显示当前的得分，每吃一颗豆子，加 10 分；
2. 判断蛇碰墙和咬到自己：识别蛇碰墙和咬到自己的情况，并结束游戏；
3. 改变贪吃蛇移动的速度，有两种方式供参考：得分每增加 300，速度提升一级；设置专门的按键，比如 **PageUp** 提升贪吃蛇速度，**PageDown** 降低速度；

请读者实现上述 3 个细节，使得贪吃蛇游戏更加完整。

练习：

问题 1：请读者在 **snakeview8** 代码的基础上，完成本章最后提到的 3 个细节的实现，让整个贪吃蛇游戏更加完整。

问题 2：请读者完成扫雷游戏的 **Model** 开发，并且与《Java 界面开发》中的扫雷界面整合，完成完整的扫雷游戏的开发。