2、具体内容

在实际的项目开发过程之中，肯定要一直存在有包的概念，利用包可以实现类的包装，在以后的实际开发之中，所有的类都必须放在包里面。

■包的定义与使用

对于项目而言，尤其是现代的项目是不可能一个人开发完成的，往往在一个项目的开发团队之中有多人开发者进行项目业务的实现，于是在这样的情况下就不得不去面对一个问题，有可能产生类的重用定义。

在操作系统之中已经明确严格的定义了一个要求：同一个目录之中不允许存放有相同的程序类文件，但是在程序开发之中很难保证类的不重复，所以为了可以进行类的方便管理，那么往往可以将程序文件放在不同的目录下，不同的目录之中是可以提供有相同文件的，而这个目录就称为包；包=目录。

范例：定义包

```
package cn.mldn.demo ; // 定义包，其中.表示分割子目录（子包）
public class Hello {
	public static void main(String args[]) {
		System.out.println("Hello World !") ;
	}
}
```

一旦程序开发之中出现有包，此时程序编译后的结果就必须将*.class文件保存在指定的目录之中，但是如果手工建立则非常麻烦，那么此时最好的做法是可以进行打包编译处理：javac -d . Hello.java

- “-d”：表示要生成目录，而目录的结构就是package定义的结构；
- “.”：表示在当前所在的目录中生成程序类文件；

在程序执行的时候一定要待着包执行程序类 ： java cn.mldn.demo.Hello，也就是说从此之后完整的类名称是“包.类”名称。

■包的导入

利用包的定义实际上就可以将不同的功能的类保存在不同的包之中，但是这些类彼此之间也一定会存在有相互调用的关系，那么在这个时候就需要用import语句来导入其他包中的程序类。

范例：定义一个程序类“cn.mldn.util.Message”，这个类负责进行一个消息数据获取

```
package cn.mldn.util ;
public class Message {
	public String getContent() {
		return "www.mldn.cn" ;
	}
}
```

范例：定义一个测试类使用Message类“cn.mldn.test.TestMessage”

```
package cn.mldn.test ;
import cn.mldn.util.Message ; // 导入其它包的类
public class TestMessage {
	public static void main(String args[]) {
		Message msg = new Message() ; // 实例化类对象
		System.out.println(msg.getContent()) ;
	}
}
```

此时按照使用顺序来讲肯定要先编译Message.java，而后再编译TestMessage.java，但是你思考一个问题，如果现在你写了一个程序代码，里面有100各类，彼此之间互相引用严格，此时你怎么区分？那么这个时候最好的做法不是区分顺序，而是直接交给Java自己完成，编译的命令：javac -d . *.java;

注意：关于public class 与class定义的区别?
·public class：类名称必须与文件名称保持一致，一个*.java文件里面只允许有一个public class，同时一个类需要被其它的包所使用，那么这个类一定要定义为public class;
·class：类名称可以与文件名不一致，并且在一个*.java文件里面可以提供有多个class定义，编译后将形成不同的*.class文件，但是这些类只能够被本包所访问，外包无法访问。
·在实际的开发之中往往在一个*.java源代码里面只会提供有一个程序类，而这个程序类一般都使用pubic clss定义：
程序类中定义的包名称必须采用小写字母的形式定义，例如：cn.mldn.util;

但是这个时候会有一个新的问题产生了，有些时候可能会使用某一个保中的很多类，于是这样分开进行类的导入会比较麻烦，为了解决这样大问题，也可以使用通配符“*”来处理。

```
package cn.mldn.test ;
import cn.mldn.util.* ; // 导入其它包的类
public class TestMessage {
	public static void main(String args[]) {
		Message msg = new Message() ; // 实例化类对象
		System.out.println(msg.getContent()) ;
	}
```

```
}
```

即便此时使用了“包.*”的导入形式，那么也不表示要进行全部的加载，它会根据自己的需要加载所需要的程序类，而不需要的程序类是不会被加载的，所以使用“*”还是使用具体的类其最终的性能是完全相同的。

但是如果在开发智障采用的是“包.*”的形式进行包的导入时，那么有一点会比较麻烦：有可能两个不同的包中存在相同的类名称，例如，现在假设TESTMessage类由于某种需要导入两个包：cn.mldn.util、org.demo，但是这两个包里面都有Message类。

cn.mldn.util.Message：

```
package cn.mldn.util ;
public class Message {
 public String getContent() {
  return "www.mldn.cn" ;
 }
}
```

org.demo.Message：

```
package org.demo ;
public class Message {
 public String getInfo() {
  return "人民万岁！" ;
 }
}
```

由于某种需要在TESTMessage类里面导入了两个包：

```
package cn.mldn.test ;
import cn.mldn.util.* ; // 导入其它包的类
import org.demo.* ; // 导入其它包的类
public class TestMessage {
 public static void main(String args[]) {
  Message msg = new Message() ; // 实例化类对象
  System.out.println(msg.getContent()) ;
 }
}
```

程序编译

```
javac -d . TestMessage.java
```

```
TestMessage.java:6: 错误: 对Message的引用不明确
                Message msg = new Message() ; // 实例化类对象
                ^
  org.demo 中的类 org.demo.Message 和 cn.mldn.util 中的类 cn.mldn.util.Message 都匹配
TestMessage.java:6: 错误: 对Message的引用不明确
                Message msg = new Message() ; // 实例化类对象
                              ^
  org.demo 中的类 org.demo.Message 和 cn.mldn.util 中的类 cn.mldn.util.Message 都匹配
2 个错误
```

这个时候就会发现类名称相同的时候救护出现不明确的引用处理，所以此时最简单的处理形式就是直接写上类的完成名称

```
package cn.mldn.test ;
import cn.mldn.util.* ; // 导入其它包的类
import org.demo.* ; // 导入其它包的类
public class TestMessage {
    public static void main(String args[]) {
        cn.mldn.util.Message msg = new cn.mldn.util.Message() ; // 实例化类对象
        System.out.println(msg.getContent()) ;
    }
```

```
}
```

在日后的开发过程之中经常会见到大量的重名的类（包不重名），此时为了更好的解决问题，往往会使用类的完整名称进行操作。

■包的静态导入

假如说现在有一个类，这个类中的全部方法都是static方法，那么按照原始的做法肯定要导入程序所在的“包.类”，而后才可以通过类名称调用这些静态方法；

范例：定义一个MyMath数学类

```
package cn.mldn.util ;
public class MyMath {
	public static int add(int ... args) {
		int sum = 0 ;
		for (int temp : args) {
			sum += temp ;
		}
		return sum ;
	}
	public static int sub(int x,int y) {
		return x - y ;
	}
}
```

如果此时按照原始的方式进行导入处理，那么此时就需要导入包.类，而后通过类名称调用方法。

范例：原始方式使用

```
package cn.mldn.test ;
import cn.mldn.util.MyMath ;
public class TestMath {
	public static void main(String args[]) {
		System.out.println(MyMath.add(10,20,30)) ;
		System.out.println(MyMath.sub(30,20)) ;
	}
}
```

从JDK1.5开始对于类中全部静态方法提供的特殊类时可以采用静态导入形式的。

范例：静态导入处理

```
package cn.mldn.test ;
import static cn.mldn.util.MyMath.* ;
public class TestMath {
	public static void main(String args[]) {
		System.out.println(add(10,20,30)) ;
		System.out.println(sub(30,20)) ;
	}
}
```

当使用了静态导入处理之后就好比该方法是直接定义在主类中的，可以由主方法直接调用。

■Jar命令

当一个项目开发完成之后一定会存在有大量的*.class文件，那么对于这些文件的管理往往可以利用一种压缩的结构来进行处理，而这样的结构在Java之中就被称为jar文件，如果要想程序打包为Jar文件，那么可以直接使用jdk中提供的jar命令。

在最原始的时候如果要想知道jar命令的时候直接输入jar即可，而在JDK1.9之后为了统一化，所以需要使用“--help”查看相关的说明。

```
D:\cn\mldn\demo\test> jar --help
非法选项: -
用法: jar {ctxui}[vfmn0PMe] [jar-file] [manifest-file] [entry-point] [-C dir] files ...
选项:
    -c  创建新档案
    -t  列出档案目录
    -x  从档案中提取指定的 (或所有) 文件
    -u  更新现有档案
    -v  在标准输出中生成详细输出
    -f  指定档案文件名
    -m  包含指定清单文件中的清单信息
    -n  创建新档案后执行 Pack200 规范化
    -e  为捆绑到可执行 jar 文件的独立应用程序
        指定应用程序入口点
    -0  仅存储; 不使用任何 ZIP 压缩
    -P  保留文件名中的前导 '/' (绝对路径) 和 "../" (父目录) 组件
    -M  不创建条目的清单文件
    -i  为指定的 jar 文件生成索引信息
    -C  更改为指定的目录并包含以下文件
如果任何文件为目录, 则对其进行递归处理。
清单文件名, 档案文件名和入口点名称的指定顺序
与 'm', 'f' 和 'e' 标记的指定顺序相同。

示例 1: 将两个类文件归档到一个名为 classes.jar 的档案中:
       jar cvf classes.jar Foo.class Bar.class
示例 2: 使用现有的清单文件 'mymanifest' 并
           将 foo/ 目录中的所有文件归档到 'classes.jar' 中:
       jar cvfm classes.jar mymanifest -C foo/ .
```

下面通过程序的具体演示来实现jar的使用与配置的操作。

1、定义一个程序类，这个类的代码如下：

```
package cn.mldn.util ;
public class Message {
    public String getContent() {
        return "www.mldn.cn" ;
    }
}
```

2、对程序进行编译与打包处理：

- 对程序打包编译：javac -d . Message.java ；
- 此时会形成cn的包，包里面有相应的子包与*.class文件，将其打包为mldn.jar：jar -cvf mldn.jar cn；

|- “-c”：创建一个新的jar文件；

|- “-v”：得到一个详细输出；

|- “-f”：设置要生成的jar文件名称，本处定义的是“mldn.jar”；

```
D:\cn\mldn\demo\test>jar -cvf mldn.jar cn
已添加清单
正在添加: cn/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: cn/mldn/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: cn/mldn/test/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: cn/mldn/test/TestMessage.class(输入 = 514) (输出 = 338)(压缩了 34%)
正在添加: cn/mldn/util/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: cn/mldn/util/Message.class(输入 = 295) (输出 = 224)(压缩了 24%)
```

3、每一个*.jar文件都是一个独立的程序路径，如果要想在Java程序之中使用此路径，则必须通过CLASSPATH进行配置；

```
SET CLASSPATH=.;d:\mldndemo\mldn.jar
```

4、建立测试类，直接导入Message类并且调用方法：

```
package cn.mldn.test ;
public class TestMessage {
	public static void main(String args[]) {
		cn.mldn.util.Message msg = new cn.mldn.util.Message() ; // 实例化类对象
		System.out.println(msg.getContent()) ;
	}
}
```

随后就可以正常编译TestMessage类并且使用这个类：

- 编译程序类：javac -d . TestMessage.java;
- 解释程序类：java cn.mldn.test.TestMessage;

如果此时程序编译通过之后，但是由于CLASSPATH发生了改变，类无法加载到了，则执行TestMessage类的时候将胡出现如下的错误提示：

```
Exception in thread "main" java.lang.NoClassDefFoundError: cn/mldn/util/Message
```

出现这种错误只有一种情况：*.jar包没有配置正确。

JDK1.9之后出现的模块化操作

- 在JDK1.9以前所有的历史版本之中实际上提供的是一个所有类的*.jar文件（rt.jar、tools.jar），在传统的开发之中只要启动了Java虚拟机，那么就需要加载这几十兆的类文件；
- 在JDK1.9之后提供了一个模块化的设计，将原本很大的要加载的一个*.jar文件变成了若干个模块文件，这样在启动的时候可以根据加载指定的模块（模块中有包），就可以实现启动速度变快的效果。

■系统常见包

Java语言从发展至今一直提供有大量的支持类库，这些类库一般由两个方面组成：

• Java自身提供的 （除了JDK提供的类库之外还有一些标准）；

• 由第三方厂商提供的Java支持类库，可以完成各种所需要的功能，并且支持的厂商很多；

而在JDK之中也会提供大量的类库，并且这些类库都是封装在不同的开发包之中；

• java.lang：像String、Number、Object等类都在这个包里面，这个包在JDK1.1之后自动默认导入；

• java.lang.reflect：反射机制处理包，所有的设计从此开始；

• java.util：工具类的定义，包括数据结构的定义；

• java.io：进行输入与输出流操作的程序包；

• java.net：网络程序开发的程序包；

• java.sql：进行数据库编程的开发包；

• java.applet：Java的最原始的使用形式，直接嵌套在网页上执行的程序类；

|- 现在的程序以及以Application为主（有主方法的程序）；

• java.awt、javax.swing：Java的图形界面开发包（GUI），其中awt是属于重量级的组件，而swing是轻量级的组件。

2、具体内容

在面向对象的开发过程之中有三大主要特点：封装、继承、多态。那么对于封装性而言主要的实现依靠的就是访问控制权限，而访问控制权限在程序之中一共定义有四种：private、default（不写）、protected、public，这四种权限的作用如下：

No	访问范围	private	default	protected	public
01	同一包中的同一类	√	√	√	√

02	同一包中的不同类		√	√	√
03	不同包的子类			√	√
04	不同包的所有类				√

在整个访问控制权限之中，只有protected（受保护）的权限是比较新的概念，那么下面对这一访问权限的使用进行说明，本次要定义两个类：

- cn.mldn.a.Message类：提供有protected访问权限；
- cn.mldn.b.NetMessage类：将直接访问protected属性；

范例：定义Message类

```
package cn.mldn.a ;
public class Message {
	protected String info = "www.mldn.cn" ;
}
```

范例：定义子类，与父类不在同一个包中

```
package cn.mldn.b ;
import cn.mldn.a.Message ;
public class NetMessage extends Message {
	public void print() {
		System.out.println(super.info) ;
	}
}
```

范例：编写测试类，通过子类实现操作

```
package cn.mldn.test ;
import cn.mldn.b.* ;
public class TestMessage {
	public static void main(String args[]) {
		new NetMessage().print() ;
	}
}
```

此时的程序是通过子类访问了父类中的protected属性。但是如果说此时你直接通过了Message访问info属性，那么将会出现错误的提示。

范例：在测试类中直接访问Message

```
package cn.mldn.test ;
import cn.mldn.a.* ;
public class TestMessage {
	public static void main(String args[]) {
		System.out.println(new Message().info) ;
	}
}
```

```
TestMessage.java:5: 错误: info 可以在Message中访问protected
```

在程序之中的封装一共有三个对应的访问权限：private、default、protected，但是如果每次在使用的时候进行区分会比价麻烦，所以可以给出一个参考的选择方案（90%的设

计问题）：

- 只要是进行属性的定义，全部使用private；
- 只要是进行方法的定义，全部使用public；