第七章

包和模块

程序被组织为一组包。一个包(§7.1)的成员是类和接口，它们在包的编译单元和子包中声明，子包可以包含它们自己的编译单元和子包。

每个包都有自己的类和接口名称集，这有助于防止名称冲突。包的命名结构是分层的。

如果一组包具有足够的内聚性，则可以将这些包分组到一个模块中。模块将其导出的部分或全部包分类，这意味着可以从模块外部的代码访问它们的类和接口。如果一个包不是由模块导出的，那么只有模块内部的代码可以访问它的类和接口。此外，如果一个模块中的代码希望访问由另一个模块导出的包，那么第一个模块必须显式地依赖于第二个模块。因此，一个模块控制它的包如何使用其他模块(通过指定依赖项)，并控制其他模块如何使用它的包(通过指定导出它的哪个包)。

模块和包可以存储在文件系统或数据库中(§7.2)。存储在文件系统中的模块和包可能对其编译单元的组织有一定的约束，从而允许一个简单的实现轻松地查找模块、类和接口声明。

编译单元中的代码可以自动访问其包中声明的所有类和接口，还可以自动导入预定义包 java.lang 中声明的所有公共类和接口。

只有在类或接口声明为公共的情况下，顶级类或接口才能在声明它的包之外访问（§6.6）。只有当类或接口声明为公共的并且是导出包的成员时，顶级类或接口才能在声明它的模块外部访问。声明为公共但不是导出包成员的类或接口只能由模块内的代码访问。

对于小型程序和随意开发，包可以不命名（§7.4.2）或具有简单名称，但如果代码要广泛分发，则应使用限定名称选择唯一的包名称。如果两个开发组碰巧选择了相同的包名，并且这些包后来在单个程序中使用，这可以防止冲突的发生。

7.1 包成员

包的成员是其子包以及包的所有编译单元（§7.3）中声明的所有顶级类（§8（类））和顶级接口（§9（接口））。

例如，在 Java SE 平台 API 中：

- 包 java 有子包 awt, applet, io, lang, net, 和 util, 但没有编译单元。
- 包 java.awt 有子包名为 image, 以及许多包含类和接口声明的编译单元。

如果包的完全限定名称（§6.7）是 P，并且 Q 是 P 的子包，那么 P.Q 是子包的完全限定名称，并且进一步表示包。

一个包不能包含两个同名的成员，否则会导致编译时错误。

以下是一些例子:

- 因为包 java.awt 有一个子包 image, 它不能（也不）包含名为 image 的类或接口的声明。
- 如果有一个包名为 mouse 以及一个成员类 Button 在那个包里(然后可以将其称为 mouse.Button)，则不能有任何具有完全限定名称 mouse.Button 或 mouse.Button.Click 的包。
- 如果 com.nighthacks.java.jag 是类的完全限定名，那么就不能有任何包的完全限定名为 com.nighthacks.java.jag 或 com.nighthacks.java.jag.scrabble。

但是，不同包的成员可能具有相同的简单名称。例如，可以声明一个包:

```
package vector;
public class Vector { Object[] vec; }
```

它有一个名为 Vector 的公共类作为成员，尽管包 java.util 还声明了一个名为 Vector 的类。这两个类是不同的，这反映在它们具有不同的完全限定名这一事实上（§6.7）。这个例子中 Vector 的完全限定名是 vector.Vector, 而 java.util.Vector 是 Java SE 平台包含的类 Vector 的完全限定名。因为包 vector 包含一个名为 Vector 的类，它不能再有一个名为 Vector 的子包。

包的分层命名结构旨在便于以常规方式组织相关包，但其本身没有意义，只是禁止具有与该包中声明的顶级类或接口相同简单名称的子包（§7.6）。

例如，名为 oliver 的包以及名为 oliver.twist 的包或者名为 evelyn.wood 的包以及名为 evelyn.waugh 的包之间没有特殊的访问关系。那也就是说，在名为 oliver.twist 的包里的代码并没有必任何其他包里的代码对包 oliver 里声明的类和接口有更好的访问。

7.2 主机对模块和包的支持

每个主机系统决定如何创建和存储模块、包和编译单元。

每个主机系统决定在一个特定的编译中哪些编译单元是可观察的(§7.3)。每个主机系统还确定哪些可观察编译单元与某个模块相关联。与模块相关的编译单元的可观察性决定了哪些模块是可观察的(§7.7.3)，哪些包在这些模块中是可见的(§7.4.3)。

主机系统可以自由地确定包含模块声明的编译单元实际上不是可观察对象，因此与其中声明的模块没有关联。这使得编译器能够选择 modulesourcepath 上的哪个目录“真正”是给定模块的体现。但是，如果主机系统确定包含模块声明的编译单元是可观察的，那么§7.4.3 要求编译单元必须与其中声明的模块关联，而不能与任何其他模块关联。

主机系统可以自由地确定包含类或接口声明的编译单元是(第一个)可观察对象，(第二个)与未命名模块或自动模块相关联——尽管在任何编译单元、可观察对象或其他类型中都不存在未命名模块或自动模块的声明。

在 Java SE 平台的简单实现中，包和编译单元可以存储在本地文件系统中。其他实现可能使用分布式文件系统或某种形式的数据库存储它们。

如果主机系统在数据库中存储包和编译单元，则数据库不得对基于文件的实现中允许的编译单元施加可选限制（§7.6）。

例如，使用数据库存储包的系统可能不会强制每个编译单元最多使用一个公共类或接口。

但是，使用数据库的系统必须提供一个选项，将程序转换为符合限制的形式，以便导出到基于文件的实现。

作为在文件系统中存储包的一个非常简单的示例，项目中的所有包以及源代码和二进制代码都可能存储在单个目录及其子目录中。该目录的每个直接子目录将代表一个顶级包，即完全限定名由单个简单名称组成的包。每一级子目录都代表包含目录所代表的包的子包，依此类推。

该目录可能包含以下直接子目录:

```
com
gls
jag
java
wnj
```

其中目录 java 包含 Java SE 平台的包; 目录 jag, gls, 和 wnj 可能包含本规范的三位作者为其个人使用而创建的包，并在这个小组内彼此共享; 目录 com 将包含从使用§6.1 所述约定为其包生成唯一名称的公司获得的包。

继续这个例子，目录 java 还将包含以下子目录:

```
applet
awt
io
lang
net
util
```

对应于定义为 Java SE 平台 API 一部分的包 java.applet, java.awt, java.io, java.lang, java.net, 和 java.util。

继续这个例子，如果我们查看目录 util，我们可能会看到以下文件:

```
BitSet.java         Observable.java
BitSet.class        Observable.class
Date.java           Observer.java
Date.class          Observer.class
```

每个.java 文件都包含一个编译单元(§7.3)的源代码，该编译单元包含一个类或接口的定义，其二进制编译形式包含在相应的.class 文件中。

在这种简单的包组织方式下，Java SE 平台的实现将通过连接包名的组件将包名转换为路径名，在相邻组件之间放置一个文件名分隔符(目录指示符)。

例如，如果在文件名分隔符为/的操作系统上使用这个简单的组织方式，则包名:

```
jag.scrabble.board
```

将被转换为目录名:

```
jag/scrabble/board
```

包名组件或类名可能包含不能正确出现在主机文件系统的普通目录名称中的字符，例如在只允许在文件名中包含 ASCII 字符的系统上的 Unicode 字符。作为一种约定，字符可以使用转义，比如@字符后面跟着四个十六进制数字，给出字符的数值，如\uxxxx 转义(§3.3)。

在此约定下，包名:

```
children.activities.crafts.papierM\u00e2ch\u00e9
```

也可以使用完整的 Unicode 编写为:

```
children.activities.crafts.papierMache
```

可以映射到目录名:

```
children/activities/crafts/papierM@00e2ch@00e9
```

如果@字符不是某个给定主机文件系统的文件名中的有效字符，则可以使用标识符中无效的其他字符。

7.3 编译单元

CompilationUnit 是 Java 程序语法（§2.3）的目标符号（§2.1）。 其定义如下:

CompilationUnit:
OrdinaryCompilationUnit
ModularCompilationUnit

OrdinaryCompilationUnit:
[PackageDeclaration] {ImportDeclaration}
{TopLevelClassOrInterfaceDeclaration}

ModularCompilationUnit:
{ImportDeclaration} ModuleDeclaration

普通编译单元由三个部分组成，每个部分都是可选的:

- 一种包声明（§7.4），给出编译单元所属包的完全限定名称（§6.7）。

  没有包声明的编译单元是未命名包的一部分（§7.4.2）。

- 导入声明(§7.5)，这些声明允许来自其他包的类和接口，以及类和接口的静态成员，使用它们的简单名称来引用。

- 类和接口的顶级声明(§7.6)。

模块编译单元由模块声明(§7.7)组成，前面有 import 声明(可选)。导入声明允许本模块和其他模块中的包中的类和接口，以及类和接口的静态成员在模块声明中使用它们的简单名称来引用。

每个编译单元隐式导入预定义包 java.lang 中声明的每个公共类或接口，就像声明 import

java.lang.*;出现在每个编译单元的开头，紧跟在任何包声明之后。因此，所有这些类和接口的名称在每个编译单元中都可以作为简单名称使用。

主机系统决定哪些编译单元是可观察的，除了预定义包 java 及其子包 lang 和 io 中的编译单元，它们都是可观察的。

每个可观察对象编译单元可以与一个模块相关联，如下所示:

- 主机系统可以确定一个可观察的普通编译单元与主机系统选择的模块相关联，除了(i)预定义包 java 及其子包 lang 和 io 中的普通编译单元，它们都与 java.base 模块相关联，以及(ii)未命名包中与§7.4.2 中指定的模块相关联的任何普通编译单元。
- 主机系统必须确定一个可观察模块编译单元与该模块编译单元声明的模块相关联。

  编译单元的可观察性影响其包的可观察性(§7.4.3)，而可观察编译单元与模块的关联影响该模块的可观察性(§7.7.6)。

当编译与模块 M 相关的模块和普通编译单元时，主机系统必须尊重 M 的声明中指定的依赖项。具体来说，主机系统必须将普通的可观察的编译单元限制为只对 M 可见的编译单元。对 M 可见的普通编译单元是与 M 读取的模块相关联的可观察的普通编译单元。M 读取的模块由解析结果给出，如 java.lang.module 包规范中所述，其中 M 是唯一的根模块。主机系统必须执行解析，确定 M 读取的模块;如果由于 java.lang.module 包规范中描述的任何原因导致解析失败，则会出现编译时错误。

可读性关系是自反的，因此 M 读取自身，因此与 M 相关的所有模块化和普通编译单元对 M 都是可见的。

M 读取的模块驱动对 M 唯一可见的包(§7.4.3)，这反过来驱动与 M 相关的模块化和普通编译单元中代码的作用域内的顶级包和包名的含义(§6.3、§6.5.3、§6.5.5)。

上述规则确保模块编译单元中注解（特别是应用于模块声明的注解）中使用的包和类型名称被解释为好像它们出现在与模块相关联的普通编译单元中。

在不同的普通编译单元中声明的类和接口可以循环地相互引用。Java 编译器必须安排同时编译所有此类类和接口。

7.4 包声明

包声明出现在普通编译单元中，以指示编译单元所属的包。

7.4.1 命名包

普通编译单元中的包声明指定编译单元所属包的名称（§6.2）。

PackageDeclaration:
  {PackageModifier} `package` *Identifier {. Identifier}* `;`

PackageModifier:

包声明中提到的包名称必须是包的完全限定名称（§6.7）。

§6.3 和§6.4 规定了包声明的作用域和遮蔽。

有关包声明注解修饰符的规则在§9.7.4 和§9.8.5 中有规定。

对于给定的包，最多允许一个带注解的包声明。

实施这一限制的方式必须因实施情况的不同而有所不同。T 强烈建议以下方案用于基于文件系统的实现：唯一的带注解的包声明(如果存在的话)被放置在一个名为 package-info.java 的源文件中，该文件位于包含包源文件的目录中。这个文件不包含名为 package-info 的类的源文件;实际上，这样做是非法的，因为 package-info 不是合法的标识符。通常，package-info.java 只包含一个包声明，前面紧跟着包上的注解。虽然文件在技术上可以包含一个或多个具有包访问权限的类的源代码，但这将是非常糟糕的形式。

建议在 javadoc 和其他类似的文档生成系统中使用 package-info.java(如果存在的话)代替 package.html。如果存在这个文件，文档生成工具应该查找 package-info.java 中紧挨着(可能带注解的)包声明前面的包文档注释。通过这种方式，package-info.java 成为包级注解和文档的唯一存储库。 如果将来需要添加任何其他包级别的信息，那么这个文件应该可以方便地存放这些信息。

7.4.2 未命名包

普通的编译单元没有包声明，但至少有一种其他类型的声明，它是未命名包的一部分。

Java SE 平台提供未命名名包主要是为了在开发小型或临时应用程序或刚开始开发时方便。

未命名的包不能有子包，因为包声明的语法总是包含对已命名顶级包的引用。

Java SE 平台的实现必须至少支持一个未命名的包。一个实现可以支持多个未命名的包，但不是必须这样做。每个未命名包中有哪些普通编译单元由主机系统决定。

主机系统必须将未命名包中的普通编译单元与未命名模块(§7.7.5)关联起来，而不是一个已命名模块。

例子 7.4.2-1. 未命名包

编译单元:

```
class FirstCall {
    public static void main(String[] args) {
        System.out.println("Mr. Watson, come here. "
                           + "I want you.");
    } }
```

将一个非常简单的编译单元定义为未命名包的一部分。

在使用分层文件系统存储包的 Java SE 平台的实现中，一个典型的策略是将一个未命名的包与每个目录关联; 一次只能观察到一个未命名的包，即与“当前工作目录”相关联的包。“当前工作目录”的确切含义取决于主机系统。

7.4.3 包的可观察性和可见性

一个包是可观察的，当且仅当以下至少有一个为真:

- 包含包声明的普通编译单元是可观察的 (§7.3)。
- 包的子包是可观察的。

包 java, java.lang, 和 java.io 总是可观察的。

我们可以从上面的规则和可观察的编译单元的规则中得出这个结论，如下所示。预定义包 java.lang 声明了类 Object, 所以 Object 的编译单元总是可观察的 (§7.3)。因此, java.lang 包是可观察的，并且 java 包也是可观察的。 此外,因为 Object 是可观察的，数组类型 Object[]隐式地存在。它的超接口 java.io.Serializable (§10.1) 也存在, 因此 java.io 包是可观察的。

当且仅当包含包声明的普通编译单元对 M 可见时，包对模块 M 可见。

包可见性意味着一个包对给定模块来说是可以观察到的。仅仅因为子包 P.Q 是可观察的，就知道包 P 是可观察的，这通常是没有用的。例如，假设 P.Q 是可观察的(在模块 M1 中)，而 P.R 是可观察的(在模块 M2 中); 那么，P 是可观察的，但是在哪里呢?M1 还是 M2，或者两者都是? 这个问题是多余的;在编译只需要 M1 的模块 N 时，P.Q 是可观察的有关系，但 P 是可观察的没有关系。

当且仅当下列条件之一成立时，一个包对模块 M 是唯一可见的:

- 与 M 相关联的普通编译单元包含包的声明，M 不读取任何其他将包导出到 M 的模块。
- 与 M 相关联的普通编译单元不包含包的声明，M 只读取另一个将包导出到 M 的模块。

7.5 导入声明

导入声明允许一个命名类、接口或静态成员通过一个由单一标识符组成的简单名称(§6.2)来引用。

如果不使用适当的导入声明，对另一个包中声明的类或接口的引用，或对另一个类或接口的静态成员的引用，通常需要使用完全限定名(§6.7)。

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration
SingleStaticImportDeclaration
StaticImportOnDemandDeclaration

- 单一类型导入声明（§7.5.1）通过提及其规范名称（§6.7）导入单个命名类或接口。
- 类型按需导入声明（§7.5.2）通过提及包、类或接口的规范名称，根据需要导入命名包、类和接口的所有可访问类和接口。
- 单个静态导入声明（§7.5.3）通过提供其规范名称，从类或接口导入具有给定名称的所有可访问静态成员。
- 静态按需导入声明（§7.5.4）通过提及类或接口的规范名称，根据需要导入命名类或接口所有可访问的静态成员。

§6.3 和§6.4 规定了这些声明导入的类、接口或成员的作用域和遮蔽。

导入声明使类、接口或成员仅在实际包含导入声明的编译单元中以其简单名称可用。导入声明引入的类、接口或成员的作用域不包括同一包中的其他编译单元、当前编译单元中的其他导入声明或当前编译单元（包声明的注解除外）中的包声明。

7.5.1 单一类型导入声明

单类型导入声明通过提供其规范名称导入单个类或接口，使其在出现单类型导入声明的编译单元的模块、类和接口声明中以简单名称可用。

SingleTypeImportDeclaration:
`import` *TypeName* `;`

TypeName 必须是类或接口的规范名 (§6.7)。

类或接口必须是命名包的成员，或者是其最外层词法封闭类或接口声明（§8.1.3）是命名包成员的类或接口的成员，否则会发生编译时错误。

如果命名类或接口不可访问，则为编译时错误 (§6.6)。

如果同一编译单元中的两个单类型导入声明试图导入具有相同简单名称的类或接口，则会发生编译时错误，除非两个类或接口相同，在这种情况下，重复的声明将被忽略。

如果由单一类型导入声明导入的类或接口在包含导入声明的编译单元中声明为顶级类或接口（§7.6），则忽略导入声明。

如果单个类型导入声明导入简单名称为 x 的类或接口，并且编译单元还声明简单名称为 x 的顶级类或接口时，则会发生编译时错误。

如果编译单元既包含导入简单名称为 x 的类或接口的单一类型导入声明，也包含导入简单名字为 x 的类或接口的单一静态导入声明（§7.5.3），则会发生编译时错误，除非两个类或接口相同，在这种情况下，重复的声明将被忽略。

例子 7.5.1-1. 单类型导入

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

使简单名称 Vector 在编译单元中的类和接口声明中可用。因此，简单名称 Vector 指向 java.util 包中的类声明 Vector，在未被字段、参数、局部变量或具有相同名称的嵌套类或接口声明遮蔽（§6.4.1）或遮掩（§6.4.2）的所有位置。

注意 java.util.Vector 的真实声明是泛型 (§8.1.2)。一旦导入，名称 Vector 就可以在参数化类型(如 Vector<string>)中不加限制地使用，或者作为原始类型 Vector 使用。导入声明的一个相关限制是，可以导入泛型类或接口声明中声明的成员类或接口，但其外部类型总是被擦除。

例子 7.5.1-2. 重复的类声明

程序:

```
import java.util.Vector;
class Vector { Object[] vec; }
```

因为重复声明 Vector，导致编译时错误,下面一样:

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

其中 myVector 是一个包含编译单元的包:

```
package myVector;
public class Vector { Object[] vec; }
```

例子 7.5.1-3. 不导入子包

请注意，导入声明不能导入子包，只能导入类或接口。

例如, 尝试导入 java.util，然后使用 util.Random 来指向类型 java.util.Random，这样做是行不通的:

```
import java.util;
class Test { util.Random generator; }
  // incorrect: compile-time error
```

例子 7. 5. 1-4. 导入同时也是包名称的类型名称

在§6.1 中描述的命名约定下，包名和类型名通常是不同的。然而，在一个人为的例子中，有一个非常规命名的包 Vector，它声明了一个名为 Mosquito 的公共类:

```
package Vector;
public class Mosquito { int capacity; }
```

然后是编译单元:

```
package strange;
import java.util.Vector;
import Vector.Mosquito;
class Test {
    public static void main(String[] args) {
        System.out.println(new Vector().getClass());
        System.out.println(new Mosquito().getClass());
    }
}
```

单一类型导入声明从 java.util 包导入类 Vector 并不会阻止包名 Vector 在后续的导入声明中出现并被正确识别。这个例子编译并产生以下输出:

```
class java.util.Vector class Vector.Mosquito
```

7.5.2 按需导入类型声明

按需类型导入声明允许根据需要导入命名包、类或接口的所有可访问类和接口。

TypeImportOnDemandDeclaration:
`import` *PackageOrTypeName* `. * ;`

PackageOrTypeName 必须是包、类或接口的规范名 (§6.7)。

如果 PackageOrTypeName 表示一个类或接口 (§6.5.4), 则该类或接口必须是命名包的成员，或者是其最外层词法封闭类或接口声明（§8.1.3）是命名包成员的类或接口的成员，否则会发生编译时错误。

如果命名包对当前模块不唯一可见（§7.4.3），或者命名类或接口不可访问（§6.6），则为编译时错误。

命名 java.lang 或者在类型按需导入声明中当前编译单元的命名包，不会产生编译时错误。类型按需导入声明在这种情况下被忽略。

在同一个编译单元中的两个或多个类型按需导入声明可以命名相同的包、类或接口。这些声明中只有一个被认为是多余的;效果就好像该类型只导入了一次。

如果一个编译单元同时包含命名同一个类或接口的类型按需导入声明和静态按需导入声明(§7.5.4)，其效果就好像该类或接口的静态成员类和接口(§8.5、§9.5)只导入一次。

例子 7. 5. 2-1. 按需导入类型

```
import java.util.*;
```

导致 java.util 包中声明的所有公共类和接口的简单名称在编译单元的类和接口声明中可用。因此，简单名称 Vector 指的是 java.util 包的 Vector 类，在编译单元中，类声明未被遮掩（§6.4.1）或遮蔽（§7.4.2）的所有位置。

该声明可能被简单名称为 Vector 的类或接口的单一类型导入声明所遮掩；通过名为 Vector 的类或接口，并在编译单元所属的包中声明；或任何嵌套的类或接口。

声明可能会被名为 Vector 的字段、参数或局部变量的声明所遮蔽。

(出现上述任何一种情况都是不寻常的。)

7.5.3 单个静态导入声明

单个静态导入声明从类或接口导入具有给定简单名称的所有可访问静态成员。这使得这些静态成员在出现单个静态导入声明的编译单元的模块、类和接口声明中以其简单名称可用。

SingleStaticImportDeclaration:
`import static` *TypeName* `.` *Identifier* `;`

TypeName 必须是类或接口的规范名(§6.7)。

类或接口必须是命名包的成员，或者是其最外层词法封闭类或接口声明（§8.1.3）是命名包成员的类或接口的成员，否则会发生编译时错误。

如果命名类或接口不可访问，则为编译时错误 (§6.6)。

标识符必须至少命名命名类或接口的一个静态成员。如果没有该名称的静态成员，或者所有命名成员都不可访问，则这是编译时错误。

允许一个单一静态导入声明导入多个同名字段、类或接口，或导入多个具有相同名称和签名的方法。当命名类或接口从自己的超类型继承多个字段、成员类、成员接口或方法时，就会发生这种情况。

如果同一编译单元中的两个单一静态导入声明试图导入具有相同简单名称的类或接口，则会发生编译时错误，除非两个类或接口相同，在这种情况下，重复的声明将被忽略。

如果单一静态导入声明导入简单名称为 x 的类或接口，并且编译单元还声明简单名称为 x 的顶级类或接口（§7.6），则会发生编译时错误。

如果编译单元既包含导入简单名称为 x 的类或接口的单一静态导入声明，也包含导入简单名字为 x 的类或接口的单一类型导入声明（§7.5.1），则会发生编译时错误，除非两个类或接口相同，在这种情况下，将忽略重复的声明。

7.5.4 按需静态导入声明

静态按需导入声明允许根据需要导入命名类或接口的所有可访问静态成员。

StaticImportOnDemandDeclaration:
`import static` *TypeName* `. * ;`

TypeName 必须是类或接口的规范名称（§6.7）。

类或接口必须是命名包的成员，或者是其最外层词法封闭类或接口声明（§8.1.3）是命名包成员的类或接口的成员，否则会发生编译时错误。

如果命名类或接口不可访问，则为编译时错误 (§6.6)。

同一编译单元中的两个或多个静态按需导入声明可以命名相同的类或接口；其效果就好像只有一个这样的声明。

同一编译单元中的两个或多个静态按需导入声明可以命名同一成员；其效果就好像该成员只导入了一次。

允许一个静态按需导入声明导入多个同名字段、类或接口，或导入多个具有相同名称和签名的方法。当命名类或接口从其自己的超类型继承多个具有相同名字的字段、成员类、成员接口或方法时，就会发生这种情况。

如果编译单元同时包含命名同一类或接口的静态按需导入声明和类型按需导入宣言（§7.5.2），其效果就像该类或接口（§8.5，§9.5）的静态成员类和接口只导入一次一样。

7.6 顶级类和接口声明

顶级类或接口声明声明顶级类（§8.1）或顶级接口（§9.1）。

TopLevelClassOrInterfaceDeclaration:
ClassDeclaration
InterfaceDeclaration
`;`

出现在编译单元中的类和接口声明级别的额外的“；”标记对编译单元的含义没有影响。Java 编程语言中允许使用分号，这仅仅是 C++程序员的一种让步，他们习惯于将“；”放在类声明之后。它们不应该在新的 Java 代码中使用。

在没有访问修饰符的情况下，顶级类或接口具有包访问权限：它只能在声明它的包的普通

编译单元内访问（§6.6.1）。可以将类或接口声明为公共的，以允许从同一模块的其他包中的代码访问该类或接口，并且可能从其他模块的包中的代码访问该类和接口。

如果顶级类或接口声明包含以下任何一个访问修饰符：protected、private 或 static，则这是编译时错误。

如果顶级类或接口的名称显示为同一包中声明的任何其他顶级类或接口的名称，则这是编译时错误。

顶级类或接口的作用域和遮掩在§6.3 和§6.4 中有规定。

顶级类或接口的完全限定名称在§6.7 中指定。

例子 7.6-1. 冲突的顶级类和接口声明

```
package test;
import java.util.Vector;
class Point {
   int x, y;
}
interface Point {        // compile-time error #1
    int getR();
    int getTheta();
}
class Vector { Point[] pts; }        // compile-time error #2
```

这里，第一个编译时错误是由于将名称 Point 重复声明为同一个包中的类和接口引起的。第二个编译时错误是试图通过类声明和单一类型导入声明声明名称 Vector。

但是，请注意，类声明中的名称与可能由同一编译单元中的类型按需导入声明（§7.5.2）导入的类或接口重叠不是错误。因此，在该程序中：

```
package test;
import java.util.*;
class Vector {}        // not a compile-time error
```

类 Vector 的声明是被允许的，即使已经有一个类 java.util.Vector。在这个编译单元里，简单名 Vector 指向类 test.Vector, 而不是 java.util.Vector (它仍然可以由编译单元内的代码引用，但只能由其完全限定名引用)。

例子 7. 6-2. 顶级类和接口的作用域

```
package points;
class Point {
     int x, y;           //coordinates
     PointColor color;   //color of this point
     Point next;         //next point with this color
     static int nPoints;
}

class PointColor {
     Point first;  // first point with this color
     PointColor(int color) { this.color = color; }
     private int color; // color components
}
```

这个程序定义了两个类，它们在类成员的声明中相互使用。由于类 Point 和 PointColor 将 points 包中的所有类声明（包括当前编译单元中的所有声明）作为其作用域，因此该程序可以正确编译。也就是说，前向引用不是问题。

例子 7.6-3. 完全限定名

```
class Point { int x, y; }
```

在这个代码中，类 Point 声明在没有包声明的编译单元里，因此 Point 是它的完全限定名，而在代码中：

```
package vista;
class Point { int x, y; }
```

类 Point 的完全限定名是 vista.Point。(包名称 vista 适用于本地或个人使用；如果该包打算广泛分发，最好给它一个唯一的包名（§6.1））。

Java SE 平台的实现必须通过封装模块名和二进制名的组合来跟踪包中的类和接口（§13.1）。必须将类或接口的多种命名方式扩展为二进制名称，以确保此类名称被理解为引用同一类或接口。

例如，如果编译单元包含单一类型导入声明 (§7.5.1):

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

那么在那个编译单元里，简单名 Vector 和完全限定名 java.util.Vector 指向同一个类。

当且仅当包存储在文件系统中（§7.2），如果在由类或接口名称加扩展名（如.java 或.jav）组成的文件中未找到类或接口，则主机系统可以选择强制执行这是编译时错误的限制，前提是以下任一项为真：

- 类或接口由声明类或接口的包的其他普通编译单元中的代码引用。
- 类或接口被声明为公共的(因此可以从其他包中的代码访问)。

这个限制意味着每个编译单元最多只能有一个这样的类或接口。这个限制使得 Java 编译器很容易在包中找到指定的类或接口。在实践中，许多程序员选择将每个类或接口放在自己的编译单元中，不管它是否是公共的，还是被其他编译单元中的代码引用。

例如,公共类 wet.sprocket.Toad 的源代码可以在目录 wet/sprocket 下的 Toad.java 文件找到，相应的目标代码可以在同一目录下的 Toad.class 文件找到。

7.7 模块声明

模块声明指定一个新的命名模块。命名模块指定了对其他模块的依赖，以定义其自身代码可用的类和接口的作用域;并指定导出或开放它的哪些包，为了填充其他模块可用的类和接口的作用域，这些模块指定了对它的依赖。

“依赖”是由 require 指令表达的，与该指令指定的名称是否存在模块无关。“依赖”是由给定 requires 指令的解析（如 java.lang.module 包规范中所述）枚举的可观察模块。通常，Java 编程语言的规则对依赖性比依赖关系（后者指不正常或不必要的依赖，译者注）更感兴趣。

ModuleDeclaration:
{Annotation} [`open`] `module` *Identifier* { `.` *Identifier*}
`{` *{ModuleDirective}* `}`

模块声明引入了一个模块名称，可以在其他模块声明中使用该名称来表示模块之间的关系。模块名称由一个或多个 Java 标识符（§3.8）组成，由“.”标记分隔。

有两种模块：普通模块和开放模块。模块的类型决定了对模块外部代码访问模块类型及这些类型成员的性质。

没有 open 修饰符的普通模块在编译时和运行时仅授予显式导出的包中的类型的访问权。

带有 open 修饰符的开放模块在编译时仅授予对显式导出的包中的类型的访问权，但在运行时授予对其所有包中的类型的访问权，就像所有包都已导出一样。

对于模块外部的代码(无论该模块是普通的还是开放的)，在编译时或运行时授予模块导出包中类型的访问权是专门针对这些包中的公共类型和受保护类型，以及这些类型的公共和受保护成员(§6.6)。编译时或运行时不授予对未导出包中的类型或其成员的访问权。模块内部的代码可以在编译时和运行时访问模块中所有包中的公共和受保护类型，以及这些类型的公共和受保护成员。

不同于编译时的访问和运行时的访问，Java SE 平台通过核心反射 API(§1.4)提供了反射访问。普通模块仅授予对显式导出或显式开放(或同时满足)的包中的类型的反射访问权。开放模块授予对其所有包中的类型的反射访问权，就像所有包都是开放的一样。

对于普通模块之外的代码，授予模块导出(未开放)包中的类型的反射访问是专门针对这些包中的公共和受保护类型，以及这些类型的公共和受保护成员的。授予模块开放的包中的类型的反射访问权限(无论是否导出)是对这些包中的所有类型，以及这些类型的所有成员。对于未导出或开放的包中的类型或其成员，不授予反射访问权。模块内的代码可以反射访问模块中所有包中的所有类型及其成员。

对于开放模块之外的代码，授予模块开放包中的类型(即模块中的所有包)的反射访问权是授予这些包中的所有类型，以及这些类型的所有成员。模块内的代码可以反射访问模块中所有包中的所有类型及其成员。

模块声明的指令指定了该模块对其他模块的依赖(通过 require，§7.7.1)，其他模块可用的包(通过 exports 和 open，§7.7.2)，它消费的服务(通过 uses，§7.7.3)，以及它提供的服务(通过 provide，§7.7.4)。

ModuleDirective:
`requires` *{RequiresModifier} ModuleName* `;`
`exports` *PackageName* [`to` *ModuleName {, ModuleName}*] `;`
`opens` *PackageName* [`to` *ModuleName {, ModuleName}*] `;`
`uses` *TypeName* `;`
`provides` *TypeName* `with` *TypeName {, TypeName}* `;`

RequiresModifier:
(one of)
`transitive static`

当且仅当包存储在文件系统中(§7.2)，主机系统可以选择强制执行这样的限制:如果模块声明没有在由 module-info 和扩展名(如.java 或.java)组成的文件中找到，则为编译时错误。

为了便于理解，通常(虽然不是必需的)，模块声明将其指令进行分组，这样，与模块相关的 require 指令和与包相关的 exports 和 open 指令，以及与服务相关的 uses 和 provide 指令，在视觉上都是不同的。例如:

```
module com.example.foo {
    requires com.example.foo.http;
    requires java.logging;

    requires transitive com.example.foo.network;

    exports com.example.foo.bar;
    exports com.example.foo.internal to com.example.foo.probe;

    opens com.example.foo.quux;
    opens com.example.foo.internal to com.example.foo.network,
                                      com.example.foo.probe;

    uses com.example.foo.spi.Intf;
    provides com.example.foo.spi.Intf with com.example.foo.Impl;
}
```

如果模块是开放的，则可以避免使用 open 指令:

```
open module com.example.foo {
    requires com.example.foo.http;
    requires java.logging;

    requires transitive com.example.foo.network;

    exports com.example.foo.bar;
    exports com.example.foo.internal to com.example.foo.probe;

    uses com.example.foo.spi.Intf;
    provides com.example.foo.spi.Intf with com.example.foo.Impl; }
```

鼓励 Java 编程语言的开发工具突出显示 requires transitive 指令和不受限制的的 exports 指令，因为它们构成了模块的主要 API。

7.7.1 依赖

require 指令指定当前模块依赖的模块的名称。

require 指令不能出现在 java.base 模块的声明中，否则发生编译时错误，因为它是原始模块，没有依赖(§8.1.4)。

如果模块的声明不表示对 java.base 的依赖,并且该模块本身不是 java.base，则该模块隐式

声明了对 java.base 模块的依赖。

requires 关键字后面可能跟有 transitive 修饰符。这会导致任何需要当前模块的模块隐式声明依赖于 requires transitive 指令指定的模块。

requires 关键字后面可能跟有 static 修饰符。这指定依赖性在编译时是强制性的，但在运行时是可选的。

如果模块的声明表示对 java.base 模块的依赖性，并且该模块本身不是 java.base,如果在 requires 关键字之后出现修饰符，则为编译时错误。

如果模块声明中有多个 requires 指令指定了相同的模块名称，则这是编译时错误。

如果由于 java.lang.module 包规范中描述的任何原因，当前模块作为唯一的根模块解析失败，则这是一个编译时错误。

> 例如，如果 requires 指令指定了一个不可观察的模块，或者当前模块直接或间接地表示了对自身的依赖。

如果解析成功，则其结果指定当前模块读取的模块。当前模块读取的模块确定当前模块可见的普通编译单元（§7.3）。当前模块中的代码可以访问在这些普通编译单元（并且只有那些普通编译单元）中声明的类型（§6.6）。

> JavaSE 平台区分显式声明的命名模块（即使用模块声明）和隐式声明的命名模块（即自动模块）。然而，Java 编程语言并没有表现出这种区别：requires 指令引用命名模块，不用考虑它们是显示声明的还是隐式声明的。
>
> 虽然自动模块便于迁移，但它们不可靠，因为当作者将其转换为显式声明的模块时，它们的名称和导出的包可能会更改。如果 requires 指令引用自动模块，则鼓励 Java 编译器发出警告。如果指令中出现 transitive 修饰符，建议发出特别强烈的警告。
>
> **例子 7.1.1-1. 解析 `requires transitive` 指令**
>
> 假设有四个模块声明如下：

```
module m.A {
   requires m.B;
}

module m.B {
   requires transitive m.C;
}

module m.C {
   requires transitive m.D;
}

module m.D {
   exports p;
}
```

m.D 导出的包 p 声明如下:

```
package p;
public class Point {}
```

其中 m.A 模块中的包客户端引用了导出包 p 中的公共类型 Point:

```
package client;
import p.Point;
public class Test {
                public static void main(String[] args) {
                    System.out.println(new Point());
    }
}
```

这些模块可以这样编译，假设当前目录中每个模块有一个子目录，以它包含的模块命名:

```
javac --module-source-path . -d . --module m.D
javac --module-source-path . -d . --module m.C
javac --module-source-path . -d . --module m.B
javac --module-source-path . -d . --module m.A
```

程序 client.Test 运行如下:

```
java --module-path . --module m.A/client.Test
```


从 m.A 中的代码引用到 m.D 中导出的公共类型 Point 是合法的，因为 m.A 读取 m.D，而 m.D 导出包含 Point 的包。解析确定 m.A 读取 m.D 如下:

- m.A requires m.B 因此读取 m.B。
- 因为 m.A 读取 m.B, 并且因为 m.B requires transitive m.C, 解析确定 m.A 读取 m.C。
- 然后，因为 m.A 读取 m.C, 并且因为 m.C requires transitive m.D, 解析确定 m.A 读取 m.D。

实际上，一个模块可以通过多个依赖级别读取另一个模块，以支持任意数量的重构。一旦模块发布供他人重用（通过 requires），该模块的作者已承诺使用其名称和 API，但可以自由地将其内容重构为其他模块，原始模块将重用这些模块（通过 request transitive），以造福消费者。在上面的例子中，包 p 可能最初是由 m.B 导出的 (因此, m.A requires m.B)，但是重构导致了一些 m.B 的内容移动到了 m.C 和 m.D。通过使用 requires transitive 指令链，m.B, m.C 和 m.D 家族可以保留 m.A 中代码对包 p 的访问权限，而无需强制更改 m.A 的 require 指令。注意 m.D 中的包 p 不会被 m.C 和 m.B“重新导出”;相反，m.A 是用来直接读 m.D 的。

7.7.2 导出和开放的包

exports 指令指定当前模块要导出的包的名称。对于其他模块中的代码，这允许在编译时和运行时访问包中的公共类型和受保护类型，以及这些类型的公共成员和受保护成员（§6.6）。它还为其他模块中的代码授予对这些类型和成员的反射访问权。

opens 指令指定当前模块要开放的包的名称。对于其他模块中的代码，这将在运行时而不是编译时授予对包中的公共类型和受保护类型以及这些类型的公共成员和受保护成员的访问权。它还为其他模块中的代码授予对包中的所有类型及其所有成员的反射访问权。

如果 exports 指定的包没有通过与当前模块相关联的编译单元声明，则会出现编译时错误(§7.3)。

允许 opens 来指定一个包，它不是由与当前模块相关联的编译单元声明的。(如果这个包恰好是由一个与另一个模块关联的可观察编译单元声明的，那么 open 指令对另一个模块没有影响。)

如果一个模块声明中有多个 exports 指令指定了相同的包名，则会出现编译时错误。

如果在一个模块声明中有多个 opens 指令指定了相同的包名，则会出现编译时错误。

如果 opens 指令出现在开放模块的声明中，则会出现编译时错误。

如果 exports 或 opens 指令有一个 to 子句，那么该指令是限定的；否则是不限定的。对于限定指令，包中的 public 和 protected 类型，以及它们的 public 和 protected 成员，只能由 to 子句中指定的模块中的代码访问。to 子句中指定的模块被称为当前模块的朋友。对于非限定指令，任何模块中的代码都可以访问这些类型及其成员。

允许 exports 或 opens 指令的 to 子句指定不可观察的模块（§7.7.6）。

如果给定 exports 指令的 to 子句多次指定相同的模块名称，则这是一个编译时错误。

如果给定 opens 指令的 to 子句多次指定相同的模块名称，则这是一个编译时错误。

7.7.3 服务消费

uses 指令指定了一个服务，当前模块中的代码可以通过 java.util.ServiceLoader 发现该服务的提供者。

如果 uses 指令指定了枚举类，则为编译时错误 (§8.9)。

服务可以在当前模块或另一个模块中声明。如果服务未在当前模块中声明，则该服务必须可由当前模块中的代码访问（§6.6），否则会发生编译时错误。

如果模块声明中有多个 uses 指令指定了相同的服务，则这是一个编译时错误。

7.7.4 服务提供

provides 指令指定了一个服务，with 子句为该服务指定了 java.util.ServiceLoader 的一个或多个服务提供者。

如果 provides 指令将枚举类（§8.9）指定为服务，则为编译时错误。

服务可以在当前模块或另一个模块中声明。如果服务未在当前模块中声明，则该服务必须可由当前模块中的代码访问（§6.6），否则会发生编译时错误。

每个服务提供者都必须是顶级或静态的公共类或接口，否则会发生编译时错误。

必须在当前模块中声明每个服务提供者，否则会发生编译时错误。

如果服务提供者显式声明了一个没有形式参数的公共构造函数，或隐式声明了公共默认构

造函数（§8.8.9），则该构造函数称为提供者构造函数。

如果服务提供者显式声明了一个称为 provider 的公共静态方法，并且没有形式参数，那么该方法称为 provider 方法。

如果服务提供者具有 provider 方法，则其返回类型必须为 (i) 可以在当前模块中声明，也可以在另一个模块中声明并可访问当前模块中的代码; 并且 (ii) 是在 provides 指令中指定的服务的子类型; 否则发生编译时错误。

> 虽然必须在当前模块中声明由 provides 指令指定的服务提供程序，但其 provider 方法可能具有在另一个模块中声明的返回类型。另外，请注意，当服务提供者声明 provider 方法时，服务提供者本身不必是服务的子类型。

如果服务提供程序没有 provider 方法，则该服务提供程序必须具有 provider 构造函数，并且必须是 provides 指令中指定的服务的子类型，否则会发生编译时错误。

如果模块声明中有多个 provides 指令指定了相同的服务，则这是一个编译时错误。

如果给定的 provides 指令的 with 子句多次指定同一个服务提供者，则这是一个编译时错误。

7.7.5 未命名模块

一个可观察的普通编译单元，如果宿主系统不关联一个命名模块(§7.3)，它就会关联一个未命名模块。

Java SE 平台承认 Java SE 9 之前开发的程序不能声明命名的模块，因此提供了未命名的模块。此外，Java SE 平台提供未命名包(§7.4.2)的原因很大程度上适用于未命名模块。

Java SE 平台的实现必须支持至少一个未命名的模块。一个实现可以支持多个未命名的模块，但不是必须这样做。哪个普通编译单元和每个未命名模块关联是由主机系统决定的。

主机系统可以将命名包中的普通编译单元与未命名模块关联起来。

未命名模块的规则旨在最大限度地与已命名模块进行互操作，如下所示:

- 一个未命名的模块读取每个可观察的模块 (§7.7.6)。

  > 由于与未命名模块相关联的普通编译单元是可观察的，因此关联的未命名模块也是可观察的。因此，如果 Java SE 平台的实现支持多个未命名的模块，那么每个未命名的模块都是可观察的;每个未命名的模块读取每个未命名的模块，包括自身。
  >
  > 然而，重要的是要意识到，未命名模块的普通编译单元对已命名模块永远不可见(§7.3)，因为没有任何 require 指令可以安排已命名模块读取未命名模块。Java SE 平台的核心反射 API 可以用来安排一个命名模块在运行时读取一个未命名模块。

- 未命名模块导出其普通编译单元与该未命名模块相关联的每个包。
- 未命名模块打开其普通编译单元与该未命名模块相关联的每个包。

7.7.6 模块的可观察性

如果一个模块是可观察的，以下至少有一个为真:

- 包含模块声明的模块编译单元是可观察的 (§7.3)。
- 与该模块关联的普通编译单元是可观察的。