第六章

名字

名字用于引用程序中声明的实体。

声明实体(§6.1)是一个包、一个类、一个接口、引用类型的成员(类、接口、字段或方法)、一个类型参数、一个形式参数、一个异常参数或一个局部变量。

程序中的名称要么是由单个标识符组成的简单名称，要么是由“.”标记分隔的一系列标识符组成的限定名称(§6.2)。

每个引入名称的声明都有一个作用域(§6.3)，这是程序文本的一部分，在这一部分中，声明的实体可以通过一个简单的名称来引用。

限定名 N.x 可用于引用包或引用类型的成员，其中 N 是简单或限定名，x 是标识符。如果 N 为一个包命名，那么 x 是该包的成员，它是一个类、一个接口或一个子包。如果 N 指定引用类型或引用类型的变量，那么 x 指定该类型的成员，该成员可以是类、接口、字段或方法。

在确定一个名称的含义时(§6.5)，出现的上下文被用来消除具有相同名称的包、类型、变量和方法之间的歧义。

访问控制(§6.6)可以在类、接口、方法或字段声明中指定，以控制何时允许访问成员。访问是与作用域不同的概念。访问指定程序文本的一部分，声明的实体可以通过限定名引用其中的部分。 对声明实体的访问也与字段访问表达式（§15.11）、方法调用表达式（其中方法未由简单名称指定）（§15.12）、方法引用表达式（§14.13）或限定类实例创建表达式（§15.9）相关。在没有访问修饰符的情况下，大多数声明具有包访问，允许访问包中任何包含其声明的地方;其他的选项有 public, protected, 和 private。

完全限定名和规范名(§6.7)也在本章中讨论。

6.1 声明

一个声明将一个实体引入到一个程序中，并包含一个标识符(§3.8)，该标识符可以用于一个名称中来引用这个实体。当引入的实体是类、接口或类型参数时，对标识符进行约束以避免某些上下文关键字。

声明的实体是以下实体之一:

- 一个模块, 在 module 声明中 (§7.7)
- 一个包, 在 package 声明中 (§7.4)
- 一个导入类或接口，在单个类型导入声明或类型按需导入声明中声明 (§7.5.1, §7.5.2)
- 一个导入的 static 成员，在单个静态导入声明或静态按需导入声明中声明 (§7.5.3, §7.5.4)
- 由普通类声明（§8.1）、枚举声明（§9）或记录声明（§10）声明的类
- 接口，由普通接口声明（§9.1）或注解接口声明（§9.6）声明。
- 类型参数，作为泛型类、接口、方法或构造函数声明的一部分(§8.1.2, §9.1.2, §8.4.4, §8.8.4)
- 引用类型的成员 (§8.2, §9.2, §8.9.3, §9.6, §10.7), 以下之一:
  - 类成员 (§8.5, §9.5)
  - 接口成员 (§8.5, §9.5)
  - 字段，以下之一:
    > 类中声明的字段 (§8.3)
    > 接口中声明的字段 (§9.3)

  > 与枚举常量或记录组件相对应的类的隐式声明字段

  > 字段长度，为每个数组类型的隐式成员 (§10.7)

  - 方法，以下之一:
    > 在类中声明的方法（抽象或其他） (§8.4)
    > 在接口中声明的方法（抽象或其他） (§9.4)

  > 对应于记录组件的隐式声明的访问器方法

- 枚举常量 (§8.9.1)
- 记录组件 (§8.10.3)
- 形式参数，以下之一:
  - 类或接口的方法的形式参数 (§8.4.1)
  - 类的构造函数的形式参数 (§8.8.1)
  - lambda 表达式的形式参数 (§15.27.1)
- 在 try 语句的 catch 子句中声明的异常处理程序的异常参数(§14.20)
- 局部变量, 以下之一:
  - 由块中的局部变量声明语句声明的局部变量 (§14.4.2)
  - 由 for 语句或 try-with-resources 语句声明的局部变量 (§14.14, §14.20.3)
  - 由模式声明的局部变量 (§14.30.1)
- 局部类或接口 (§14.3), 以下之一:
  - 由普通类声明声明的局部类
  - 由枚举声明声明的局部类
  - 由记录声明声明的局部类
  - 由普通接口声明声明的局部接口

构造函数(§8.8 和§8.10.4)也是通过声明引入的，但使用声明构造函数的类名，而不是引入新名称。

> 泛型类或接口的声明，如 (class *C*<*T*> ... 或 interface *C*<*T*> ...)， 引入了一个名为 C 的类和一组类型:原始类型 C，参数化类型 C<Foo>，参数化类型 C<Bar>，等等。
>
> 当在泛型不重要的情况下发生对 C 的引用时，下面将其标识为非泛型上下文之一，对 C 的引用表示类或接口 C。在其他上下文中，对 C 的引用表示由 C 引入的类型或类型的一部分。
>
> 15 种非泛型上下文如下:
>
> 1. 在模块声明中的 uses 或 provides 指令中。
> 2. 在单类型导入声明中(§7.5.1)
> 3. 在单静态导入声明.的左边 (§7.5.3)
> 4. 在按需静态导入声明.的左边(§7.5.4)
> 5. 在 sealed 类或接口声明的 permits 子句中(§8.1.6, §9.1.4).

6. 在构造函数声明(的左边(§8.8)
7. 在注解的@符号后面(§9.7)
8. 在类字面量的.class 左侧(§15.8.2)
9. 在限定的 this 表达式中.this 的左侧(§15.8.4)
10. 在一个限定超类字段访问表达式的.super 左侧 (§15.11.2)
11. 在限定方法调用表达式中.Identifier 或.super.Identifier 的左侧(§15.12)
12. 在方法引用表达式中.super::的左侧(§15.13)
13. 后缀表达式中的限定表达式名或 try-with-resources 语句中(§15.14.1, §14.20.3)
14. 方法或构造函数的 throws 子句(§8.4.6, §8.8.5, §9.4)
15. 在异常参数声明中(§14.20)

前 12 个非泛型上下文对应§6.5.1 中 TypeName 的前 12 个句法上下文。第 13 个非泛型上下文是限定的 ExpressionName(如 C.x)可以包含 TypeName C 来表示静态成员访问。在这十三个上下文中 TypeName 的常见使用意义重大: 它表明这些上下文涉及类型的非第一级使用。 相反，第十四和第十五个非泛型上下文使用 ClassType，这表明 throws 和 catch 子句以一级方式使用类型，例如与字段声明一致。将这两个上下文描述为非泛型是因为异常类型不能参数化 (§8.1.2).

注意 ClassType 产品允许注释, 因此可以在 throws 或 catch 子句中注释类型的使用, 而 TypeName 产品不允许注释, 所以不可能在单类型导入声明中注释类型名称。

命名规范

只要有可能，Java SE 平台的类库尝试使用按照以下约定选择的名称。 这些约定有助于使代码更具可读性，并避免某些类型的名称冲突。

我们建议在所有用 Java 编程语言编写的程序中使用这些约定。然而，如果长期以来的传统用法另有规定，就不应该盲目地遵循这些惯例。因此，例如，java.lang.Math 类的 sin 和 cos 方法有数学上传统的名称，尽管这些方法名称无视这里建议的惯例，因为它们很短而且不是动词。

包名和模块名

程序员应该采取措施，通过为广泛分发的包选择唯一的包名来避免两个发布的包具有相同名称的可能性。这使得包可以很容易地自动安装和编目。本节指定了生成这种惟一包名的建议约定。鼓励 Java SE 平台的实现提供自动支持，将一组包从局部和临时包名转换为这里描述的惟一名称格式。

如果不使用惟一的包名，那么包名冲突可能会在任何一个冲突包的创建点很远的地方出现。 这可能会造成用户或程序员难以或不可能解决的情况。ClassLoader 和 ModuleLayer 类可以用来在包交互受限的情况下隔离具有相同名称的包，但不是以一种对简单的程序透明的方式。

您可以通过首先拥有一个 Internet 域名(或属于拥有 Internet 域名的组织)来形成一个惟一的包名，例如 oracle.com。然后，您可以逐个组件反转该名称，在本例中获得 com.oracle。并将此作为包名的前缀，使用组织内部开发的约定进一步管理包名。这样的约定可以指定某些包名组件是部门、

部门、项目、机器或登录名。

例子 6.1-1. 唯一包名

```
com.nighthacks.scrabble.dictionary
org.openjdk.compiler.source.tree
net.jcip.annotations
edu.cmu.cs.bovik.cheese
gov.whitehouse.socks.mousefinder
```

唯一包名称的第一个组件始终使用所有字母用小写 ASCII 书写，并且应为顶级域名之一，如 com、edu、gov、mil、net 或 org，或 ISO 标准 3166 中规定的标识国家的英文双字母代码之一。

在某些情况下，Internet 域名可能不是有效的包名。以下是处理这些情况的一些建议公约:

- 如果域名包含连字符或任何标识符中不允许的其他特殊字符（§3.8），将其转换为下划线。
- 如果产生的包名组件中有任何一个是关键字(§3.9)，在它们后面加一个下划线。
- 如果产生的任何包名组件以数字或任何其他不允许作为标识符初始字符的字符开头，则在组件前添加下划线。

模块的名称应该与它导出的主包的名称相对应。如果一个模块没有这样的包，或者由于遗留原因，它必须有一个与它导出的包不对应的名称，那么它的名称仍然应该以其作者关联的 Internet 域名的相反形式开始。

例子 6.1-2. 唯一模块名

```
com.nighthacks.scrabble
org.openjdk.compiler
net.jcip.annotations
```

包或模块的第一个组件名称不能是 java 标识符。以标识符 java 开头的包和模块名称是为 JavaSE 平台的包和组件保留的。

包或模块的名称并不意味着包或模块存储在互联网上的位置。例如，名为 edu.cmu.cs.bovik.cheese 的包不一定能从主机 cmu.edu 或 cs.cmu.edu 或 bovik.cs.cmu.edu 获取。生成唯一包和模块名称的建议约定仅仅是一种将包和模块命名约定附加到现有的、广为人知的惟一名称注册表上的方法，而不必为包和模块名称创建单独的注册表。

类和接口名

类名应该是描述性名词或名词短语，不要太长，混合大小写，每个单词的第一个字母大写。

例子 6.1-3. 描述性的类名

```
ClassLoader
SecurityManager
Thread
Dictionary
```

```
BufferedInputStream
```

同样地，接口的名称应该是短的和描述性的，不太长，混合大小写，每个单词的第一个字母大写。该名称可以是一个描述性名词或名词短语，当接口被当作抽象超类使用时，这是合适的，例如接口 java.io.DataInput 和 java.io.DataOutput;也可以是描述行为的形容词，比如 Runnable 接口和 Cloneable 接口。

类型变量名

类型变量的名称应该简洁明了(如果可能的话，使用单个字符)，而且不应该包含小写字母。这使得将类型参数与普通类和接口区分开来变得很容易。

容器类和接口应该使用 E 作为它们的元素类型。映射应该用 K 表示键的类型，用 V 表示值的类型。名称 X 应该用于任意异常类型。我们用 T 表示类型，当没有任何更具体的类型来区分它时。(这在泛型方法中经常发生。)

如果有多个类型参数表示任意类型，则应该使用字母 T 相邻的字母，例如 S。另外，也可以使用数字下标(例如 T1、T2)来区分不同类型的变量。在这种情况下，所有具有相同前缀的变量都应该被下标。

如果泛型方法出现在泛型类中，最好避免对方法和类的类型参数使用相同的名称，以避免混淆。这同样适用于嵌套泛型类。

例子 6.1-4. 常规类型变量名

```
public class HashSet<E> extends AbstractSet<E> { ... }
public class HashMap<K,V> extends AbstractMap<K,V> { ... }
public class ThreadLocal<T> { ... }
public interface Functor<T, X extends Throwable> {
    T eval() throws X;
}
```

当类型参数不能方便地归入上述类别时，应该选择在单个字母范围内尽可能有意义的名称。 不属于指定类别的类型参数不应使用上述名称(E、K、V、X、T)。

方法名

方法名应该是动词或动词短语，在混合情况下，第一个字母小写，任何后续单词的第一个字母大写。下面是一些方法名的附加约定:

- 获取和设置可能被认为是变量 V 的属性的方法应该命名为 getV 和 setV。一个例子是 Thread 类的 getPriority 和 setPriority 方法。
- 返回某个对象长度的方法应该被命名为 length，就像在类 String 中一样。
- 测试一个对象的布尔条件 V 的方法应该命名为 isV。一个例子是类 Thread 的 isInterrupted 方法。
- 将其对象转换为特定格式 F 的方法应该命名为 toF。例如类 Object 的方法 toString 和类

java.util.Date 的方法 toLocaleString 和 toGMTString。

只要可能且适当，将新类中的方法名称基于与之相似的现有类(尤其是来自 Java SE Platform API 的类)中的名称将使其更易于使用。

字段名

非 final 字段的名称应该混合大小写，首字母小写，随后单词的首字母大写。请注意，设计良好的类除了常量字段(静态 final 字段)外，只有很少的公共或受保护字段。

字段的名称应该是名词、名词短语或名词的缩写。

例如，类 java.io.ByteArrayInputStream 的字段 buf、pos 和 count 以及类 java.io.InterruptedIOException 的字段 bytesTransferred。

常量名

接口中常量的名称应该是，并且类的 final 变量通常可以是一个由一个或多个单词、首字母缩略词或缩写组成的序列，都是大写字母，组件之间用下划线"_"字符分隔。常量名称应该是描述性的，而不是不必要的缩写。按照惯例，它们可以是任何适当的词性。

常量名称的示例包括类 Character 的 MIN_VALUE、MAX_VALUE、MIN_RADIX 和 MAX_RADIX。

一组表示集合的替代值的常量，或者不太频繁地表示整型值中的掩码位，有时可以用常用首字母缩略词作为名称前缀来指定。

例如:

```
interface ProcessStates {
    int PS_RUNNING = 0;
    int PS_SUSPENDED = 1;
}
```

局部变量和参数名

局部变量和参数名称应该简短而有意义。它们通常是由小写字母组成的短序列，而不是单词，例如:

- 首字母缩写，这是一系列单词的第一个字母，如 cp 中表示持有 ColoredPoint 引用的变量
- 缩写，如在 buf 中持有一个指向某种缓冲区的指针
- 助记术语，以某种方式组织以帮助记忆和理解，通常使用一组局部变量，其传统名称与广泛使用的类的参数名称相同。例如:
  - in 和 out, 无论何时涉及某种类型的输入和输出，都以 System 字段为参照
  - off 和 len, 当涉及到偏移量和长度时，都以 java.io 的 DataInput 和 DataOutput 接口的 read 和 write 方法为参照

应该避免使用单字符的局部变量或参数名，临时和循环变量或变量包含某个类型的不可区分值的地方除外。传统的单字符名称是:

- b 表示 byte
- c 表示 char
- d 表示 double
- e 表示 Exception
- f 表示 float
- i, j, 和 k 表示 ints
- l 表示 long
- o 表示 Object
- s 表示 String
- v 表示任意类型的值

仅由两到三个小写字母组成的局部变量或参数名称不应与作为唯一包名称的第一个组成部分的初始国家代码和域名冲突。

6.2 名字和标识符

名称用于引用程序中声明的实体。

名称有两种形式:简单名称和限定名称。

简单名称是一个标识符。

限定名称由名称、“.”标记和标识符组成。

在确定一个名称的含义时(§6.5)，要考虑该名称出现的上下文。 §6.5 的规则区分了一个名称必须表示(指向)一个包 (§6.5.3);类、接口或类型参数(§6.5.5);表达式中的变量或值(§6.5.6);或者一个方法(§6.5.7) 的上下文。

包、类、接口和类型参数具有可以通过限定名访问的成员。关于限定名称的讨论和名称含义的确定，请参见§4.4、§4.5.2、§4.8、§4.9、§7.1、§8.2、§9.2 和§10.7 中的资格说明。

并非程序中的所有标识符都是名称的一部分。标识符也用于以下情况:

- 在声明中（§6.1），可能会出现一个标识符来指定被声明实体的名称。
- 作为引用语句标签的标记语句（§14.7）和 break 和 continue 语句（§15，§14.16）中的标签。
- 标记语句及其关联的 break 和 continue 语句中使用的标识符与声明中使用的标识

符完全不同。

• 在字段访问表达式（§15.11）中，标识符出现在“.”标记之后，以指示由“.”符号之前的表达式表示的对象的成员，或由“.”符号之前的 super 或 TypeName.super 表示的对象。

• 在某些方法调用表达式（§15.12）中，当标识符出现在“.”标记之后和“（”标记之前时，表示要为“.”符号之前的表达式表示的对象，或“.”符号之前的 TypeName 表示的类型表示的对象，或“. ”符号之前的 super 或 TypeName.super 表示的对象调用。

• 在某些方法引用表达式中(§15.13)，当一个标识符出现在一个“::”标记之后，以表示由“::”标记前的表达式所表示的对象的方法，或由“::”标记前的TypeName所表示的类型，或由“::”标记前的 super 或 TypeName.super 所表示的对象。

• 在限定的类实例创建表达式(§15.9)中，在新标记的右侧出现一个标识符，表示一个类型，该类型是新标记前面表达式的编译时类型的成员。

• 在元素-值对注解(§9.7.1)中，表示对应注解接口的一个元素。

在这个程序中:

```
class Test {
    public static void main(String[] args) {
        Class c = System.out.getClass();
        System.out.println(c.toString().length() +
             args[0].length() + args.length);
        }
    }
```

标识符 Test, main, 第一次出现的 args 和 c 不是名称。相反，它们是声明中用于指定声明实体名称的标识符。名称 String，Class，System.out.getClass，System.out.println，c.toString，args，和 args.length 出现在例子中。

出现在 args.length 中的 length 是一个名字，因为 args.length 是一个限定名 (§6.5.6.2) 而不是字段访问表达式(§15.11). 字段访问表达式以及方法调用表达式、方法引用表达式和限定类实例创建表达式使用标识符而不是名称来表示相关成员。因此，args[0].length()里的 length 不是名字，而是一个出现在方法调用表达式中的标识符。

有人可能会想，为什么这些类型的表达式使用标识符，而不是简单的名称，后者毕竟只是一个标识符。原因是简单表达式的名称是根据词法环境定义的;也就是说，简单表达式名必须在变量声明的作用域内(§6.5.6.1)。另一方面，字段访问、限定方法调用、方法引用和限定类实例创建都引用了名称不在词法环境中的成员。根据定义，此类名称仅在字段访问表达式、方法调用表达式、方法引用表达式或类实例创建表达式的 Primary 表达式提供的上下文中绑定；或者通过字段访问表达式、方法调用表达式或方法引用表达式的超级表达式；等等 因此，我们用标识符而不是简单的名称来表示这些成员。

更复杂的是，字段访问表达式不是表示对象字段的唯一方法。出于解析原因，限定名称用于表示

作用域内变量的字段。(变量本身用一个简单的名称表示，如上所述。) 访问控制（§6.6）必须适用于字段的两种表示。

6.3 声明作用域

声明的作用域是程序的区域，在该区域内，可以使用简单名称引用声明的实体，前提是该实体未被隐藏（§6.4.1）。

当且仅当声明的作用域包括程序中的某一点时，声明才被称为在该点的作用域内。

可观测顶层包（§7.4.3）的声明作用域是与包唯一可见的模块相关的所有可观测编译单元（§7.4.3）。

不可观察包的声明永远不在作用域内。

子包的声明从不在作用域内。

java 包始终在作用域内。

由单一类型导入声明（§7.5.1）或类型按需导入声明(§7.5.2)导入的类或接口的作用域是模块声明（§8.7）和导入声明出现的编译单元的所有类和接口声明（§9.1），以及编译单元的模块声明或包声明上的任何注解。

通过单个静态导入声明（§7.5.3）或静态按需导入声明(§7.5.4)导入的成员的作用域是模块声明和导入声明出现的编译单元的所有类和接口声明，以及编译单元的模块声明或包声明上的任何注解。

顶级类或接口（§7.6）的作用域是声明顶级类或接口的包中的所有类和接口声明。

在类或接口 C（§8.2，§9.2）中声明或由其继承的成员 m 的作用域是 C 的整个主体，包括任何嵌套的类或接口声明。如果 C 是一个记录类，那么 m 的作用域额外包括 C 的记录声明的头。

方法（§8.4.1）、构造函数（§8.8.1）或 lambda 表达式（§15.27）的形式参数的作用域是方法、构造函数或 lambda 表达式的整个主体。

类的类型参数（§8.1.2）的作用域是类声明的类型参数部分、类声明的任何超类类型或超接口类型的类型参数部分以及类主体。如果该类是记录类（§8.10），则类型参数的作用域还包括记录声明的头（§8.10.1）。

接口类型参数（§9.1.2）的作用域是接口声明的类型参数部分、接口声明的任何超接口类型的类型参数部分以及接口主体。

方法类型参数（§8.4.4）的作用域是方法的整个声明，包括类型参数部分，但不包

括方法修饰符。

构造函数类型参数（§8.8.4）的作用域是构造函数的整个声明，包括类型参数部分，但不包括构造函数修饰符。

直接由块（§14.2）包围的局部类或接口声明的作用域是直接包围块的其余部分，包括局部类或接口声明本身。

由 switch 块语句组(§14.11)包围的局部类或接口声明的作用域是包围的 switch 块语句组的其余部分，包括局部类或接口声明本身。

由局部变量声明语句(§14.4.2)在块中声明的局部变量的作用域是块的其余部分，从声明自己的初始化器开始，并在局部变量声明语句中包括往右的所有声明符。

在基本 for 语句(§14.14.1)的 ForInit 部分声明的局部变量的作用域包括以下所有内容:

- 它自己的初始化器
- 在 for 语句的初始化部分向右的任何进一步声明符
- For 语句的表达式和更新部分
- 包含的语句

增强型 for 语句（§14.14.2）头中声明的局部变量的作用域是包含的语句。

在 try-with-resources 语句（§14.20.3）的资源规范中声明的局部变量的作用域是从资源规范的其余部分和与 try-with-resources 语句相关的整个 try 块的右侧声明开始的。

try-with-resources 语句的翻译暗示了上述规则。

在 try 语句（§14.20）的 catch 子句中声明的异常处理程序的参数作用域是与 catch 关联的整个块。

例子 6.3-1. 类声明作用域

这些规则意味着类和接口类型的声明不必出现在类型的使用之前。在下面的程序中，类 Point 中 PointList 的使用是合法的，因为类声明 PointList 的作用域包括类 Point 和类 PointList，以及 points 包的其他编译单元中的任何其他类或接口声明。

```
package points;
class Point {
     int x, y;
     PointList list;
     Point next;
}
```

```
class PointList {
     Point first;
}
```

例子 6.3-2. 局部变量声明作用域

下面的程序会导致编译时错误，因为局部变量 x 的初始化在局部变量 x 声明的作用域内，但是局部变量 x 还没有值，不能使用。字段 x 的值为 0(在初始化 Test1 时赋值)，但由于它被局部变量 x 掩盖(§6.4.1)，所以它是一个转移注意力的东西。

```
class Test1 {
     static int x;
     public static void main(String[] args) {
        int x = x;
     }
}
```

下面的程序可以编译成功:

```
class Test2 {
   static int x;
   public static void main(String[] args) {
        int x = (x=2)*2;
        System.out.println(x);
    }
}
```


因为局部变量 x 是在使用之前明确赋值的 (§16)，程序可以打印结果。

在下面的程序中，three 的初始值设定项可以正确地引用在较早的声明器中声明的变量 two，下一行中的方法调用可以正确引用在块中较早声明的变量 three。

```
class Test3 {
    public static void main(String[] args) {
         System.out.print("2+1=");
         int two = 2, three = two + 1;
         System.out.println(three);
    }
}
```

程序产生输出:

```
2+1=3
```

模式变量声明的作用域（即，由模式声明的局部变量）是在值与模式匹配成功后可能执行的程序部分（§14.30.2）。它是通过考虑模式变量在从声明模式变量的模式开始的区域中明确匹配的程序点来确定的。

本节的其余部分将专门对“绝对匹配”这个词作一个精确的解释。分析考虑了语句和

表达式的结构，并对布尔表达式操作符和某些语句形式进行了特殊处理。

> 可以看出，模式变量声明的作用域是一个与流相关的概念，类似于明确赋值（§16）。本节其余部分定义的规则有意采用与明确赋值规则类似的形式。

分析依赖于“引入”这个专业术语，其形式如下:

- 模式变量在为真时由表达式引入
- 模式变量在为假时由表达式引入
- 模式变量由语句引入

> 最简单的例子是模式变量 s 由表达式 a instanceof String s（当为 true 时）引入。换句话说，如果表达式的值为 true，则模式匹配必须成功，因此必须为模式变量赋值。
>
> 相反，模式变量 t 由表达式!(b instanceof Integer t)引入（当为 false 时）。这是因为只有当表达式的值为 false 时，模式匹配才能成功。

6.3.1 表达式中模式变量的作用域

只有某些类型的布尔表达式涉及到引入模式变量和确定这些变量在何处绝对匹配。如果表达式不是条件 and 表达式、条件 or 表达式、逻辑非表达式、条件表达式、instanceof 表达式、switch 表达式或括号表达式，则不适用作用域规则。

6.3.1.1 条件与操作符 &&

以下规则适用于条件与表达式 *a* && *b* (§15.23):

- 当 a 为真时引入的模式变量在 b 处绝对匹配。

  如果 a 在为 true 时引入的任何模式变量已经在 b 的作用域内，则会发生编译时错误。

- 如果 a&&b 为 true，引入一个模式变量当且仅当 (i) 当为 true 时，它由 a 引入，或 (ii) 当为 true 时，它由 b 引入。

> 应该注意的是，当 a&b 为 false 时，没有规则引入模式变量。这是因为在编译时无法确定哪个操作数的计算结果为 false。

如果以下任何条件成立，则为编译时错误:

- 模式变量（i）在为真时由 a 引入，（ii）在为真时由 b 引入。
- 模式变量（i）在为假时由 a 引入，（ii）在为假的时由 b 引入。

这两种错误情况排除了运算符的两个操作数声明同名模式变量的可能性。例如，考虑有问题的表达式(a instanceof String s) && (b instanceof String s)。第一种错误情况包括整个表达式求值为 true，其中（如果代码合法）需要初始化模式变量 s 的两个声明，因为左侧操作数和右侧操作数都求值为 true。由于无法区分程序其余部分中称为 s 的两个变量，因此整个表达式被认为是错误的。第二个错误案例涵盖了相反的场景，其中整个表达式的计算结果为 false。

6.3.1.2 条件或操作符 ||

以下规则适用于条件或表达式 *a* || *b* (§15.24):

- 当 a 为 false 时引入的模式变量在 b 处绝对匹配。

  如果 a 在 false 时引入的任何模式变量已经在 b 的作用域内，则会出现编译时错误。

- 当 a || b 为 false 时引入的模式变量当且仅当 (i) 当为 false 时，它由 a 引入，或(ii) 当为 false 时，它由 b 引入。

  应该注意的是，当 a||b 为 true 时，没有规则引入模式变量。这是因为在编译时无法确定哪个操作数的计算结果为真。

如果以下任何条件成立，则为编译时错误:

- 模式变量（i）在为真时由 a 引入，（ii）在为真时由 b 引入。
- 模式变量（i）在为假时由 a 引入，（ii）在为假时由 b 引入。

  这两种错误情况排除了||运算符的两个操作数声明同名模式变量的可能性。例如，考虑有问题的表达式 (a instanceof String s) || (b instanceof String s)。第一种错误情况包括整个表达式求值为 true，其中（如果代码合法）模式变量 s 的一个声明将被初始化，这取决于左侧操作数还是右侧操作数求值为 true。由于在编译时无法确定哪一个操作数的值为真，因此也无法确定将初始化哪一个 s 声明，因此整个表达式被认为是错误的。第二个错误案例涵盖了相反的场景，其中整个表达式的计算结果为 false。

6.3.1.3 逻辑非运算符 !

以下规则适用于逻辑非表达式!*a* (§15.15.6):

- 当!a 为 true 时引入的模式变量当且仅当当 a 为 false 时引入。
- 当!a 为 false 时引入的模式变量当且仅当当 a 为 true 时引入。

6.3.1.4 条件运算符 ? :

以下规则适用于条件表达式 a ? b : c (§15.25):

- 当 a 为 true 时引入的模式变量在 b 处绝对匹配。

如果 a 在为 true 时引入的任何模式变量已经在 b 的作用域内，则会发生编译错误。

- 当 a 为 false 时引入的模式变量在 c 处绝对匹配。

  如果 a 为 false 时引入的任何模式变量已经在 c 的作用域内，则会发生编译错误。

  应该注意的是，当 a?b:c 为 true 或 false 时，没有规则引入模式变量。这是因为在编译时无法确定操作数 a 的值是否为 true。

如果下列任何一个条件成立，则为编译时错误:

- 模式变量（i）在为真时由 a 引入，（ii）在为真时由 c 引入。
- 模式变量（i）在为真时由 a 引入，（ii）在为假时由 c 引入。
- 模式变量（i）在为假时由 a 引入，（ii）在为真时由 b 引入。
- 模式变量（i）在为假时由 a 引入，（ii）在为假时由 b 引入。
- 模式变量（i）在为真时由 b 引入，（ii）在为真时由 c 引入。
- 模式变量（i）在为假时由 b 引入，（ii）在为假时由 c 引入。

这些错误情况类似于&&和||操作符的相似错误情况。它们消除了同一模式变量的多个声明可能跨?:操作符的操作数出现的混淆情况。

6.3.1.5 模式匹配操作符 `instanceof`

以下规则适用于具有模式操作数的 instanceof 表达式, a instanceof p (§15.20.2):

• 当 a instance of p 为 true 时引入一个模式变量当且仅当它由模式 p 声明。确定哪些模式变量由模式声明的规则见§14.30.1。

如果当 instanceof p 为 true 时引入的任何模式变量已经在 instanceof 表达式的作用域内，则会出现编译时错误。

模式变量不允许隐藏另一个局部变量 (§6.4).

应该注意的是，当 a instanceof p 为 false 时，没有规则引入模式变量。

6.3.1.6 `switch` 表达式

以下规则适用于 switch 表达式 (§15.28):

• 一个包含在 switch 标签语句组(§14.11.1)中的语句 S 引入的模式变量与在 switch 标签语句组(如果有的话)中 S 后面的所有语句都是绝对匹配的。

6.3.1.7 括号表达式

以下规则适用于括号表达式(a) (§15.8.5):

- 如果(a)为 true 引入的模式变量当且仅当当 a 为 true 时引入。
- 如果(a)为 flase 引入的模式变量当且仅当当 a 为 flase 时引入。

6.3.2 语句中模式变量的作用域

只有少数几种语句在确定模式变量的作用域方面发挥了重要作用。

如果 if、while、do 或 for 语句包含引入模式变量的表达式，那么在某些情况下，这些变量的作用域可以包括语句的子语句。

例如，在下面的 if-then-else 语句中，模式变量 s 的作用域包括一个子语句，但不包括另一个子语句:

```
Object o = ...
if (o instanceof String s)
    // s in scope for this substatement; no cast of o needed
    System.out.println(s.replace('*', '_'));
else
    // s not in scope for this substatement (hence, error)
    System.out.println(s);
```

此外，在某些情况下，模式变量可以通过语句本身引入，而不是语句中的表达式。由语句引入的模式变量在封闭块的以下语句处于作用域中。

例如，在下面的方法中，模式变量 s 的作用域包括 if 语句后面的方法体:

```
void test(Object o) {
    if (!(o instanceof String s)) {
        throw new IllegalArgumentException();
    }
    // This point is only reachable if the pattern match succeeded
    // Thus, s is in scope for the rest of the block
    ...
    System.out.println(s.repeat(5));
    ...
}
```

6.3.2.1 块

下面的规则适用于包含在非 switch 块(§14.11.1)中的块(§14.2)语句 S:

- 由 S 引入的模式变量与在块中 S 之后的所有块语句(如果有的话)都是绝对匹配的。

6.3.2.2 `if` 语句

以下规则适用于语句 if (e) S (§14.9.1):

- 当 e 为真时引入的模式变量与 S 绝对匹配。

  如果 e 在为 true 时引入的任何模式变量已经在 S 的作用域内，则将发生编译时错误。

- 引入了模式变量 if(e) S 当且仅当(i) 当 e 为 false 时引入 (ii) S 无法正常完成。

  如果任何由 if (e) S 引入的模式变量已经在 if 语句的作用域内，则会出现编译时错误。

  if-then 语句引入模式变量的规则依赖于“不能正常完成”(§14.22)的概念，而“不能正常完成”又依赖于常量表达式(§15.29)的概念。这意味着，计算模式变量的作用域可能需要确定是一个简单的名称，还是 TypeName.Identifier 形式的限定名称，引用常数变量。由于模式变量永远不会引用常数变量，因此不存在循环性。

以下规则适用于一个语句 if (e) S else T (§14.9.2):

- 当 e 为真时引入的模式变量在 S 处绝对匹配。

  如果 e 在 true 时引入的任何模式变量已经在 S 的作用域内，则这是一个编译时错误。

- 当 e 为假时引入的的模式变量在 T 处绝对匹配。

  如果 e 在 false 时引入的任何模式变量已经在 T 的作用域内，则这是一个编译时错误。

- 引入了模式变量 if (*e*) *S* else *T* 当且仅当:

  - 当 e 为真时引入，S 可以正常完成，T 不能正常完成；或
  - 当 e 为假时引入，S 不能正常完成，T 可以正常完成。

  如果 if（e）S else T 引入的任何模式变量已经在 if 语句的作用域中，则这是一个编译时错误。

  这些规则突出了模式变量作用域的类流性质。例如，在以下语句中:

  ```
  if (e instanceof String s) {
     counter += s.length();
  } else {
     System.out.println(e); // s not in scope
  }
  ```

模式变量 s 由 instanceof 表达式引入，在第一个包含的语句（then 块中的赋值语句）的作用域中，但不在第二个包含的语句（else 块中的表达式语句）的作用域中。

此外，结合对布尔表达式的处理，模式变量的作用域对于利用常见的布尔逻辑等价的代码重构是稳健的。例如，前面的代码可以重写为:

```
if (!(e instanceof String s)) {
     System.out.println(e); // s not in scope }
else {
     counter += s.length();
}
```

代码甚至可以重写如下，尽管不推荐重复使用!!操作符:

```
if (!!(e instanceof String s)) {
   counter += s.length();
} else {
   System.out.println(e); // s not in scope
}
```

6.3.2.3 `while` 语句

以下规则应用于语句 while (*e*) *S* (§14.12):

- 当 e 为真时引入的模式变量在 S 绝对匹配。

  如果 e 在为 true 时引入的任何模式变量已经在 S 的作用域内，则将发生编译时错误。

- 由 while (e) S 引入的模式变量当且仅当 (i) 当 e 为 false 时被引入(ii) S 不包含一个可达的 break 语句，其 break 目标包含 S(§14.15)。

  如果 while (e) S 引入的模式变量已经在 while 语句的作用域内，则会出现编译时错误。

6.3.2.4 `do` 语句

以下规则适用于语句 do *S* while (*e*) (§14.13):

- 由 do S while (e)引入的模式变量当且仅当 (i) 当 e 为 false 时被引入(ii) S 不包含一个可达的 break 语句，其 break 目标包含 S(§14.15)。

  如果 do S while (e)引入的模式变量已经在 do 语句的作用域内，则会出现编译时错误。

6.3.2.5 `for` 语句

以下规则适用于一个基本的 for 语句 (§14.14.1):

- 当条件表达式为真时引入的模式变量，在递增部分和所包含的语句中都绝对匹配。

  如果条件表达式在为 true 时引入的任何模式变量已经在所包含语句的递增部分的作用域内，则会出现编译时错误。由一个基本的 for 语句引入的模式变量当且仅当(i)当条件表达式为 false 时被引入并且(ii) 被包含的语句 S 不包含一个 break 目标包含 S 的可达 break 语句(§14.15)。

  如果基本 for 语句引入的任何模式变量已经在 for 语句的作用域内，那么这是编译时错误。

增强的 for 语句(§14.14.2)是通过转换为基本的 for 语句来定义的，因此不需要为它提供特殊的规则。

6.3.2.6 `switch` 语句

以下规则适用于一个 switch 语句 (§14.11):

- 由 switch 块语句组（§14.11.1）中包含的标记语句 S 引入的模式变量与 switch 块语句组中 S 之后的所有语句（如果有的话）都是绝对匹配的。

6.3.2.7 标签语句

以下规则适用于一个标签语句(§14.7):

- 模式变量由带标签的语句引入，当它由带标签语句的直接包含语句引入时。

6.4 遮蔽和遮掩

局部变量(§14.4)、形式参数(§8.4.1、§8.8.1、§15.27.1)、异常参数(§14.20)、局部类或局部接口(§14.3)只能使用简单名称来引用，而不能使用限定名称(§6.2)。

有些声明在局部变量声明、正式形参声明、异常参数声明、局部类声明或局部接口声明的作用域内是不允许的，因为仅使用简单名称无法区分声明的实体。

> 例如，如果方法的形式参数的名称可以被重新声明为方法体中局部变量的名称，那么局部变量将会遮蔽形式参数，并且没有办法引用形式参数——这是一个不希望看到的结果。

如果使用形式参数的名称在方法、构造函数或 lambda 表达式的体中声明一个新变量，除非新变量是在方法、构造函数或 lambda 表达式所包含的类或接口声明中声明的，否则将是编译时错误。

如果使用局部变量 v 的名称在 v 的作用域内声明一个新变量，除非新变量是在 v 作用域内的类或接口声明中声明的，否则将会出现编译时错误。

如果使用异常参数的名称在 catch 子句的块中声明一个新变量，除非新变量是在

catch 子句的块包含的类或接口声明中声明的，否则将是编译时错误。

如果使用局部类或接口 C 的名称在 C 的作用域内声明新的局部类或接口，除非新的局部类或接口是在 C 的作用域内出现的类或接口声明中声明的，否则将出现编译时错误。

这些规则允许在嵌套的类或接口声明中重新声明变量、局部类或局部接口，这些声明发生在变量、局部类或局部接口的作用域内;这种嵌套的类或接口声明可以是局部类或接口声明(§14.3)，也可以是匿名类声明(§15.9.5)。因此，形式参数、局部变量、局部类或局部接口的声明可能被嵌套在方法、构造函数或 lambda 表达式中的类或接口声明遮蔽；并且异常参数的声明可以被嵌套在 catch 子句块内的类或接口声明遮蔽。

有两种设计方案可用于处理由 lambda 参数和 lambda 表达式中声明的其他变量创建的名称冲突。一种是模仿类声明：与局部类一样，lambda 表达式为名称引入了一个新的“级别”，表达式之外的所有变量名称都可以重新声明。另一种是“局部”策略：与 catch 子句一样，对于循环和块，lambda 表达式在与封闭上下文相同的“级别”上运行，表达式外部的局部变量不能被遮蔽。上述规则使用局部策略；不存在允许 lambda 表达式中声明的变量遮蔽在封闭方法中声明的变量的特殊豁免。

例子 6.4-1. 尝试遮蔽局部变量

由于将标识符声明为方法、构造函数或初始化块的局部变量不得出现在同名参数或局部变量的作用域内，因此以下程序会发生编译时错误:

```
class Test1 {
    public static void main(String[] args) {
         int i;
         for (int i = 0; i < 10; i++)
            System.out.println(i);
    }
}
```

这个限制有助于检测一些非常模糊的 bug。对局部变量遮蔽成员的类似限制被认为是不切实际的，因为在超类中添加成员可能会导致子类不得不重命名局部变量。相关的考虑使得限制嵌套类的成员遮蔽局部变量，或者限制嵌套类中声明的局部变量遮蔽局部变量也变得没有吸引力。

因此，以下程序编译正常:

```
class Test2 {
    public static void main(String[] args) {
         int i;
         class Local {
            {
                    for (int i = 0; i < 10; i++)
                       System.out.println(i);
            }
        } new Local();
    }
}
```

另一方面，具有相同名称的局部变量可以在两个单独的块或 for 语句中声明，其中任何一个都不包含另一个:

```
class Test3 {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
        for (int i = 10; i > 0; i--)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

程序编译正常，执行时产生输出:

```
0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1
```

这种风格在模式匹配中也很常见，重复的模式通常使用相同的名称:

```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
}

class Test4 {
  static void test(Object a, Object b, Object c) {
       if (a instanceof Point p) {
          System.out.println("a is a point ("+p.x+","+p.y+")");

       }
       if (b instanceof Point p){
         System.out.println("b is a point ("+p.x+","+p.y+")");
       } else if (c instanceof Point p) {
         System.out.println("c is a point ("+p.x+","+p.y+")");
       }
     }

  public static  void main(String[] args) {
       Point   p = newPoint(2,3);
       Point   q = newPoint(4,5);
       Point   r = newPoint(6,7);
       test(p, q, r);
     }
  }
```

然而，模式变量不允许遮蔽局部变量，包括其他模式变量，因此以下程序出现了两个编译时错误:

```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
}

class Test5 {
  static void test(Object a, Object b, Object c) {
```

```
 if (a instanceof Point p) {
        System.out.println("a is a point ("+p.x+","+p.y+")");

        if (b instanceof Point p) {  // compile-time error
            System.out.println("b is a point ("+p.x+","+p.y+")");
        }
      }
}

public static  void main(String[] args) {
    Point   p = newPoint(2,3);
    Point   q = newPoint(4,5);
    Point   r = newPoint(6,7);
    test(p, q, r);

    if (new Object() instanceof Point q) // compile-time error
        System.out.println("I get your point");
} }
```

6.4.1 遮蔽

有些声明的部分作用域可能会被同名的另一个声明遮蔽，在这种情况下，不能使用简单的名称来引用声明的实体。

遮蔽与隐藏是不同的(§8.3，§8.4.8.2，§8.5，§9.3，§9.5)，后者只适用于那些本来可以被继承但由于子类中的声明而不能被继承的成员。遮蔽与遮掩也是不同的 (§6.4.2)。

一个名为 n 的类型的声明 d 遮蔽了在整个 d 作用域内出现的点上的任何其他名为 n 的类型的声明。

一个名为 n 的字段或形式参数的声明 d 在整个 d 的作用域内遮蔽了在 d 出现点的作用域内的任何其他名为 n 的变量的声明。

局部变量或名为 n 的异常参数的声明 d 在 d 的整个作用域内遮蔽（a）在 d 出现的点的作用域内的任何其他名为 n 的字段的声明，以及（b）在 d 发生的点上处于作用域内但未在声明 d 的最内层类中声明的任何其他称为 n 的变量的声明。

一个名为 n 的方法的声明 d 在 d 出现在整个 d 的作用域内的点处遮蔽了封闭作用域内任何其他名为 n 的方法的声明。

包声明从不遮蔽任何其他声明。

类型按需导入声明不会导致任何其他声明被遮蔽。

静态按需导入声明不会导致任何其他声明被遮蔽。

在包 p 的编译单元 c 中的单个类型导入声明 d，导入名为 n 的类型，在整个 c 中，

遮蔽以下声明:

- 在 p 的另一个编译单元中声明的名为 n 的任何顶级类型
- 由 c 中的类型按需导入声明导入的任何名为 n 的类型
- 由 c 中的静态按需导入声明导入的任何名为 n 的类型

在包 p 的编译单元 c 中导入名为 n 的字段的单个静态导入声明 d 在整个 c 中遮蔽了由 c 中的静态按需导入声明导入的名为 n 任何静态字段的声明。

在包 p 的编译单元 c 中的单个静态导入声明 d 导入名为 n 且签名为 s 的方法，在整个 c 中遮蔽了由 c 中的静态按需导入声明导入的名为 n 并签名为 s 任何静态方法的声明。

在包 p 的编译单元 c 中的单个静态导入声明 d，导入名为 n 的类型，在整个 c 中，遮蔽以下声明:

- 由 c 中的静态按需导入声明导入的任何名为 n 的静态类型;
- 在 p 的另一个编译单元（§7.3）中声明的名为 n 的任何顶级类型（§7.6）；
- 由 c 中的类型按需导入声明（§7.5.2）导入的任何名为 n 的类型。

例子 6.4.1-1. 局部变量声明对字段声明的遮蔽

```
class Test {
   static int x = 1;
   public static void main(String[] args) {
        int x = 0;
        System.out.print("x=" + x);
        System.out.println(", Test.x=" + Test.x);
   }
}
```

程序产生输出:

```
x=0, Test.x=1
```

程序声明了:

- 一个类 Test
- 类 Test 的一个静态成员变量 x
- 类 Test 的一个成员方法 main
- main 方法的参数 args
- main 方法的局部变量 x

由于类变量的作用域包括类的整个主体（§8.2），因此类变量 x 通常可以在 main 方法的整个主体中使用。然而，在本例中，类变量 x 被局部变量 x 的声明遮蔽在 main 方法的主体中。

局部变量的作用域为其声明的块的其余部分（§6.3）；在本例中，这是 main 方法主体的其余部分，即其初始值设定项“0”和 System.out.print 和 System.out.println 方法的调用。

这意味着：

- 调用 print 时的表达式 x 引用（表示）局部变量 x 的值。
- 调用 println 使用限定名称（§6.6）Test.x，它使用类类型名称 Test 来访问类变量 x，因为 Test.x 的声明在这一点上被遮蔽，不能通过其简单名称来引用。

关键字 this 也可以用于访问遮蔽字段 x，使用 this.x 的形式。事实上，这种习惯用法通常出现在构造函数中（§8.8）：

```
class Pair {
    Object first, second;
    public Pair(Object first, Object second){
        this.first = first;
        this.second = second;
    }
}
```

这里，构造函数接受与要初始化的字段名称相同的参数。这比为参数创建不同的名称要简单得多，而且在这种风格化的上下文中也不会太混乱。但是，通常情况下，使用与字段名称相同的局部变量被认为是糟糕的风格。

例子 6.4.1-2. 用另一个类型声明遮蔽一个类型声明

```
import java.util.*;

class Vector {
    int[] val = { 1 , 2 };
}

class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.val[0]);
    }
}
```

程序编译打印输出：

```
1
```

使用这里声明的 Vector 类，而不是可以按需导入的泛型类 java.util.Vector(§8.1.2)。

6.4.2 遮掩

在可能被解释为变量、类型或包的名称的上下文中，可能会出现一个简单的名称。在这种情况下，§6.5.2 的规则规定，变量的选择要优先于类型，类型的选择要优先于包。因此，有时可能不可能通过一个类型或包的简单名称来引用它，即使它的声明在作用域中并且没有被遮蔽。我们说这样的声明是遮掩。

遮掩不同于遮蔽(§6.4.1)和隐藏(§8.3、§8.4.8.2、§8.5、§9.3、§9.5)。

模块名称与变量、类型或包名称之间没有遮掩；因此，模块可能与变量、类型和包共享名称，尽管不一定建议在模块包含的包之后命名模块。

§6.1 中的命名约定有助于减少遮掩，但如果确实发生这种情况，请注意如何避免这种情况。

当包名称出现在表达式中时：

- 如果包名被字段声明遮掩，则通常可以使用导入声明（§7.5）来提供包中声明的类型名。
- 如果包名被参数或局部变量的声明所遮掩，则可以在不影响其他代码的情况下更改参数或局部变量的名称。

包名的第一个组件通常不容易被误认为是类型名，因为类型名通常以单个大写字母开头。 (Java 编程语言实际上并不依赖大小写来确定名称是包名还是类型名。)

涉及类和接口类型名称的遮掩很少。字段、参数和局部变量的名称通常不会遮掩类型名称，因为它们通常以小写字母开头，而类型名称通常以大写字母开头。

方法名称不能遮掩或被其他名称遮掩(§6.5.7)。

涉及字段名的遮掩是罕见的；然而：

- 如果字段名遮掩了包名，则通常可以使用导入声明（§7.5）来提供该包中声明的类型名。
- 如果字段名遮掩了类型名，则可以使用该类型的完全限定名，除非类型名表示局部类或接口（§14.3）。
- 字段名不能遮掩方法名。
- 如果字段名被参数或局部变量的声明所遮蔽，则可以在不影响其他代码的情况下更改参数或局部变量的名称。

涉及常量名称的遮掩很少：

- 常量名称通常没有小写字母，因此它们通常不会遮掩包或类型的名称，也不会遮蔽字段，字段的名称通常至少包含一个小写字母。
- 常量名称不能遮掩方法名称，因为它们在语法上是可区分的。

6.5 确定名称的含义

名称的含义取决于其使用的上下文。 确定名称的含义需要三个步骤：

- 首先，上下文导致名称在语法上分为七类：ModuleName,PackageName,TypeName,ExpressionName,MethodName,PackageOrTypeName 或 AmbiguousName。

  TypeName 和 MethodName 与其他五个类别相比表现力较差，因为它们分别用 TypeIdentifier 和 UnqualifiedMethodIdentifier 表示（§3.8）。

- 其次，最初根据上下文分类为 AmbiguousName 或 PackageOrTypeName 的名称，然后重新分类为 PackageName、TypeName 或 ExpressionName。

- 第三，生成的类别决定了名称含义的最终确定（如果名称没有含义，则为编译时错误）。

ModuleName:
Identifier
ModuleName . *Identifier*

PackageName:
Identifier
PackageName . *Identifier*

TypeName:
TypeIdentifier
PackageOrTypeName . *TypeIdentifier*

PackageOrTypeName:
Identifier
PackageOrTypeName . *Identifier*

ExpressionName:
Identifier
AmbiguousName . *Identifier*

MethodName:
UnqualifiedMethodIdentifier

AmbiguousName:
Identifier
AmbiguousName . *Identifier*

上下文的使用有助于最小化不同类型实体之间的名称冲突。如果遵循§6.1 中描述的命名约定，则此类冲突将非常罕见。然而，随着不同程序员或不同组织开发的类型的发展，冲突可能会无意中出现。例如，类型、方法和字段可能具有相同的名称。始终可以区分具有相同名称的方法和字段，因为使用的上下文总是告诉我们是否要使用方法。

6.5.1 根据上下文对名称进行句法分类

在这些上下文中，名称在语法上被分类为 ModuleName：

- 在模块声明中的 requires 指令中 (§7.7.1)
- 模块声明中 exports 或 opens 指令 to 的右侧（§7.7.2）

在以下上下文中，名称在语法上被分类为 PackageName：

- 模块声明中 exports 或 opens 指令的右侧
- 在一个限定的 PackageName 的"."的左侧

在这些上下文中，名称在语法上被分类为 TypeName：

为了命名类或接口：

1. 在模块声明中的 uses 或 provides 指令 (§7.7.1)
2. 在单一类型导入声明中 (§7.5.1)
3. 在单个静态导入声明中.的左侧 (§7.5.3)
4. 在静态按需导入声明中.的左侧 (§7.5.4)
5. 在密封类或接口声明的 permits 子句中(§8.1.6, §9.1.4)
6. 在构造函数声明中(的左侧 (§8.8)
7. 注解的@符号之后 (§9.7)
8. 在类字面量中.class 的左侧 (§15.8.2)
9. 在限定的 this 表达式.this 的左侧 (§15.8.4)
10. 在限定的超类字段访问表达式.super 的左侧 (§15.11.2)
11. 在限定的方法调用表达式.Identifier 或.super.Identifier 的左侧 (§15.12)
12. 在方法引用表达式.super::的左侧 (§15.13)

•在使用类型的 17 个上下文中，作为构成任何 ReferenceType（包括数组类型中括号左侧的 ReferenceType，或参数化类型中 < 的左侧，或参数型的非通配符类型参数，或参数类型的通配符型参数的 extends 或 super 子句）的标识符或点标识符序列（§4.11）：

1. 类声明的 extends 或 implements 子句 (§8.1.4, §8.1.5)

2. 接口声明的 extends 子句 (§9.1.3)

3. 方法的返回类型 (§8.4.5, §9.4), 包括注解接口的元素类型 (§9.6.1)

4. 方法或构造函数的 throws 子句 (§8.4.6, §8.8.5, §9.4)

5. 在泛型类、接口、方法或构造函数的类型参数声明的 extends 子句中(§8.1.2, §9.1.2, §8.4.4, §8.8.4)

6. 类或接口字段声明的类型 (§8.3, §9.3)

7. 方法、构造函数或 lambda 表达式的形式参数声明中的类型 (§8.4.1, §8.8.1, §9.4, §15.27.1)

8. 方法的接收参数的类型 (§8.4)

9. 语句（§14.4.2、§14.14.1、§14.14.2、§14.20.3）或模式（§14.30.1）中局部变量声明中的类型

10. 异常参数声明中的类型 (§14.20)

11. 记录类的记录组件声明中的类型 (§8.10.1)

12. 在显式类型参数列表中，指向显式构造函数调用语句或类实例创建表达式或方法调用表达式 (§8.8.7.1, §15.9, §15.12)

13. 在非限定类实例创建表达式中，作为要实例化的类类型（§15.9），或作为要实例化匿名类的直接超类或直接超接口 (§15.9.5)

14. 数组创建表达式中的元素类型 (§15.10.1)

15. 强制转换表达式的强制转换运算符中的类型(§15.16)

16. 关系运算符 instanceof 后面的类型(§15.20.2)

17. 在方法引用表达式（§15.13）中，作为搜索成员方法的引用类型，或作为要构造的类类型或数组类型。

在上述 17 个上下文中，从 ReferenceType 的标识符中提取 TypeName 旨在递归地应用于 ReferenceType 中的所有子项，例如其元素类型和任何类型参数。

例如，假设字段声明使用类型 p.q.Foo[]。忽略数组类型的方括号，将术语 p.q.Foo 提取为数组类型中方括号左侧的点标识符序列，并将其分类为 TypeName。后面的步骤确定 p、q 和 Foo 中的哪一个是类型名或包名。

作为另一个示例，假设一个强制类型转换操作符使用类型 p.q.Foo<? extends String>。术语 p.q.Foo 再次被提取为标识符术语的点序列，这次在参数化类型中位于<的左侧，并被分类为 TypeName。术语字符串被提取为参数化类型的通配符类型参数的 extends 子句中的标识符，并

分类为 TypeName。

在以下上下文中，名称在语法上被分类为 ExpressionName：

- 作为限定超类构造函数调用中的限定表达式 (§8.8.7.1)
- 作为限定类实例创建表达式中的限定表达式 (§15.9)
- 作为数组访问表达式中的数组引用表达式 (§15.10.3)
- 作为 PostfixExpression (§15.14)
- 作为赋值运算符的左操作数 (§15.26)
- 作为 try-with-resources 语句中的 VariableAccess (§14.20.3)

在此上下文中，名称在语法上被分类为 MethodName：

- 在方法调用表达式中"(" 的前面 (§15.12)

在以下上下文中，名称在语法上被分类为 PackageOryTypeName：

- 在限定 TypeName 中"." 的左侧
- 在类型按需导入声明中 (§7.5.2)

在这些上下文中，名称在语法上被归类为 AmbiguousName：

- 在限定 ExpressionName 中"."的左侧
- 在方法调用表达式"("的左侧的最右侧的.的左侧
- 在限定 AmbiguousName 的"."的左侧
- 在注解元素声明的默认值子句 (§9.6.2)
- 元素值对中“=”的右侧 (§9.7.1)
- 方法引用表达式::的左侧 (§15.13)

句法分类的作用是将某些类型的实体限制在表达式的某些部分：

- 字段、参数或局部变量的名称可用作表达式 (§15.14.1)。
- 方法名称只能作为方法调用表达式的一部分出现在表达式中 (§15.12)。
- 类或接口的名称只能作为类字面量（§15.8.2）、限定 this 表达式（§15.8.4）、类实例创建表达式（§15.9）、数组创建表达式（§15.10.1）、强制转换表达式（§14.16）、instanceof 表达式（§5.20.2）、枚举常量（§8.9）或字段或方法的限定名称的一部分出现在表达式中。

- 包的名称只能作为类或接口的限定名称的一部分出现在表达式中。

6.5.2 上下文模糊名称的重新分类

然后将 AmbiguousName 重新分类如下。

如果 AmbiguousName 是由单个标识符组成的简单名称，则：

• 如果标识符出现在声明作用域内（§6.3），表示局部变量、形式参数、异常参数或具有该名称的字段（§14.4、§8.4.1、§8.8.1、§15.27.1、§14.20、§8.3），则 AmbiguousName 将重新分类为 ExpressionName。

• 否则，如果标识符是有效的 TypeIdentifier（§3.8），并且出现在表示具有该名称的类、接口或类型参数的声明作用域内（§8.1、§9.1、§8.4.4、§8.8.4），则模糊名称将重新分类为 TypeName。

• 否则，AmbiguousName 将重新分类为 PackageName。稍后的步骤确定该名称的包是否实际存在。

如果 AmbiguousName 是一个限定名称，由名称、“.”和标识符组成，则首先重新分类“.”左侧的名称，因为它本身就是一个 AmbiguousName。然后有一个选择：

• 如果“.”左侧的名称被重新分类为 PackageName，则：

- 如果标识符是有效的 Typeidentifier，并且有一个包的名称是“.”左边的名称，并且该包包含一个名称与标识符相同的类型声明，则此 AmbiguousName 将重新分类为 TypeName。
- 否则，此 AmbiguousName 将重新分类为 PackageName。稍后的步骤确定该名称的包是否实际存在。

• 如果“.”左侧的名称被重新分类为 TypeName，则：

- 如果标识符是 TypeName 表示的类型的方法或字段的名称，则此 AmbiguousName 将重新分类为 ExpressionName。
- 否则，如果标识符是有效的 Typeidentifier，并且是由 TypeName 表示的类型的成员类型的名称，则此 AmbiguousName 将重新分类为 TypeName。
- 否则，产生一个编译错误。

• 如果“.”左侧的名称重新分类为 ExpressionName，则此 AmbiguousName 将重新分类为 ExpressionName。后面的步骤确定具有名称标识符的成员是否实际存在。

潜在类型名必须是“有效的类型标识符”，这一要求阻止了将 var 和 yield 视为类型名。这通常是多

余的，因为声明的规则已经阻止了名为 var 和 yield 的类型的引入。然而，在某些情况下，编译器可能会找到一个名为 var 或 yield 的二进制类，我们希望澄清的是，这种类永远无法命名。最简单的解决方案是一致地检查有效的 TypeIdentifier。

例子 6.5.2-1. 上下文模糊名称的重新分类

考虑以下人为的“库代码”:

```
package org.rpgpoet;
import java.util.Random;
public interface Music { Random[] wizards = new Random[4]; }
```

然后考虑另一个包中的示例代码:

```
package bazola;
class Gabriel {
    static int n = org.rpgpoet.Music.wizards.length;
}
```

首先，名称 org.rpgpoet.Music.wizards.length 被归类为 ExpressionName，因为它的功能像是 PostfixExpression。因此，以下每一个名字:

```
org.rpgpoet.Music.wizards
org.rpgpoet.Music
org.rpgpoet
org
```

被初始分类为 AmbiguousName。然后将其重新分类:

- 简单名字 org 重新分类为 PackageName (由于作用域中没有名为 org 的变量或类型)。
- 接下来，假设在包 org 的任何编译单元中都没有名为 rpgpoet 的类或接口（我们知道没有此类类或接口，因为包 org 有一个子包名为 rpgpoet），则限定名称 org.rpgpoet 被重新分类为 PackageName。
- 其次，由于包 org.rpgpoet 有一个名为 Music 的可访问（§6.6）接口类型，因此限定名为 org.rpgpoet.Music 被重新分类为 TypeName。
- 最后，因为名称 org.rpgpoet.Music 是一个 ypeName,限定名 org.rpgpoet.Music.wizards 被重新分类为 ExpressionName。

6.5.3 模块名称和包名称的含义

模块名 M（无论是简单的还是限定的）表示具有该名称的模块（如果有的话）。

如果没有可观察到具有该名称的模块，则本节不强制执行编译时错误。相反，模块声明中的 requires 指令（§7.7.1）对模块名称执行自己的验证，而 exports 和 opens 指令（§6.7.2）允许不存在的模块名称。

分类为 PackageName 的名称的含义确定如下。

6.5.3.1 简单包名

如果包名由单个标识符组成，则该标识符必须出现在具有该名称的顶级包的一个声明作用域内（§6.3），并且该包必须对当前模块唯一可见（§7.4.3），否则会发生编译时错误。包名称的含义是该包。

6.5.3.2 限定包名

如果包名的形式为 Q.Id，那么 Q 也必须是包名。包名 Q.Id 命名一个包，该包是由 Q 命名的包中名为 Id 的成员。

如果 Q.Id 没有命名当前模块唯一可见的包（§7.4.3），则会发生编译时错误。

6.5.4 PackageOrTypeNames 的含义

6.5.4.1 简单 *PackageOrTypeNames*

如果 PackageOrTypeName，Q 是有效的 TypeIdentifier，并且出现在名为 Q 的类、接口或类型参数的作用域内，则 PackageOrTypeName 将重新分类为 TypeName。

否则，PackageOrTypeName 将重新分类为 PackageName。PackageOrTypeName 的含义是重新分类名称的含义。

6.5.4.2 限定 *PackageOrTypeNames*

给定形式为 Q.Id 的限定 PackageOrTypeName，如果 Id 是有效的 TypeIdentifier，并且由 Q 表示的类、接口、类型参数或包具有名为 Id 的成员类或接口，则限定的 PackageOrTypeName 名称将重新分类为 TypeName。

否则，它将重新分类为 PackageName。限定 PackageOrTypeName 的含义是重新分类名称的含义。

6.5.5 类型名称的含义

分类为 TypeName 的名称的含义确定如下。

6.5.5.1 简单类型名

如果类型名由单个标识符组成，则标识符必须出现在具有该名称的类、接口或类型参数的一个声明作用域内（§6.3），否则会发生编译时错误。

如果声明表示泛型类或接口 C 的类型参数（§8.1.2，§9.1.1），则以下两项必须为真，否则会发生编译时错误：

- 类型名称不会出现在静态上下文中(§8.1.3)。
- 如果类型名出现在 C 的嵌套类或接口声明中，则类型名的直接封闭类或接口声明是 C 的内部类。

例如，类型名不能出现在 C 声明的静态方法体中，也不能出现在嵌套在 C 中的静态类的实例方法体中。

如果声明表示泛型方法或构造函数 m 的类型参数(§8.4.4，§8.8.4)，并且类型名直接或间接地出现在 m 声明的局部类、局部接口或匿名类 D 的主体中，则下列两个必须为真，否则将发生编译时错误:

- 类型名不会出现在静态上下文中。
- D 是一个内部类，类型名的直接封闭类或接口声明是 D 或 D 的内部类。

例如，类型名不能出现在由 D 声明的静态方法的主体中，也不能出现在 D 的默认方法主体中（如果 D 是局部接口）。

类型名称的含义是作用域内的类、接口或类型参数。

例子 6.6.6.1-1. 对类型参数的引用

```
class Box<T> {
    T val;
    Box(T t) { val = t; }

    static Box<T> empty() {  // compile-time error
        return new Box<>(null);
    }

    static <U> Box<U> make(U val) {
       interface Checker {
          void check(U val); // compile-time error
       }
       class NullChecker implements Checker {
          public void check(U val) {
             if (val == null) {
                throw new IllegalArgumentException();
             }
          }
       }

       new NullChecker().check(val);
       return new Box<U>(val);
    }
}
```

类类型参数 T 在类 Box 的整个声明过程中都处于作用域中;然而，在静态方法 empty 的声明中使

用名称 T 是非法的。

同样，方法类型参数 U 在方法 make 的整个声明过程中都处于作用域中；然而，在局部接口 Checker(隐式静态)的声明中使用名称 U 是非法的。

6.5.5.2 限定类型名

如果类型名是 Q. id 的形式，那么 Q 必须是当前模块唯一可见的包中的类、接口或类型参数的名称，或者是当前模块唯一可见的包的名称(§7.4.3)。

如果 Id 恰好命名了一个可访问的类或接口(§6.6)，它是由 Q 表示的类、接口、类型参数或包的成员，那么限定的类型名表示这个类或接口。

如果 Id 没有在 Q 中命名成员类或接口(§8.5，§9.5)，或者在 Q 中名为 Id 的成员类或接口不可访问，或者 Id 在 Q 中命名了多个成员类或接口，那么就会发生编译时错误。

例子 6.5.5.2-1. 限定类型名

```
class Test {
    public static void main(String[] args){
         java.util.Date date =
            new java.util.Date(System.currentTimeMillis());
         System.out.println(date.toLocaleString();
    }
}
```


这个程序第一次运行时产生了以下输出:

```
Sun Jan 21 22:56:29 1996
```

在本例中，名称 java.util. date 必须表示类型，因此我们首先递归地使用过程来确定是否是 java.util 是一个可访问的类或接口或类型参数，或者一个包，它是，然后我们看看类 Date 在这个包中是否可访问。

6.5.6 表达式名称的含义

被归类为 ExpressionName 的名称的含义如下所示。

6.5.6.1 简单表达式名

如果表达式名由一个标识符组成，那么必须只有一个声明，在标识符出现的点表示局部变量、形式参数、异常参数或作用域中的字段。否则，将出现编译时错误。

如果声明的是 C 类的实例变量(§8.3.1.1)，则下列两个条件都必须为真，否则将发生编译时错误:

- 表达式名不会出现在静态上下文中(§8.1.3).
- 如果表达式名称出现在 C 的嵌套类或接口声明中，则表达式名称的直接封闭类或接口声明是 C 的内部类。

  例如，表达式名称不得出现在 C 声明的静态方法体中，也不得出现在嵌套在 C 中的静态类的实例方法体中。

如果声明表示局部变量、形式参数或异常参数，则将 X 设为最内部的方法声明、构造函数声明、实例初始值、静态初始值、字段声明或包含局部变量或参数声明的显式构造函数调用语句。如果表达式名称直接或间接出现在 X 中直接声明的局部类、局部接口或匿名类 D 的主体中，则以下两项必须为真，否则会发生编译时错误:

- 表达式名称不出现在静态上下文中。
- D 是一个内部类，表达式名称的直接封闭类或接口声明是 D 或 D 的内部类。

  例如，表达式名称不得出现在由 D 声明的静态方法的主体中，也不得出现在 D 的默认方法体中（如果 D 是局部接口）。

如果声明表示的局部变量、形式参数或异常参数既不是 final 的也不是实际上 final 的（§4.12.4），如果表达式名称出现在由 X 直接或间接包围的内部类中，或出现在由 X 包含的 lambda 表达式中（§15.27），则为编译时错误。

这些规则的最终效果是局部变量、形式参数或异常参数只能从其作用域内声明的嵌套类或接口引用，如果(i) 该引用不在静态上下文中，(ii) 从引用到变量声明有一系列内部（非静态）类，以及 (iii) 变量是 final 的或实际上是 final 的。来自 lambda 表达式的引用也要求变量为 final 或实际上为 final。

如果声明声明了一个 final 变量，该变量肯定是在简单表达式之前赋值的，那么名称的含义就是该变量的值。否则，表达式名称的含义就是声明中声明的变量。

如果表达式名称出现在赋值上下文、调用上下文或强制转换上下文中，则表达式名称的类型是捕获转换后字段、局部变量或参数的声明类型（§5.1.10）。

否则，表达式名称的类型是字段、局部变量或参数的声明类型。

也就是说，如果表达式名称出现在“右侧”，则其类型将进行捕获转换。如果表达式名称是出现在“左侧”的变量，则其类型不受捕获转换的约束。

例子 6.5.6.1-1. 简单表达式名

```
class Test {
    static int v;

    static final int f = 3;
```

```
    public static void main(String[] args){
         int i;
         i = 1;
         v = 2;
         f = 33;   // compile-time error
         System.out.println(i + " " + v + " " + f);
    }
}
```

在这个程序中，在 i、v 和 f 的赋值中用作左手边的名称表示局部变量 i、字段 v 和 f 值（不是变量 f，因为 f 是 final 变量）。因此，该示例在编译时产生错误，因为最后一个赋值的左侧没有变量。如果删除了错误的赋值，则可以编译修改后的代码，并生成输出：

```
1 2 3
```

例子 6.5.6.1-2. 对实例变量的引用

```
class Test {
    static String a;
    String b;

    String concat1() {
       return a + b;
    }

    static String concat2() {
       return a + b; // compile-time error
    }

    int index() {
       interface I {
          class Matcher {
             void check() {
                if (a == null ||
                    b == null) { // compile-time error
                    throw new IllegalArgumentException();
                 }
             }
             int match(String s, String t) {
                return s.indexOf(t);
             }
          }
 }
 I.Matcher matcher = new I.Matcher();
 matcher.check();
 return matcher.match(a, b);
    }
 }
```

字段 a 和 b 在整个 Test 类的作用域内。但是，在 concat2 方法的静态上下文中或在不是 Test 内

部类的嵌套类 Matcher 的声明中使用名称 b 是非法的。

例子 6.5.6.1-3. 对局部变量和形式参数的引用

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

          String first = args[0];

          class Checker {
             void checkWhitespace(int x) {
                String arg = args[x];
                if (!arg.trim().equals(arg)) {
                   throw new IllegalArgumentException();
                }
             }

             static void checkFlag(int x) {
                String arg = args[x]; // compile-time error
                if (!arg.startsWith("-")) {
                      throw new IllegalArgumentException();
                 }
             }

             static void checkFirst() {
                Runnable r = new Runnable() {
                public void run() {
                   if (first == null) { // compile-time error
                       throw new IllegalArgumentException();
                   }
                 }
              };
                r.run();
              }
           }

          final Checker c = new Checker();
          c.checkFirst();
          for (int i = 1; i < args.length; i++) {
             Runnable r = () -> {
                c.checkWhitespace(i); // compile-time error
                c.checkFlag(i); // compile-time error
             };
          }
    }
}
```

形式参数 args 在整个 main 方法的作用域内。args 实际上是 final，因此名称 args 可以在局部类 Checker 的实例方法 checkWhitespace 中使用。但是，在局部类 Checker 的 checkFlag 方法的静态上下文中使用名字 args 是非法的。

局部变量 first 在 main 方法主体的其余部分的作用域内。First 实际上是 final。但是，checkFirst 中声明的匿名类不是 Checker 的内部类，因此在匿名类主体中使用名称 first 是非法的。（checkFirst 主体中的 lambda 表达式同样无法引用 first，因为 lambda 表达将出现在静态上下文中。）

局部变量 c 位于 main 方法体的最后几行的作用域内，并被声明为 final，因此可以在 lambda 表达式体中使用名称 c。

局部变量 i 在整个 for 循环的作用域内。但是，i 实际上不是 final，因此在 lambda 表达式体中使用名称 i 是非法的。

6.5.6.2 限定表达式名

如果表达式名称的形式为 Q.Id，则 Q 已被分类为包名称、类型名称或表达式名称。

如果 Q 是包名，则会发生编译时错误。

如果 Q 是命名类类型的类型名称，则：

- 如果类类型中没有一个可访问的成员（§6.6）是名为 Id 的字段，则会发生编译时错误。
- 否则，如果单个可访问成员字段不是类变量（也就是说，它不是静态的），则会发生编译时错误。
- 否则，如果类变量被声明为 final，则 Q.Id 表示类变量的值。

  表达式 Q.Id 的类型是捕获转换后类变量的声明类型（§5.1.10）。

  如果 Q.Id 出现在需要变量而不是值的上下文中，则会发生编译时错误。
- 否则，Q.Id 表示类变量。

  表达式 Q.Id 的类型是捕获转换后类变量的声明类型（§5.1.10）。

  请注意，本条款涵盖了枚举常量的使用（§8.9），因为这些常量始终具有相应的 final 类变量。

如果 Q 是命名接口类型的类型名称，则：

- 如果接口类型中没有一个可访问的成员是名为 Id 的字段，则会发生编译时错误。
- 否则，Q.Id 表示字段的值。

  表达式 Q.Id 的类型是捕获转换后字段的声明类型（§5.1.10）。

  如果 Q.Id 出现在需要变量而不是值的上下文中，则会发生编译时错误。

如果 Q 是表达式名称，则 T 是表达式 Q 的类型：

- 如果 T 不是引用类型，则会发生编译时错误。

- 如果 T 类型中没有一个可访问的成员是名为 Id 的字段，则会发生编译时错误。
- 否则，如果该字段为以下任何一个字段：
  - 接口类型的字段
  - 类类型的 final 字段（可以是类变量或实例变量）
  - 数组类型的 final 字段 length (§10.7)

  则 Q.Id 表示该字段的值，除非它出现在需要变量的上下文中，并且该字段是一个明确未分配的空白 final 字段，在这种情况下，它生成一个变量。

  表达式 Q.Id 的类型是捕获转换后字段的声明类型（§5.1.10）。

  如果 Q.Id 出现在需要变量而不是值的上下文中，并且由 Q.Id 表示的字段被明确分配，则会发生编译时错误。

- 否则，Q.Id 表示变量，T 类的字段 Id，可以是类变量或实例变量。

  表达式 Q.Id 的类型是捕获转换后字段成员的类型（§5.1.10）。

例子 6.5.6.2-1. 限定表达式名

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

class Test {
    public static void main(String[] args){
       int i = 0;
       i.x++;  // compile-time error
       Point p = new Point();
       p.nPoints(); // compile-time error
    }
}
```

此程序遇到两个编译时错误, 因为 int 变量 i 没有成员, 以及 nPoints 不是 Point 类的方法。

例子 6.5.6.2-2. 使用类型名称限定表达式

请注意，表达式名称可以按类型名称限定，但一般不按类型限定。结果是不可能通过参数化类型访问类变量。例如，给定代码：

```
class Foo<T> {
   public static int classVar = 42;
}
```

以下赋值是非法的：

```
Foo<String>.classVar = 91; // illegal
```

相反，可以写成：

```
Foo.classVar = 91;
```

这并不以任何有意义的方式限制 Java 编程语言。类型参数不能用于静态变量的类型，因此参数化类型的类型参数永远不会影响静态变量的类型。因此，不会失去表达能力。类型名 Foo 似乎是原始类型，但不是；相反，它是要访问其静态成员的非泛型类型 Foo 的名称（§6.1）。由于没有使用原始类型，因此没有未检查的警告。

6.5.7 方法名称的含义

分类为 MethodName 的名称的含义确定如下。

6.5.7.1 简单方法名

简单方法名称出现在方法调用表达式的上下文中（§15.12）。简单方法名称由一个 UnqualifiedMethodIdentifier 组成，该标识符指定要调用的方法的名称。方法调用规则要求 UnqualifiedMethodIdentifier 表示在方法调用点的作用域内方法。规则还禁止（§15.12.3）在静态上下文（§8.1.3）或嵌套类或接口（声明实例方法的类或接口的内部类除外）中引用实例方法。

例子 6.5.7.1-1. 简单方法名

下面的程序演示了在确定调用哪个方法时作用域的作用。

```
class Super {
    void  f2(String s)       {}
    void  f3(String s)       {}
    void  f3(int i1, int i2) {}
}

class Test {
    void f1(int i) {}
    void f2(int i) {}
    void f3(int i) {}

    void m() {
        new Super() {
            {
                f1(0);  // OK, resolves  to Test.f1(int)
                f2(0);  // compile-time  error
                f3(0);  // compile-time  error
            }
        };
```

```
        }
    }
```

对于调用 f1(0), 只有一个方法名 f1 在作用域内。这是方法 Test.f1(int), 其声明在整个 Test 体（包括匿名类声明）的作用域内。§15.12.1 选择在类 Test 中搜索，因为匿名类声明没有名为 f1 的成员。最终, Test.f1(int)被解析。

对于调用 f2(0), 两个名为 f2 的方法在作用域内。首先，方法声明 Super.f2(String) 在整个匿名类声明中都在作用域内。其次，方法声明 Test.f2(int)在包括匿名类声明的整个 Test 体中都在作用域内。(请注意，两个声明都不遮蔽另一个，因为在每个声明的点上，另一个不在作用域内。) §15.12.1 选择在类 Super 中搜索因为它有一个名为 f2 的成员。然而，Super.f2(String) 不适用于 f2(0),因此会发生编译时错误。注意 Test 类没有被搜索。

对于调用 f3(0), 作用域内有三个名为 f3 的方法。首先和其次，Super.f3(String)和 Super.f3(int,int)方法的声明在整个匿名类声明的作用域内。第三，Test.f3(int) 方法的声明在包括匿名类声明的整个 Test 体的作用域内。§15.12.1 选择在类 Super 中搜索因为它有一个名为 f3 的成员。然而，Super.f3(String) 和 Super.f3(int,int) 不适用于 f3(0), 因此会发生编译时错误。注意 Test 类没有被搜索。

选择在词法包围作用域之前搜索嵌套类的超类层次结构称为“梳子规则” (§15.12.1)。

6.6 访问控制

Java 编程语言提供了访问控制机制，以防止包或类的用户依赖于该包或类实现的不必要细节。如果允许访问，则被访问的实体称为可访问。

请注意，可访问性是一个静态属性，可以在编译时确定；它仅取决于类型和声明修饰符。

限定名称是访问包、类、接口、类型参数和引用类型成员的一种手段。当此类成员的名称从其上下文（§6.5.1）分类为限定类型名称（表示包、类、接口或类型参数的成员）或限定表达式名称（表示引用类型的成员）时，将应用访问控制。

例如，单个类型导入声明使用限定类型名称（§7.5.1），因此必须可以从包含导入声明的编译单元访问命名类或接口。另一个例子是，类声明可能会对超类类型使用限定类型名（§8.1.5），因此，命名类也必须是可访问的。

§6.5.1 中的上下文分类中“缺少”了一些明显的表达式：基本类的字段访问（§15.11.1）、基本类的方法调用（§15.12）、基本类中的方法引用（§15.13）以及限定类实例创建中的实例化类（§15.9）。出于§6.2 中给出的原因，这些表达式均使用标识符而不是名称。因此，通过字段访问表达式、方法调用表达式、方法引用表达式和限定类实例创建表达式显式应用对成员（无论是字段、方法、类还是接口）的访问控制。(注意，对字段的访问也可以用后缀表达式表示的限定名来表示。)

此外，许多语句和表达式允许使用非专用于类型名的类型。例如，类声明可以使用参数化类型

（§4.5）来表示超类类型。由于参数化类型不是限定的类型名，因此类声明必须显式地对所表示的超类执行访问控制。因此，在§6.5.1 中提供上下文以分类 TypeName 的语句和表达式中，大多数都执行自己的访问控制检查。

除了访问包、类、接口或类型参数的成员之外，还有访问类的构造函数的问题。当显式或隐式调用构造函数时，必须检查访问控制。因此，访问控制通过显式构造函数调用语句（§8.8.7.1）和类实例创建表达式（§15.9.3）进行检查。此类检查是必要的，因为§6.5.1 未提及显式构造函数调用语句(因为它们间接引用构造函数，而不是通过名称) 并且不知道由非限定类实例创建表达式表示的类与该类的构造函数之间的区别。此外，构造函数没有限定名称，因此我们不能依赖于在限定类型名称分类期间检查的访问控制。

可访问性影响类成员的继承（§8.2），包括隐藏和方法重写（§8.4.8.1）。

6.6.1 确定可访问性

- 如果顶级类或接口（§7.6）被声明为公共类或接口，并且是由模块导出的包的成员，则该类或接口可以由同一模块中的任何代码访问，也可以由导出该包的另一模块的任何代码来访问，前提是声明该类或接口的编译单元对该另一模块可见（§7.3）。
- 如果顶级类或接口被声明为公共的，并且是未由模块导出的包的成员，则该类或接口可以由同一模块中的任何代码访问。
- 如果使用包访问来声明顶级类或接口，则只能从声明它的包中访问它。

  未使用访问修饰符声明的顶级类或接口隐式具有包访问权限。
- 类的成员（类、接口、字段或方法）、接口、类型参数或引用类型，或类的构造函数，只有在以下情况下才可访问：(i) 类、接口、类型参数或引用类型是可访问的，并且(ii) 声明成员或构造函数允许访问：

- 如果成员或构造函数声明为公共，则允许访问。

  缺少访问修饰符的接口的所有成员都是隐式公共的。
- 否则，如果声明该成员或构造函数受保护级别，则仅当以下任一项为真时才允许访问：

  > 从包含声明受保护成员或构造函数的类的包中访问成员或构造函数。

  > 如 §6.6.2 所述，访问是正确的。
- 否则，如果使用包访问来声明成员或构造函数，则只有在从声明类、接口、类型参数或引用类型的包中进行访问时，才允许访问。

未使用访问修饰符声明的类成员或构造函数隐式具有包访问权限。

- 否则，该成员或构造函数将被声明为私有。只有当下列情况之一为 true 时，才允许访问:
  - 访问发生在包含成员或构造函数声明的顶级类或接口的主体内。
  - 访问发生在包含成员声明的顶级类或接口的 permits 子句中。
  - 访问发生在包含成员声明的顶级记录类的记录组件列表中。

• 当且仅当数组类型的元素类型可访问时，数组类型才可访问。

例子 6.6-1. 访问控制

考虑两个编译单元:

```
package points;
class PointVec { Point[] vec; }
```

以及:

```
package points;
   public class Point {
      protected int x, y;
      public void move(int dx, int dy) { x += dx; y += dy; }
      public int getX() { return x; }
      public int getY() { return y; }
}
```


在包 points 里声明了两个类类型:

- 类类型 PointVec 不是 public 的，也不是包 points 的 public 接口的一部分, 而是只能由包中的其他类使用。
- 类类型 Point 被声明为公共的，并且可用于其他包。它是 points 包的公共接口的一部分。
- 类 Point 的 move、getX 和 getY 方法被声明为公共的，因此任何使用类型 Point 的对象的代码都可以使用。
- 字段 x 和 y 被声明为受保护的，并且只有在类 Point 的子类中，并且只有当它们是由访问它们的代码实现的对象的字段时，才可以在 points 包之外访问。

  有关受保护访问修饰符如何限制访问的示例，请参见§6.6.2。

例子 6.6-2. 访问公共字段、方法和构造函数

公共类成员或构造函数可在其声明的整个包以及任何其他包中访问，前提是声明它的包是可观察的（§7.4.3）。例如在编译单元中:

```
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
        moves++;
    }
    public static int moves = 0;
}
```

公共类 Point 有 move 方法和 moves 字段作为公共成员。这些公共成员可以访问任何其他可以访问 points 包的包。字段 x 和 y 不是公共的，因此只能从 points 包内访问。

例子 6.6-3. 访问公共和非公共类

如果类缺少公共修饰符，则对类声明的访问仅限于声明该类的包（§6.6）。在该示例中：

```
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
class PointList {
    Point next, prev;
}
```

在编译单元中声明了两个类。Point 类在 points 包之外可用，而 PointList 类仅可在包内访问。因此，另一个包中的编译单元可以访问 points.Point，通过使用其完全限定名称：

```
package pointsUser;
class Test1 {
    public static void main(String[] args){
       points.Point p = new points.Point();
       System.out.println(p.x + " " + p.y);
    }
}
```

或者通过使用提及完全限定名称的单一类型导入声明（§7.5.1），以便此后可以使用简单名称：

```
package pointsUser;
import points.Point;
class Test2 {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y);
    }
}
```

但是，此编译单元不能使用或导入 points.PointList，其未被声明为公共的，因此在 points 包之外是不可访问的。

例子 6.6-4. 使用包访问权限访问字段、方法和构造函数

如果未指定访问修饰符 public、protected 或 private，则类成员或构造函数具有包访问权限：它在包含声明类成员的类的声明的整个包中都是可访问的，但在任何其他包中都无法访问类成员或构造函数。

如果公共类有一个具有包访问权限的方法或构造函数，则此方法或构造函数不能被此包外部声明的子类访问或继承。

例如，如果我们有：

```
package points;
public class Point {
    public int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    public void moveAlso(int dx, int dy) { move(dx, dy); }
}
```

然后，另一个包中的子类可以声明一个不相关的 move 方法，具有相同的签名（§8.4.2）和返回类型。由于无法从包 morepoints 访问原始 move 方法，因此可能无法使用 super：

```
package morepoints;
public class PlusPoint extends points.Point {
   public void move(int dx, int dy) {
      super.move(dx, dy); // compile-time error
      moveAlso(dx, dy);
   }
}
```

因为 Point 类的 move 函数没有被 PlusPoint 类的 move 函数重写，Point 类的方法 moveAlso 从来不会调用 PlusPoint 类的方法 move。因此如果你从 PlusPoint 的调用中删除 super.move 然后执行测试程序：

```
import points.Point;
import morepoints.PlusPoint;
class Test {
    public static void main(String[] args) {
        PlusPoint pp = new PlusPoint();
        pp.move(1, 1);
    }
}
```

它正常终止。如果 Point 类的 move 方法被 PlusPoint 里的 move 方法重写，那么该程序将无限递归，直到发生 StackOverflower 错误。

例子 6.6-5. 访问私有字段、方法和构造函数

私有类成员或构造函数只能在包含成员或构造函数声明的顶级类（§7.6）的主体内访问。它不是由子类继承的。在该示例中：

```
class Point {
    Point() { setMasterID(); }
    int x, y;
    private int ID;
    private static int masterID = 0;
    private void setMasterID() { ID = masterID++; } }
```

私有成员 ID, masterID,和 setMasterID 只能在类 Point 体内使用。在 Point 声明的主体之外，不能通过限定名称、字段访问表达式或方法调用表达式访问它们。

有关使用私有构造函数的示例，请参见§8.8.10。

6.6.2 受保护访问的详细信息

对象的受保护成员或构造函数可以从包外部访问，在包中，它仅由负责该对象实现的代码声明。

6.6.2.1 访问受保护的成员

设 C 是声明受保护成员的类。仅允许在 C 的子类 S 的主体内访问。

子类 S 被视为负责实现类 C 的对象。根据 C 的可访问性，S 可以与 C 在同一个包中声明，也可以与 C 声明在同一模块的不同包中，或者在不同模块的包中声明。

此外，允许根据限定名称、字段访问表达式（§15.11）、方法调用表达式（§14.12）或方法引用表达式（§5.13）的形式访问实例字段或实例方法:

- 如果访问是通过(i) 形式为 ExpressionName.Id 或 TypeName.Id 的限定名, 或者(ii)形式为 Primary.Id 的字段访问表达式, 那么访问实例字段 Id 是被允许的，当且仅当限定类型是 S 或 S 的子类。

  限定类型是 ExpressionName 或 Primary 的类型，或者由 TypeName 表示的类型。

- 如果访问是通过(i)形式为 ExpressionName.Id(...) 或 TypeName.Id(...) 或 Primary.Id(...)的方法调用表达式, 或者(ii)形式为 ExpressionName :: Id 或 Primary :: Id 或 ReferenceType :: Id 的方法引用表达式, 那么访问实例方法 Id 是被允许的当且仅当限定类型是 S 或 S 的子类。

  限定类型是 ExpressionName 或 Primary 的类型，或者由 TypeName 或 ReferenceType 表示的类型。

  有关访问受保护成员的更多信息，请参阅 Alessandro Coglio 在 2005 年 10 月《对象技术杂志》上发表的《Java 虚拟机中检查受保护成员的访问》。

6.6.2.2 访问受保护的构造函数

设 C 为声明受保护构造函数的类，设 S 为最内层类，在其声明中使用受保护构造函数。那么:

- 如果访问是通过超类构造函数调用 super(…)或限定的超类构造函数访问 E.super(…)，其中 E 是 Primary 表达式，则允许访问。

- 如果通过匿名类实例创建表达式 new C(…){…}或限定的匿名类实例创建表达式 E.new C(…){…}进行访问，其中 E 是 Primary 表达式，则允许访问。
- 如果通过简单类实例创建表达式 new C(...)或限定类实例创建表达式 E.new C(...)（其中 E 是 Primary 表达式）或方法引用表达式 C :: new（其中 C 是 ClassType）进行访问，则不允许访问。受保护的构造函数只能从定义它的包中由类实例创建表达式（不声明匿名类）或方法引用表达式访问。

例子 6.6.2-1. 访问受保护的字段、方法和构造函数

考虑这个例子，points 包声明:

```
package points;
public class Point {
    protected int x, y;
    void warp(threePoint.Point3d a) {
        if (a.z > 0)  // compile-time error: cannot access a.z
            a.delta(this);
    }
}
```

threePoint 包声明:

```
package threePoint;
import points.Point;
public class Point3d extends Point {
   protected int z;
   public void delta(Point p) {
        p.x += this.x; // compile-time error: cannot access p.x
        p.y += this.y; // compile-time error: cannot access p.y
    }
    public void delta3d(Point3d q) {
        q.x += this.x;
        q.y += this.y;
        q.z += this.z;
    }
}
```

这里的方法 delta 中出现编译时错误：它无法访问其参数 p 的受保护成员 x 和 y，因为虽然 Point3d（引用字段 x 和 y 的类）是 Point 的子类（声明 x 和 y 所在的类），但它不参与 Point（参数 p 的类型）的实现。方法 delta3d 可以访问其参数 q 的受保护成员，因为类 Point3d 是 Point 的子类，并且参与了 Point3d 的实现。

方法 delta 可以尝试将其参数强制转换（§5.5，§15.16）为 Point3d，但如果运行时的 p 类不是 Point3d，则该转换将失败，导致异常。

方法 warp 中也会出现编译时错误：它无法访问其参数 a 的受保护成员 z，因为虽然类 Point（字段 z 引用所在的类）参与了 Point3d（参数 a 的类型）的实现，但它不是 Point3d（声明 z 的类）的子类。

6.7 完全限定名和规范名

每个原生类型、命名包、顶级类和顶级接口都有一个完全限定的名称:

- 原生类型的完全限定名称是该原生类型中的关键字，即 byte, short, char, int, long, float, double,或 boolean。
- 不是命名包的子包的命名包的完全限定名称是它的简单名称。
- 一个命名包的完全限定名是另一个命名包的子包，它由包含包的完全限定名，后跟".", 再后跟子包的简单(成员)名组成。
- 在未命名包中声明的顶级类或顶级接口的完全限定名称是类或接口的简单名称。
- 在命名包中声明的顶级类或顶级接口的完全限定名称由包的完全限定名称，后跟".", 再后跟类或接口的简单名称组成。

每个成员类、成员接口和数组类型可以有一个完全限定的名称:

- 当且仅当 C 具有完全限定名称时，另一个类或接口 C 的成员类或成员接口 M 具有完全限定名称。

  在这种情况下，M 的完全限定名由 C 的完全限定名，后跟".", 再后跟 M 的简单名组成。

- 当且仅当数组的元素类型具有完全限定名称时，数组类型具有完全限定名称。

  在这种情况下，数组类型的完全限定名由数组类型的组件类型的完全限定名后跟"[]"组成。

局部类、局部接口或匿名类没有完全限定名称。

每个原生类型、命名包、顶级类和顶级接口都有一个规范名称:

- 对于每个原生类型、命名包、顶级类和顶级接口，规范名称与完全限定名称相同。

每个成员类、成员接口和数组类型可以有一个规范名称:

- 在另一个类或接口 C 中声明的成员类或成员接口 M 具有规范名称，当且仅当 C 具有规范名称。

  在这种情况下，M 的规范名称由 C 的规范名称，后跟".", 再后跟 M 的简单名

称组成。

- 当且仅当数组的组件类型具有规范名称时，数组类型才具有规范名称。

  在这种情况下，数组类型的规范名称由数组类型的组件类型的规范名称后跟“[]”组成。

局部类、局部接口或匿名类没有规范名称。

例子 6.7-1. 完全限定名

- 类型 long 的完全限定名是"long"。
- 包 java.lang 的完全限定名是"java.lang"，因为它是包 java 的子包 lang。
- 类 Object 的完全限定名, 定义在包 java.lang 里, 是 "java.lang.Object"。
- 接口 Enumeration 的完全限定名, 定义在包 java.util 里, 是 "java.util.Enumeration"。
- 类型"array of double" 的完全限定名是"double[]"。
- 类型"array of array of array of array of String" 的完全限定名是"java.lang.String[][][][]"。

在代码中:

```
package points;
class Point { int x, y; }
class PointVec { Point[] vec; }
```

类型 Point 的完全限定名是"points.Point";类型 PointVec 的完全限定名是 "points.PointVec"; 类 PointVec 的字段 vec 的类型的完全限定名是 "points.Point[]"。

例子 6.7-2. 完全限定名称与规范名称

完全限定名和规范名之间的区别可以在如下代码中看到:

```
package p;
class O1 { class I {} }
class O2 extends O1 {}
```

p.O1.I 和 p.O2.I 都是完全限定名表示成员类 I,但只有 p.O1.I 是它的规范名。